GUIDE: Towards Progressive Visual Query Autocompletions for Graph Databases

Zhaoyu Chen[1,2,†], Zixu Li[1,2,†], Xi He[2], Kai Huang[1,3,§]
[1] East China Normal University, Shanghai, China
[2] Macau University of Science and Technology, Macau SAR
[3] Shanghai Engineering Research Center on Big Data Management System, Shanghai, China
zhaoyvc536@gmail.com, lzixu00@gmail.com, hexichn@gmail.com, kylehuangk@gmail.com

***Abstract*—The growing prevalence of graph data in domains such as bioinformatics necessitates intuitive query formulation methods. While visual query interfaces mitigate the inherent complexity of graph query languages, constructing queries remains challenging. Current Graph Query Autocompletion (GQAC) systems assist users in iteratively building query graphs through visual interfaces by autocompleting partial queries, thereby reducing cognitive load. However, they suffer critical limitations such as tedious incremental suggestions (1-2 edges per iteration) and reliance on days-long preprocessing that fails for dynamic data. We propose GUIDE, an online visual query autocompletion framework that eliminates preprocessing by operating directly on the underlying database. Unlike existing systems constrained to suggesting small substructures, GUIDE generates suggestions of arbitrary size. This approach significantly reduces iterative interactions and cognitive burden, accelerating complex query formulation. We further introduce novel optimizations to ensure efficient autocompletion. Extensive experiments demonstrate GUIDE's significant superiority over traditional techniques.**

***Index Terms*—Graph, Graph Query Autocompletion, Visual Query Interfaces**

I. Introduction

The rapid proliferation of graph-structured data across domains such as biological databases, chemical compounds, social networks, and knowledge graphs has significantly amplified the demand for intuitive and user-friendly graph querying mechanisms. Despite the development of expressive graph query languages like SPARQL, Cypher, and Gremlin, crafting graph queries using these languages often imposes a steep learning curve and requires programming proficiency that many domain experts—such as chemists or biologists—may not possess.

To bridge this usability gap, *visual graph query interfaces* (*i.e.,* GUI) have been developed to enable users to construct queries through *interactive graph drawing* rather than complex syntax. Tools and platforms such as *PubChem Sketcher* [1], *ChemSpider* [2], and academic systems like AURORA [24] demonstrate the growing demand and practical relevance of such interfaces. However, even with visual support, formulating queries remains cognitively demanding, particularly when users must explicitly define intricate topological relationships within large or unfamiliar graph structures. To mitigate this burden, ***Graph Query Autocompletion (GQAC)*** has emerged as a promising solution. GQAC systems [1], [2] assist users during the query construction process by automatically suggesting top-k relevant query fragments [3] based on the partially constructed query (*i.e.,* partial query). This not only reduces cognitive effort but also supports exploratory search, allowing non-expert users to iteratively refine and expand their queries while engaging more effectively with complex graph data.

However, existing GQAC systems such as AutoG [1] and GFocus [2] suffer from key practical limitations. First, their incremental query construction is tedious—each suggestion adds only minimal structures (*e.g.,* 1-2 edges), forcing users through repetitive iterations to build complex queries, resulting in high cognitive and computational overhead. More critically, both systems rely on exhaustive preprocessing, requiring hours or days to index moderate-sized databases. For example, AutoG requires nearly 10 days to index the PubChem database. Given the dynamic nature of graph databases, indexes must be rebuilt as the database evolves, rendering existing systems impractical for dynamic scenarios. Moreover, these systems frequently suggest theoretically valid but non-existent query patterns, misleading users into constructing empty-result searches. In addition, both AutoG and GFocus suffer from a cold-start problem by invariably suggesting identical initial queries regardless of the target query.

To address these limitations of existing GQAC systems, we propose GUIDE (proGressive visual qUery autocompletIons for graph DatabasEs), a novel framework for efficient visual query autocompletion in graph databases. Unlike existing approaches requiring index construction, GUIDE operates directly on the graph database using an online query suggestion mechanism. This eliminates preprocessing overhead and enables instant adaptation to database changes. Specifically, GUIDE employs Partial-query Occurrence Localization (POLocate, Section III-A) to retrieve occurrences of a partial query. Based on these occurrences, it generates candidate suggestions via the Candidate Suggestions Generation process (CSGenerate, Section III-B). GUIDE then ranks

†Equal contributions. §Kai Huang is the corresponding author.
[1]https://pubchem.ncbi.nlm.nih.gov//edit3/index.html
[2]https://www.chemspider.com/StructureSearch

[3]A fragment refers to a graph structure containing a partial query. Throughout this paper, the terms "query fragment" and "query suggestion" are used interchangeably.

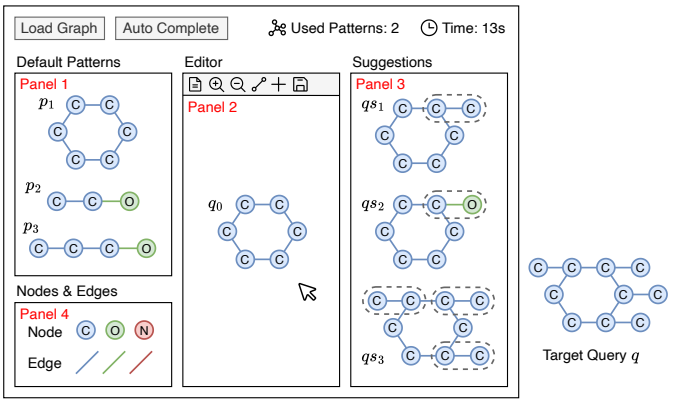


Fig. 1: The sketch of GUIDE interface for visual graph query formulation.

each candidate suggestion and selects final query suggestions using an evaluation function (Query Suggestions Selection (QSSelect), Section III-C). Crucially, final query suggestions can be of any size, significantly accelerating the overall query formulation process and greatly reducing the number of iterative interactions required. Furthermore, since suggestions are derived directly from the underlying graph database, users are effectively guided away from constructing empty-result searches. To mitigate the cold-start problem, GUIDE provides default visual patterns on the GUI, enabling users to visually formulate initial queries. Additionally, several optimizations (see Section IV) are incorporated to handle larger datasets and enhance the autocompletion performance.

Example 1. *Fig. 1 illustrates the sketch of the GUIDE interface for visual graph query formulation. Given a target query q (as shown in Fig. 1), a user can drag-and-drop predefined patterns from Panel 1 and nodes/edges from Panel 4 into the visual editor (i.e., Panel 2) to construct an initial query q_0. In this example, the user can drag the pattern p_1 from Panel 1 into Panel 2 to form the initial query q_0, which is a partial query with respect to the query q, since p_1 is a subgraph of the target query.*

While the user can continue to manually build the remaining part of q by repeatedly adding nodes and edges from Panel 4 to q_0, this process can become time-consuming for larger query graphs. To streamline this, GUIDE provides an "Auto Complete" button at the top of the interface, which, when clicked, generates several query fragment suggestions (e.g., qs_1, qs_2, and qs_3; see Panel 3). Each suggestion is a supergraph of the partial query q_0 and a subgraph of the target query q. In this example, the user can select qs_3 to directly formulate the target query q. As such, only one iteration (i.e., clicking the "Auto Complete" button once) is required to construct the complete query. In contrast, existing systems typically require more iterative interactions and impose a higher cognitive burden to formulate complex queries, as they often suggest only 1-edge or 2-edge increments per iteration (e.g., *the parts highlighted by the dashed cycles in Panel 3).*

TABLE I: List of key notations.

Notation	Description
g, D	A data graph, a graph database
$g = (V, E, l)$	V: Vertex set, E: Edge set, l: Label function
q, q_i	A query graph, a query graph at i-th iteration
$\mathcal{G}$	Set of supergraphs, where $\mathcal{G} \in D$
σ_{min}, σ_{max}	user-defined min/max increment sizes
cs	Candidate suggestion mined from supergraph $\mathcal{G}$, where $q \subseteq_\lambda cs$ and $cs \subseteq_\lambda \mathcal{G}$
$\mathcal{CS}$	A set of candidate suggestions
m	Mapping between q and $\mathcal{G}$
$\mathcal{M}$, $\mathcal{M}_i$	A mapping set, a mapping set at i-th iteration
k	Number of query suggestions
qs	An individual query suggestion candidate selected from $\mathcal{CS}$ to be added to the $\mathcal{QS}$ collection
$\mathcal{QS}$	Top-k query suggestions
$\text{Cov}(\mathcal{QS}, D)$	Edge coverage function, counting unique edges in D covered by $\mathcal{QS}$

Experimental results demonstrate that our framework significantly outperforms existing methods in terms of query construction efficiency. Compared to AutoG and GFocus, GUIDE enables each autocompletion iteration to introduce, on average, twice as many new nodes and edges. In terms of structural expansion, each accepted suggestion in GUIDE contributes at least 50% more structural content. Overall, our framework allows users to construct complex graph queries in fewer steps, with greater structural impact per interaction, resulting in a more efficient and streamlined query formulation process. Our main contributions are summarized as follows.

- We propose GUIDE, a novel framework for visual query autocompletion in graph databases. GUIDE operates directly on the database through an online mechanism, eliminating exhaustive preprocessing requirements and enabling instant adaptation to data changes.
- Unlike existing systems limited to suggesting minimal structures, GUIDE generates suggestions of any size. This drastically reduces iterative interactions and cognitive load, accelerating complex query formulation.
- We develop key optimizations to enhance autocompletion performance and solve the cold-start problem via default visual patterns for initial queries.
- We demonstrate GUIDE's superiority through extensive experiments on real-world datasets.

The remainder of this paper is organized as follows. Section II provides preliminary concepts. Section III presents our framework. Section IV discusses our optimization strategies. Section V reports experimental results. Section VI reviews related work, and Section VII concludes the paper. Formal proofs of theorems are in the Appendix.

II. Preliminaries

In this section, we present the fundamental concepts and formalize the problem addressed in this paper. Key notations and acronyms are summarized in Table I.

A. Subgraph Queries and Matching

A graph database D is defined as a collection of data graphs, *i.e.,* $\{g_1, g_2, \ldots, g_n\}$, where each graph $g = (V, E, l)$ consists of a set of vertices V, a set of edges E, and a label function l that assigns labels to both vertices and edges. The size of a graph is typically measured by $|E|$.

Definition 1 (Subgraph Isomorphism)**.** *Given two graphs $g = (V, E, l)$ and $g' = (V', E', l')$, we say that g is a subgraph of g', denoted $g \subseteq_\lambda g'$, if there exists an injective mapping $\lambda : V \to V'$ such that:*

1) *For every $u \in V$, $\lambda(u) \in V'$ and $l(u) = l'(\lambda(u))$;*
2) *For every $(u, v) \in E$, $(\lambda(u), \lambda(v)) \in E'$ 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 subgraph query is to find $\{g \mid q \subseteq_\lambda g, g \in D\}$,* i.e., *all graphs in the database that contain q as a subgraph.*

In this work, we focus on non-induced subgraph isomorphism, which only requires that the query's labeled nodes and edges exist in the target graph without enforcing the absence of additional edges between matched vertices.

Example 2. *Consider a graph database D storing chemical compounds, such as a triangle $g = (V, E, l)$, where $V = \{v_1, v_2, v_3\}$, $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$, and all vertices are labeled "C" for carbon. Suppose a researcher queries with a carbon chain of 3 carbons $q = (V_q, E_q, l_q)$, where $V_q = \{u_1, u_2, u_3\}$, $E_q = \{(u_1, u_2), (u_2, u_3)\}$, and all vertices are also labeled "C". If there exists an injective mapping $\lambda : V(q) \to V(g)$, such as $\lambda(u_1) = v_1$, $\lambda(u_2) = v_2$ and $\lambda(u_3) = v_3$, that preserves both node labels and adjacency (i.e., for every $(u_i, u_j) \in E_q$, $(\lambda(u_i), \lambda(u_j)) \in E$), then q is a subgraph of g under λ, denoted $q \subseteq_\lambda g$.*

B. Graph Query Autocompletion

Graph Query Autocompletion (GQAC) aims to facilitate visual query formulation by suggesting relevant query fragments that users may want to add to their partially constructed queries. This capability has become increasingly important as graph databases grow in complexity and users seek more intuitive ways to explore graph-structured data.

Definition 3 (Graph Query Autocompletion (GQAC))**.** *Given an integer k, a partial query graph q, a graph database D, and the user preference (*i.e., *the size of minimum (resp. maximum) increment σ_{min} (resp. σ_{max})), graph query autocompletion generates top-k suggestions $\mathcal{QS} = \{qs_1, qs_2, ..., qs_k\}$, where each suggestion $qs_j = q + \Delta q_j$ represents the query q extended by an increment Δq_j ($\sigma_{min} \leq |\Delta q_j| \leq \sigma_{max}$). The goal is to maximize the likelihood that users will find useful suggestions within the top-k results.*

AutoG [1] and GFocus [2] are two representative GQAC systems.

C. AutoG: The First Graph Query Autocompletion System

AutoG [1] pioneered the systematic study of graph query autocompletion by introducing a approach that suggests query increments based on frequent structural patterns in the underlying database. The key innovation of AutoG lies in its use of c-prime features as the logical units for query expansion.

Definition 4 (c-prime Features)**.** *A c-prime feature is a frequent subgraph whose composability (i.e., the number of ways it can be composed from smaller features) is no more than a threshold c. These features serve as building blocks for query suggestions while maintaining computational efficiency.*

By leveraging frequent structural patterns as building blocks, AutoG transforms the complex task of suggesting arbitrary graph fragments into a more structured process of combining pre-identified features. Specifically, AutoG's workflow operates as follows:

- *Feature extraction*. AutoG extracts c-prime features from the graph database, which are frequent subgraphs with limited composability. These features serve as reusable structural units for downstream suggestions.
- *Query decomposition*. The current partial query q_0 is represented using c-prime features, allowing the system to track coverage and identify potential extension points.
- *Suggestion generation*. For each c-prime feature f not yet present in q_0, AutoG computes possible ways to compose q_0 with f, yielding a set of candidate completions.
- *Suggestion ranking*. These candidate suggestions are ranked based on selectivity and structural diversity, ensuring both relevance and informativeness for users.

D. GFocus: User Focus-based Autocompletion

Observe that the suggestions generated by AutoG can be extended anywhere in the partial query. This "user focus-unaware" strategy may adversely impact not only the quality of the suggestions but also the response time of GQAC. GFocus [2] addresses this limitation of generating suggestions at arbitrary locations by introducing the concept of user focus—a subgraph of the query that represents the user's current area of attention during query formulation.

Definition 5 (User Focus [2])**.** *The user focus $F \subseteq q_0$ is a connected subgraph of the current query q_0 that has the highest normalized attention weight, computed based on temporal and structural locality principles.*

GFocus models user attention through two complementary locality principles to automatically identify and track where users are focusing their efforts:

(i) *Temporal locality*: More recently manipulated query elements receive higher attention weights.
(ii) *Structural locality*: Attention propagates to neighboring structures around actively manipulated elements.

By constraining suggestions to the identified focus area, GFocus achieves several key advantages over AutoG. The system significantly reduces cognitive load by limiting suggestions to where users are actively working, preventing the

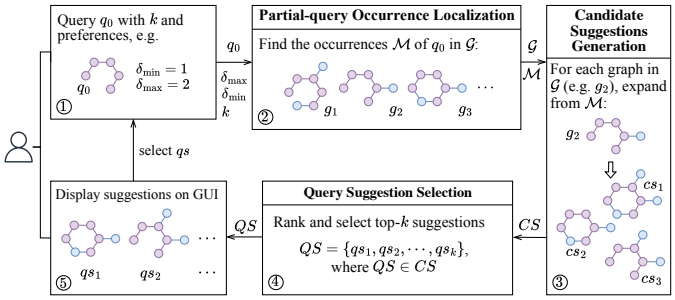


Fig. 2: Overview of the GUIDE framework.

overwhelming effect of having suggestions appear at arbitrary locations throughout the query. This focused approach also improves computational efficiency, as the constrained search space enables faster suggestion generation compared to exploring all possible extension points. Most importantly, suggestions generated at the focus area are more likely to align with user intent, as they naturally extend from the subgraph where users have demonstrated recent interest through their interactions.

E. Limitations of Existing Approaches

Despite the advances made by AutoG and GFocus, both AutoG [1] and GFocus [2] suffer from key practical limitations. First, their incremental query construction is tedious—each suggestion adds only minimal structures (*e.g.,* 1-2 edges), forcing users through repetitive iterations to build complex queries, resulting in high cognitive and computational overhead. More critically, both systems rely on exhaustive preprocessing, requiring hours or days to index moderate-sized databases. For example, AutoG requires nearly 10 days to index the PubChem database. Given the dynamic nature of graph databases, indexes must be rebuilt as the database evolves, rendering existing systems impractical for dynamic scenarios. Moreover, these systems frequently suggest theoretically valid but non-existent query patterns, misleading users into constructing empty-result searches. Finally, they suffer from a cold-start problem by invariably suggesting identical initial queries regardless of the target query.

III. GUIDE: A Novel Graph Autocompletion Framework

In this section, we present a novel framework called GUIDE for efficient visual query autocompletion in graph databases. GUIDE addresses the aforementioned limitations as follows. Unlike existing approaches requiring index construction, GUIDE operates directly on the graph database using an online iterative query suggestion mechanism, which is outlined in Figure 2. This eliminates preprocessing overhead and enables instant adaptation to database changes. Specifically, GUIDE employs Partial-query Occurrence Localization (step ②, Figure 2) to retrieve occurrences of a partial query. Based on these occurrences, it generates candidate suggestions via the Candidate Suggestions Generation process (step ③). GUIDE then ranks each candidate suggestion and selects final query suggestions using an evaluation function (Query Suggestions Selection (step ④). Crucially, final query suggestions can be of any size, significantly accelerating the overall query formulation process and greatly reducing the number of iterative interactions required. Furthermore, since suggestions are derived directly from the underlying graph database, users are effectively guided away from constructing empty-result searches. To mitigate the cold-start problem, GUIDE provides default visual patterns on the GUI, enabling users to visually formulate initial queries. Additionally, several optimizations are incorporated to handle larger datasets and enhance the autocompletion performance.

Algorithm 1 outlines the GUIDE framework. Given an integer k, a partial query graph q, a graph database D, and user preferences—specifically, the minimum and maximum allowed increment sizes $\sigma_{\min}$ and $\sigma_{\max}$, respectively—the framework proceeds as follows.

It first generates a set of default patterns (Line 1)—diversified subgraphs extracted from the graph database using the TED algorithm [4]. Users can select and combine these default patterns to formulate the initial partial query q_0 (Line 3). Next, the framework performs Partial-query Occurrence Localization (POLocate, Line 5) to establish mappings between the current query graph q_i and the database D, resulting in a mapping set $\mathcal{M}$. During the candidate generation phase (Lines 9–10), the algorithm processes each mapping $m \in \mathcal{M}$ while patterns remain extendable, collecting all potential extension patterns into the candidate set $\mathcal{CS}$ via the CSGenerate procedure. For each candidate pattern $cs \in \mathcal{CS}$ (Lines 11–14), the algorithm checks whether it satisfies the size constraints (Line 12) and updates the top-k heap using QSSelect (Line 13). After processing, the algorithm displays the top suggestions (Line 15) and obtains user feedback (Line 16), iterating until the user is satisfied. Finally, it returns the final query suggestions $\mathcal{QS}$ (Line 17).

Lemma 1. *The time complexity of GUIDE (Algorithm 1) is* $\mathcal{O}\Big(|D| \cdot 2^{\mathcal{O}(|V(q_i)|)} + |\mathcal{M}| \cdot \delta_{\max} + |\mathcal{CS}| \cdot \log k\Big)$*, where* $|D|$ *is the number of graphs in the database,* $|V(q_i)|$ *is the number of vertices in query graph* q_i*,* $|\mathcal{M}|$ *is the number of subgraph matches,* $\delta_{\max}$ *is the maximum extension size,* $|\mathcal{CS}|$ *is the size of the candidate set, and* k *is the pre-defined number of suggestions.*

A. Partial-query Occurrence Localization (POLocate)

The partial-query occurrence localization stage forms the cornerstone of GUIDE's progressive autocompletion by efficiently identifying all occurrences of the current query graph q_0 within the graph database D. This process establishes mappings as anchor points for pattern expansion, directly influencing the quality of suggestions and the system's responsiveness.

GUIDE leverages Algorithm 2 for subgraph isomorphism detection, tailored for interactive autocompletion. The algorithm incorporates optimizations such as early termination

Algorithm 1 The GUIDE Framework

Input: Graph database D, k, user-defined min/max increment sizes $\sigma_{\min}, \sigma_{\max}$
Output: The query result $\mathcal{QS}$

Load default patterns $\mathcal{P}_{\text{default}}$
$i = 0$
$q_i \leftarrow \mathsf{ConstructInitQuery}(\mathcal{P}_{\text{default}})$
while user is not satisfied **do**
  $\mathcal{M}_i \leftarrow \mathsf{POLocate}(q_i, D, \mathcal{M}_{i-1})$
  $\mathcal{QS} \leftarrow \emptyset$
  cur_size $\leftarrow$ sizeof(q_i)
  $\mathcal{CS} \leftarrow \emptyset$
  for each mapping m in $\mathcal{M}_i$
    $\mathcal{CS} \leftarrow \mathsf{CSGenerate}(q_i, m, \mathcal{CS}, \delta_{\max})$
  for each candidate cs in $\mathcal{CS}$
    if $\delta_{\min} \leq |V(cs)| -$ cur_size $\leq \delta_{\max}$
      $\mathcal{QS} \leftarrow \mathsf{QSSelect}(\mathcal{QS}, cs, k)$
  end for
  $\mathsf{DisplayTopK}(\mathcal{QS})$
  $q_{i+1} \leftarrow \mathsf{GetUserChoice}()$
return $\mathcal{QS}$

upon identifying sufficient high-quality mappings and prioritization of frequently accessed database regions. By maintaining state information, it supports incremental matching, enabling efficient updates as users refine q_0. To further enhance performance, each iteration focuses matching efforts on supergraphs $g \in \mathcal{G}$ derived from the previous iteration's results, rather than the entire database D, substantially reducing computational overhead.

For each supergraph $g \in \mathcal{G}$, where $\mathcal{G} = \{g \mid q_0 \subseteq_\lambda g, g \in D\}$, the algorithm generates a mapping $m : V(q_0) \to V(g)$ satisfying the subgraph isomorphism constraints from Section II. The mapping set is defined as:

Definition 6 (Mapping Set)**.** *Given a query graph q_0 and a graph database D, the mapping set $\mathcal{M}$ is:*

$$\mathcal{M} = \{(g, m) \mid g \in G, m : V(q_0) \to V(g), q_0 \subseteq_m g\} \quad (1)$$

where $q_0 \subseteq_m g$ indicates that q_0 is subgraph isomorphic to g under the injective mapping m.

Algorithm 2 Partial-query Occurrence Localization (POLocate)

Input: Query graph q_i, database D, last mapping set $\mathcal{M}_{i-1}$
Output: Mapping set $\mathcal{M}_i$

$\mathcal{M}_i \leftarrow \emptyset$
if $M_{i-1} = \emptyset$
  $M_i \leftarrow \mathsf{GetMapping}(V(q_0) \to D))$
else
  $M_i \leftarrow \mathsf{GetMapping}(V(q_0) \to M_{i-1}))$
return $\mathcal{M}_i$

B. Candidate Suggestions Generation (CSGenerate)

Following the identification of mappings in $\mathcal{M}$, GUIDE generates candidate query suggestions through systematic pattern expansion. The CSGenerate algorithm produces a comprehensive set of subgraphs cs where $q_0 \subseteq_\lambda cs$ and $cs \subseteq g$ for each $(g, m) \in \mathcal{M}$. CSGenerate extends the classic gSpan algorithm with query-specific optimizations that enable efficient exploration of topologically diverse patterns. Unlike gSpan's frequency-driven mining approach, CSGenerate performs exhaustive structure-aware enumeration to capture all valid extensions. The algorithm employs four complementary extension operations:

(i) *Backward extension*: Connects the rightmost vertex to an existing vertex in the subgraph, enhancing connectivity without expanding the vertex set. This facilitates the formation of cycles and denser structures.
(ii) *RmExt extension*: Adds a new vertex linked to the rightmost vertex, promoting outward growth along the current path and expanding the subgraph systematically.
(iii) *General extension*: Attaches a new vertex to any existing vertex (not just the rightmost one), relaxing the rightmost-path constraint to enable richer and more flexible topological exploration.
(iv) *Final cycle extension*: Re-examines all non-adjacent vertex pairs after expansion attempts, adding any valid closing edges from the database that weren't available earlier. This comprehensive final check ensures no potential cycle formation opportunities are missed before termination.

Figure 3 illustrates how CSGenerate generates diverse extensions from a simple triangular query pattern. The algorithm produces various topological variations including chain extensions, branching structures, and cyclic closures, demonstrating its ability to explore different structural directions that users might pursue in their query formulation process.

Example 3. *Consider a query graph q_0 with vertices v_1 through v_6 arranged along a rightmost path $v_1 \to v_2 \to \cdots \to v_6$, where v_6 serves as the rightmost vertex according to the DFS traversal order. To illustrate how CSGenerate generates diverse pattern suggestions from this base structure, we examine four expansion strategies. First, through backward extension g_1, the algorithm explores potential cycles by attempting to connect the rightmost vertex v_6 back to existing vertices in the pattern, such as forming edge v_6-v_1. Second, the rightmost extension strategy g_2 systematically grows the pattern by attaching new vertices like v_7 exclusively to v_6 via edges such as v_6-v_7, maintaining the rightmost growth principle. Third, general extension g_3 relaxes this constraint and enables connections from any vertex along the rightmost path (v_1 through v_5) to external vertices, exemplified by edges like v_3-v_7. Finally, after any structural modification, the final cycle detection phase g_4 performs a comprehensive scan to identify all possible closing edges between non-adjacent vertices, ensuring that cyclic opportunities such as v_7-v_1 are captured even when they emerge after the initial expansion. This multi-faceted approach ensures that GUIDE explores*

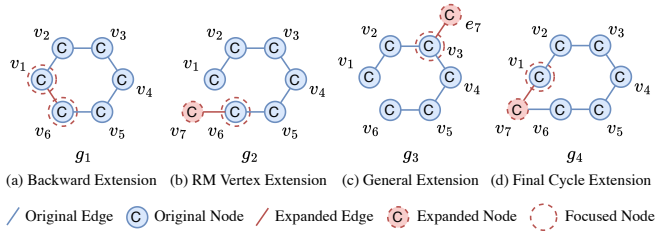


Fig. 3: Extension operations in CSGenerate.

the full spectrum of topological variations while maintaining computational efficiency.

A key innovation in CSGenerate is its non-recursive, breadth-first task queue architecture, which contrasts with gSpan's recursive depth-first approach. Each task encodes a pattern's DFS code along with its projection state, enabling efficient parallel processing and reducing memory overhead via explicit stack management. This design is particularly well-suited for interactive applications where response time is critical.

To ensure suggestion quality, CSGenerate validates each extension against the database before inclusion. For every proposed edge or vertex addition, the algorithm verifies its existence in the corresponding data graph g, guaranteeing that all generated patterns represent valid substructures. This validation step is essential for maintaining precision in the suggestion process and preventing the generation of spurious patterns that would mislead users. The algorithm incorporates specialized cyclic structure handling, particularly for domains where cycles are prevalent (e.g., benzene rings in chemistry or cliques in social networks). When a candidate pattern's size approaches the maximum expansion threshold ($cur\_size + \delta_{\max} - 1$), CSGenerate initiates an exhaustive cycle detection phase to identify all potential closing edges. This structural completeness guarantee ensures no important topological motifs are missed due to the rightmost extension paradigm's constraints.

Unlike frequency-centric systems like FastPat [3] that rely on aggressive pruning, CSGenerate employs comprehensive structure-aware enumeration to maximize topological diversity—a critical feature for effective progressive query refinement. The $\delta_{\max}$ parameter directly controls this behavior by defining the maximum allowable node increment during pattern expansion.

C. Query Suggestion Selection (QSSelect)

After generating candidate extensions via CSGenerate, GUIDE evaluates each subgraph cs to assess its suitability as a query suggestion. The system prioritizes patterns that maximize edge coverage in the database D, delivering diverse and impactful suggestions to support effective query formulation.

The top-k suggestions are maintained in a max-heap $\mathcal{QS}$, where each pattern cs is ranked by its edge coverage gain,

Algorithm 3 Candidate Suggestions Generation (CSGenerate)

Input: Query graph q_i, mapping m, candidate set $\mathcal{CS}$, $\delta_{\max}$
Output: Updated $\mathcal{CS}$

$cur\_size \leftarrow |V(q_i)|$, $V_m \leftarrow \{v \mid (v \mapsto u) \in m\}$, $U_m \leftarrow \{u \mid (v \mapsto u) \in m\}$
/ * Backward extension * /
$v_r \leftarrow$ GetRightmostVertex(q_i)
for each $v_j \in V(q_i)$ where (v_r, v_j) non-adjacent
  if edge e exists between $m(v_r)$ and $m(v_j)$ in D
    $\mathcal{CS} \leftarrow \mathcal{CS} \cup \{q_i \cup e\}$
/ * RmExt extension * /
$v_r \leftarrow$ GetRightmostVertex(q_i)
for each edge e from $m(v_r)$ **to new vertex** v_{new}
  if $v_{new} \notin U_m$ **and** $|V(q_i \cup v_{new})| - cur\_size \leq \delta_{\max}$
    $\mathcal{CS} \leftarrow \mathcal{CS} \cup \{q_i \cup v_{new}\}$
/ * General extension * /
for each vertex $v \in V_m$ and incident edge e to $m(v)$ **with target** v_{new}
  if $v_{new} \notin U_m$ **and** $|V(q_i \cup v_{new})| - cur\_size \leq \delta_{\max}$
    $\mathcal{CS} \leftarrow \mathcal{CS} \cup \{q_i \cup v_{new}\}$
/ * Final Cycle extension * /
for each non-adjacent $(v_i, v_j) \in q_i$
  if edge e exists between $m(v_i)$ and $m(v_j)$ in D
    $\mathcal{CS} \leftarrow \mathcal{CS} \cup \{q_i \cup e\}$
return $\mathcal{CS}$

defined as:

$$\text{Gain}(cs, \mathcal{QS}) = |\text{Cov}(cs, D) \setminus \text{Cov}(\mathcal{QS}, D)|, \quad (2)$$

where $\text{Cov}(cs, D)$ denotes the set of edges in D covered by pattern cs, and $\text{Cov}(\mathcal{QS}, D) = \bigcup_{cs' \in \mathcal{QS}} \text{Cov}(cs', D)$ represents the collective edges covered by patterns in the heap. To measure the impact of removing a pattern, the edge coverage loss for a pattern $qs_{\min} \in \mathcal{QS}$ is defined as:

$$\text{Loss}(qs_{\min}, \mathcal{QS}) = |\text{Cov}(qs_{\min}, D) \setminus \text{Cov}(\mathcal{QS} \setminus \{qs_{\min}\}, D)|, \quad (3)$$

capturing the edges uniquely covered by $qs_{\min}$.

For a new pattern cs, the heap update proceeds as follows: if the heap size $|\mathcal{QS}| < k$, cs is inserted with its gain $\text{Gain}(cs, \mathcal{QS})$. If $|\mathcal{QS}| = k$, let $qs_{\min} \in \mathcal{QS}$ be the pattern with the minimum gain $\text{Gain}(qs_{\min}, \mathcal{QS} \setminus \{qs_{\min}\})$. The system computes $\text{Gain}(cs, \mathcal{QS})$ and $\text{Loss}(qs_{\min}, \mathcal{QS})$, replacing $qs_{\min}$ with cs if:

$$\text{Gain}(cs, \mathcal{QS}) > 2 \cdot \text{Loss}(qs_{\min}, \mathcal{QS}). \quad (4)$$

This update, with a time complexity of $O(\log k)$, ensures a diverse set of high-coverage suggestions, aligning with TED's objective of maximizing edge coverage by balancing gain and loss.

Example 4. *Consider a graph database D representing a social network, with nodes as users and edges as friendships. Suppose the current query aims to extend a pattern with a single friendship edge, and CSGenerate generates a candidate subgraph cs that adds a mutual friend edge, forming a*

Algorithm 4 Query Suggestion Selection (QSSelect)

Input: Top-k heap $\mathcal{QS}$, subgraph cs, integer k
Output: Updated Top-k heap $\mathcal{QS}$

```
GainSocre ← Gain(cs, QS)
LossSocre ← Loss(qs_min, QS)
if size(QS) < k
    Insert(cs, LossSocre, QS)
else if ShouldSwap(GainScore, LossScore)
        RemoveMinimum(QS)
        Insert(cs, LossSocre, QS)
Return QS
```

triangle pattern. If Cov(cs, D) covers 60 new friendship edges not in Cov$(\mathcal{QS}, D)$, and the minimum pattern qs_{min} in the heap uniquely covers 25 edges (Loss$(qs_{min}, \mathcal{QS}) = 25$), with $|Cov(\mathcal{QS}, D)| = 250$, $k = 5$, the replacement condition is evaluated as: Gain$(cs, \mathcal{QS}) = 60 > 2 \cdot 25 = 50$. Since $60 > 50$, cs replaces qs_{min}, incorporating the triangle pattern into the suggestions, enhancing query diversity and coverage.

Users may select any of the k suggestions in $\mathcal{QS}$ that aligns with their intent. The chosen suggestion becomes the new query input q_{i+1} for the next iteration. As shown in Figure 2, the GUIDE workflow—encompassing query decomposition, candidate generation, and ranking—is re-executed with q_{i+1} to produce refined suggestions. This iterative process continues until the user constructs a satisfactory graph query structure.

IV. Optimizations

The efficiency of progressive query autocompletion depends critically on the ability to explore large search spaces while maintaining interactive response times. In this section, we present three optimization strategies that enable GUIDE to scale to large graph databases without sacrificing suggestion quality or diversity.

A. Canonical Representation and Pruning

To prevent redundant exploration of isomorphic subgraphs during expansion, `CSGenerate` employs a canonical depth-first search encoding scheme that uniquely identifies each graph topology. This canonicalization is crucial for progressive autocompletion, as different expansion sequences can yield structurally identical patterns that would otherwise be processed multiple times.

Definition 7 (Canonical DFS Code)**.** *The canonical DFS code of a graph g is an ordered sequence of edge tuples:*

$$\text{DFS-Code}(g) = \big\langle (i_0, i_1, \ell(v_{i_0}), \ell(v_{i_1}), \ell(e_{(i_0,i_1)})),\ \ldots \big\rangle$$

where i_k denotes the DFS visit order, and ℓ returns the label of vertices or edges.

The canonical representation ensures that isomorphic graphs produce identical codes regardless of their generation path. `CSGenerate` implements this pruning through a global hash table that stores signatures of previously generated patterns. Each DFS code is hashed into a fixed-length signature by concatenating vertex indices, labels, and edge attributes. When a new subgraph is generated during expansion, its canonical code is computed and checked against the hash table in $O(1)$ time. Patterns with existing signatures are immediately pruned, preventing redundant processing downstream.

B. Edge Coverage-based Search Pruning

Despite the use of canonical representations to eliminate isomorphic duplicates, the search space for mining subgraphs in a graph database remains exponentially large. To address this, GUIDE introduces a branch-and-bound pruning strategy that leverages edge coverage benefits to eliminate expansion paths unlikely to yield high-quality subgraphs for the Top-k suggestion heap $\mathcal{QS}$. This approach focuses on maximizing the coverage function Cov$(\mathcal{QS}, D)$.

Definition 8 (Edge Coverage Pruning)**.** *For a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a current top-k heap of candidate suggestions $\mathcal{QS}$ containing k query suggestions, during the CSGenerate process, for a newly enumerated candidate suggestion $cs \in \mathcal{CS}$, where $\mathcal{CS}$ denotes the set of candidate suggestions mined from supergraphs $\mathcal{G}$ such that $q \subseteq_\lambda cs$ and $cs \subseteq_\lambda \mathcal{G}$ for some $\mathcal{G} \in D$, the edge coverage pruning decision is defined as follows:*

A candidate suggestion cs is pruned if its incremental coverage, defined as $|Cov(cs, D) \setminus Cov(\mathcal{QS}, D)|$, is insufficient to replace the query suggestion in $\mathcal{QS}$ with the smallest contribution to the total edge coverage, as determined by the swapping criteria, where:

- *$Cov(cs, D) = \bigcup_{G_j \in D} Cov(cs, G_j)$, and $Cov(cs, G_j) = \bigcup_{m \in \mathcal{M}} (m(u), m(v))$ is the edge coverage set of cs in graph G_j, with $\mathcal{M}$ being the set of subgraph isomorphic mappings of cs to G_j.*
- *$Cov(\mathcal{QS}, D) = \bigcup_{qs_i \in \mathcal{QS}} \bigcup_{G_j \in D} Cov(qs_i, G_j)$ is the total edge coverage of the current top-k heap $\mathcal{QS}$.*

The edge coverage pruning mechanism relies on key parameters to guide its operation. The term $|\text{Cov}(g, D) \setminus \text{Cov}(\mathcal{QS}, D)|$ represents the number of edges covered by a subgraph g that are not already covered by the current Top-k candidate pattern set $\mathcal{QS}$, quantifying the incremental contribution of g to the total edge coverage. The parameter k denotes the top-k retrieval parameter, specifying the number of patterns to be selected for the final set. The swapping criteria determine whether g should replace an existing pattern in $\mathcal{QS}$ by comparing its incremental coverage to that of the least effective pattern in $\mathcal{QS}$.

Example 5. *Suppose $k = 5$, and during subgraph enumeration, a new subgraph g is evaluated. Its incremental coverage is computed as $|Cov(g, D) \setminus Cov(\mathcal{QS}, D)| = 10$ edges, while the least effective pattern in $\mathcal{QS}$ contributes 15 edges to $Cov(\mathcal{QS}, D)$. Since g's incremental coverage is insufficient to replace the least effective pattern, it is pruned according to the swapping criteria. Thus, Prune(g) = True, and subgraph g is not added to the candidate pattern set $\mathcal{QS}$.*

Theorem 1. *CSGenerate with edge coverage pruning rules has no effect on the quality of final suggestions.*

C. Sampling for Large-scale Databases

To enable efficient processing of large-scale graph databases containing millions of graphs while maintaining suggestion quality, we employ a sampling framework that preserves the statistical properties of the original database. Our sampling methodology consists of four key components:

(1) Feature Vector Representation. Each graph $g \in D$ is transformed into a feature vector that captures its structural characteristics through label frequency distributions. Given a label set $LS = \{\ell_1, \ell_2, \ldots, \ell_{|LS|}\}$, we construct a vector $v_g \in \mathbb{N}^{|LS|}$ where $v_g[i]$ represents the frequency of label ℓ_i in graph G.

(2) Clustering-based Stratification. We apply k-means clustering on the feature vectors to partition the database into groups with similar structural properties. This stratification ensures that graphs with comparable label distributions are grouped together. For example, graphs with vectors $[3, 4, 0, 0, 0, 0]$ and $[4, 3, 0, 0, 0, 0]$ are clustered together as they have minimal Euclidean distance.

(3) Proportional Sampling Strategy. To maintain the database's structural distribution in the sample, we perform proportional sampling from each cluster. For a target sample size $|D'|$ and cluster C_i, we select $|C_i'| = \lfloor |C_i| \times \frac{|D'|}{|D|} \rfloor$ graphs uniformly at random. This ensures that the relative representation of different structural patterns in the sample mirrors that of the original database.

(4) Sample Size Determination. We determine the minimum sample size using statistical bounds to guarantee representation quality. The required sample size is formalized as follows:

Theorem 2. *Let D be a graph database where each graph $G \in D$ has $|E(G)|$ edges with average count $|E_{avg}|$. For any $\epsilon \in (0, 1)$ and $\rho \in (0, 1)$, a uniformly random sample $D' \subseteq D$ satisfies:*

$$\mathbb{P}\left[\sup_{e \in \bigcup E(G)} |\hat{p}_e - p_e| \leq \epsilon\right] \geq 1 - \rho \tag{5}$$

when the sample size meets:

$$|D'| \geq \frac{\ln(2/\rho) + \ln|\mathcal{E}|}{2\epsilon^2 |E_{avg}|} \tag{6}$$

where $\mathcal{E} = \bigcup_{G \in D} E(G)$, p_e is the true edge frequency, and $\hat{p}_e$ its empirical estimate.

This bound ensures that the sampled database D' maintains statistical properties sufficient for accurate pattern discovery.

This sampling framework achieves over 90% pattern coverage while reducing processing time by orders of magnitude, as validated in our experiments on datasets ranging from 10K to 1M graphs. The stratified approach ensures that rare but important structural patterns are preserved in the sample, maintaining the quality of query suggestions while enabling interactive response times.

TABLE II: Characteristics of Dataset

Dataset	$\lvert D\rvert$	**avg**($\lvert V\rvert$)	**avg**($\lvert E\rvert$)	$\lvert l(V)\rvert$	$\lvert l(E)\rvert$
AIDS	10,000	25.42	27.40	51	4
eMolecules	10,000	15.52	15.86	45	1
PubChem23238	23,238	36.10	36.63	100	1
PubChem1000000	1,000,000	42.30	43.78	91	1

V. Experiment

In this section, we present a comprehensive experimental evaluation of GUIDE to demonstrate its effectiveness for progressive graph query autocompletion. Our experiments are designed to answer the following key research questions:

(i) *RQ1 (Effectiveness)*: How effective is GUIDE in reducing user effort during query formulation compared to existing GQAC systems?
(ii) *RQ2 (Efficiency)*: Can GUIDE maintain interactive response times while generating progressive multi-step suggestions?
(iii) *RQ3 (Scalability)*: How does GUIDE perform on large-scale graph databases with millions of graphs?
(iv) *RQ4 (User Experience)*: Do real users find GUIDE's progressive autocompletion helpful?
(v) *RQ5 (Parameter Sensitivity)*: How do system parameters affect GUIDE's suggestion quality?

To systematically address these research questions, we conduct seven experiments. *RQ1* is addressed by Exp 1, Exp 2, and Exp 6. *RQ2* is evaluated through Exp 3 and Exp 5. *RQ3* is investigated in Exp 4 and Exp 5. *RQ4* is addressed by Exp 3. *RQ5* is examined in Exp 7.

A. Experimental Setup

1) Implementation: We implemented GUIDE's prototype in Java, integrating three core components: a modified VF2 algorithm [5] for efficient partial-query occurrence localization, a non-recursive, queue-based implementation of CSGenerate for progressive pattern extension, and the TEDQ scoring mechanism [4] for ranking and suggestions. The source code is publicly available at https://github.com/hahahumble/GUIDE.

2) Datasets: We evaluate GUIDE on three widely-used real-world graph databases from the iGraph benchmark [7]:

- **PubChem** [8]: A chemical compound database containing up to 1 million molecular graphs with complex structural patterns.
- **AIDS** [9]: An antiviral screening database with 10,000 molecular graphs used in drug discovery research.
- **eMolecules**: A curated collection of 10,000 commercially available chemical compounds.

Table II summarizes the key characteristics of these datasets, including graph count $|D|$, average vertices avg($|V|$), average edges avg($|E|$), and label diversity $|l(V)|$, $|l(E)|$.

TABLE III: Evaluation Metrics

Metric	Meaning
Total Steps	Number of steps required to formulate a query.
ANPI	Average number of new nodes added per iteration, counting both accepted suggestions and manual additions.
IncE	Average number of new nodes added per accepted suggestion, measuring exclusively successful autocompletions.
TPM (%)	The percentage of mouse clicks saved compared to manual query construction without autocompletion support.

3) Evaluation Metrics: To assess GUIDE's performance, we employ four metrics as detailed in Table III: Total Steps measures the number of steps required to formulate a query, ANPI quantifies the average number of new nodes added per iteration (including both accepted suggestions and manual additions), IncE evaluates the average number of new nodes added per accepted suggestion (measuring exclusively successful autocompletions), and TPM represents the percentage of mouse clicks saved compared to manual construction.

4) Experimental Environment: All experiments were conducted on a server equipped with an Intel Core i7 processor, 32GB RAM, and Ubuntu 20.04 LTS.

5) Query Sets: Following the iGraph benchmark protocol, we use query sets containing 100 queries per size category, ranging from 4 to 24 edges. These queries represent diverse structural patterns commonly found in real-world applications, ensuring comprehensive evaluation coverage.

6) System Configuration: GUIDE operates with the following settings.

GUIDE Parameters: We configure GUIDE with a maximum increment size of 5 edges per suggestion, balancing diversity and efficiency. The number of suggestions $k = 10$, with TED scoring weights $\lambda_1 = 1.0$ and $\lambda_2 = 0.5$, optimizing the trade-off between edge coverage and structural diversity.

Sampling Configuration: For large-scale databases, we employ proportional sampling with minimum sample size determined by Equation 6. With error tolerance $\epsilon = 0.05$ and confidence level 99% ($\rho = 0.01$), this configuration ensures the sampled subset D' preserves the edge frequency distribution of the original database D, where the required sample size depends on the average edge count $|E_{\text{avg}}|$ of the target database.

B. Experimental Results

Exp 1. Overall Performance Evaluation. We evaluated GUIDE's query construction efficiency across three datasets, configuring the system to expand queries by 3-4 nodes per iteration. As shown in Tables IV, V, and VI, GUIDE demonstrates strong efficiency, requiring only 2-3 steps to construct small queries (8 edges) and scaling to approximately 8-10 steps for complex queries (24 edges). The ANPI varies within our configured range: PubChem achieves the highest expansion for small queries but stabilizes around 2.5 nodes for larger ones, while AIDS and eMolecules maintain more consistent expansion rates throughout. IncE consistently exceeds 3 nodes across all datasets, showing slight improvement with query complexity. Most notably, GUIDE achieves TPM ranging from approximately 50% to over 70%, with PubChem showing the highest savings due to its regular structural patterns. These findings demonstrate that GUIDE facilitates efficient query construction across diverse datasets.

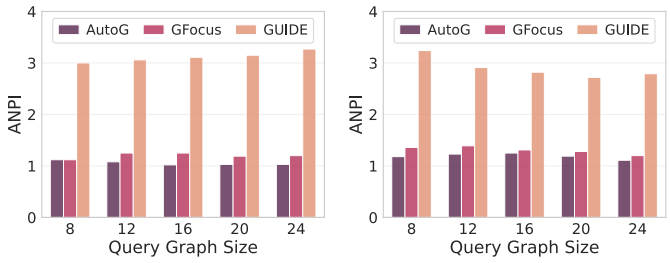
TABLE IV: Overall Performance on AIDS

$\|q\|$	Total Steps	ANPI	IncE	TPM (%)
8	3.39	2.36	3.00	51.38
12	5.09	2.36	3.07	53.42
16	6.18	2.59	3.18	58.25
20	7.37	2.71	3.25	60.65
24	9.12	2.63	3.32	59.92

TABLE V: Overall Performance on PubChem

$\|q\|$	Total Steps	ANPI	IncE	TPM (%)
8	2.21	3.62	3.14	72.37
12	4.18	2.87	3.15	65.16
16	6.06	2.64	3.28	62.12
20	7.99	2.50	3.27	60.05
24	9.63	2.49	3.34	59.87

Exp 2. Experimental Comparison with Existing Methods. We compared GUIDE against AutoG and GFocus across the AIDS and eMolecules datasets, evaluating both ANPI and IncE, as shown in Fig. 4 and Fig. 5. For node expansion, GUIDE consistently outperforms both baselines, achieving approximately 2-3× improvement over AutoG and 2× improvement over GFocus. This superiority is maintained across all query sizes (8-24 edges). In terms of nodes per accepted suggestion, GUIDE demonstrates similarly gains, achieving 1.5-2× higher IncE values compared to both AutoG and GFocus throughout all experiments. The performance gap becomes particularly evident for larger queries. These results demonstrate that GUIDE's strategy enables users to construct complex graph queries with fewer iterations while

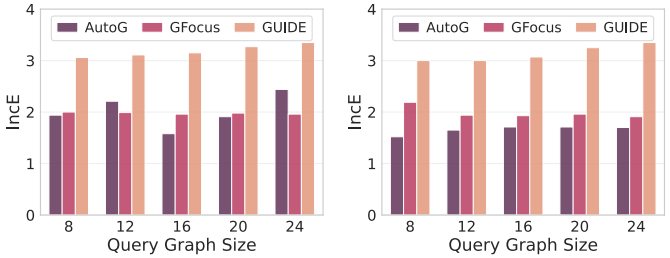


(a) AIDS (b) eMolecules

Fig. 4: ANPI comparison by varying query graph size.

TABLE VI: Overall Performance on eMolecules

$\|q\|$	Total Steps	ANPI	IncE	TPM (%)
8	2.43	3.29	3.11	63.38
12	3.96	3.03	3.01	62.83
16	5.35	2.99	3.08	63.44
20	6.89	2.90	3.30	63.05
24	7.92	3.03	3.42	64.92



(a) AIDS (b) eMolecules

Fig. 5: IncE comparison by varying query graph size.

achieving richer structural development per step compared to existing approaches.

Exp 3. User Study. We recruited 10 undergraduate volunteers from chemistry, biology, and physics programs to evaluate GUIDE's usability in real-world scenarios. Each participant completed four query construction tasks with increasing complexity (8, 12, 16, and 20 edges) across all three datasets. As shown in Fig. 6, we measured four key metrics: mouse clicks, pattern usage, response time, and total completion time.

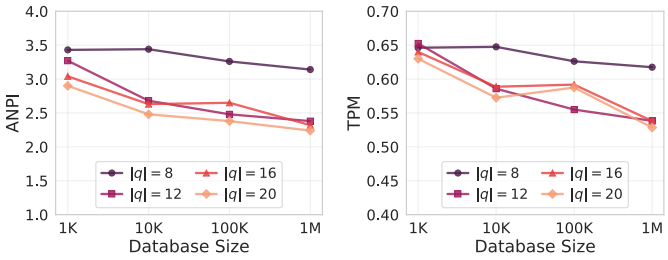
The results reveal clear scaling patterns across all metrics.

(a) Mouse Clicks (b) Selected Suggestions

(c) Response Time (d) Total Time

Fig. 6: User study results by varying query graph size.

Mouse clicks increase linearly with query size, rising from approximately 7 clicks for 8-edge queries to 21-22 clicks for 20-edge queries, demonstrating that user effort scales predictably with query complexity. Selected suggestions follows a similar trend, with users employing 2-3 suggestions for small queries and 5-6 suggestions for complex ones, confirming the value of GUIDE's progressive autocompletion in accelerating query construction. Response times remain under 0.4 seconds for small queries but increase to 0.7-1.1 seconds for 20-edge queries, with PubChem showing the highest latency due to its larger search space. Total completion time ranges from approximately 100 seconds for simple tasks to 350-370 seconds for complex queries, reflecting both the increased interaction steps and cumulative response delays. Despite the scaling challenges, participants successfully completed all tasks with calculated TPM values of 50-65%, validating our simulation results and confirming GUIDE's practical effectiveness for interactive query construction.

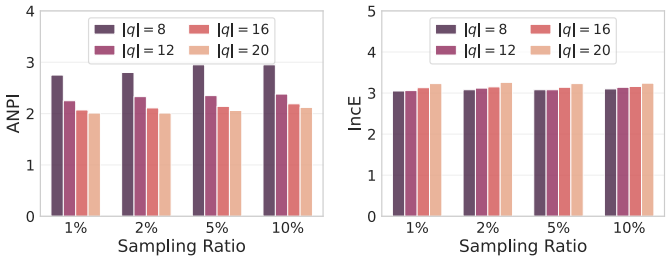


(a) ANPI (b) TPM

Fig. 7: Scalability Test.

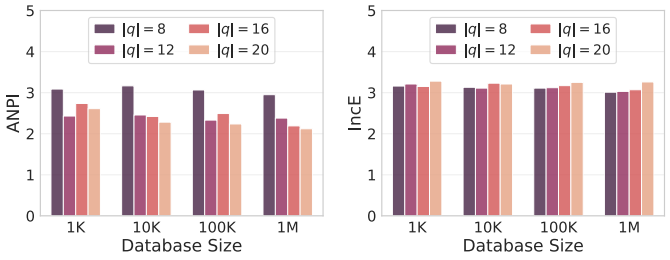
Exp 4. Scalability Test. We evaluated GUIDE's scalability on PubChem datasets ranging from 1K to 1M graphs, measuring both ANPI and TPM across different query sizes (8-20 edges), as shown in Fig. 7. ANPI demonstrates a clear negative correlation with database size across all query complexities. For small queries (8 edges), ANPI maintains relatively high values, declining from 3.45 on 1K graphs to 3.15 on 1M graphs. However, larger queries exhibit more pronounced degradation: 20-edge queries drop from 2.9 to 2.25, indicating that complex queries are more sensitive to database scale. TPM shows similar scaling characteristics, with small queries maintaining better efficiency at scale. For 8-edge queries, TPM decreases from 65% on 1K graphs to 62% on 1M graphs, demonstrating robust performance retention. Larger queries experience more substantial impact, with 20-edge queries dropping from 63% to 53%. Despite the gradual decline, GUIDE maintains over 50% interaction savings even on million-graph databases without any preprocessing or sampling, confirming its practical applicability for large-scale graph query systems.

Exp 5. Sampling Strategy Evaluation. We further evaluated our proportional sampling strategy on PubChem datasets with 1M graphs to assess its effectiveness in maintaining performance while reducing computational overhead. The evaluation consisted of two experiments: varying sampling ratios on



(a) ANPI (b) IncE

Fig. 8: Effect of sampling ratio on performance for different query graph sizes during sampling.



(a) ANPI (b) IncE

Fig. 9: Effect of database size on performance for different query graph sizes during sampling.

a fixed database size and scaling database sizes with fixed sampling ratio.

Effect of Sampling Ratio (Fig. 8): We tested sampling ratios from 1% to 10% on PubChem while measuring ANPI, IncE and response time across different query sizes (8-20 edges). As expected, larger query graphs consistently yield lower ANPI values across all sampling ratios, with 8-edge queries achieving around 2.7-2.9 while 20-edge queries drop to approximately 2.0-2.2, reflecting the increased complexity of finding suitable expansions for larger structures. Interestingly, ANPI shows a weak positive correlation with sampling ratio, improving marginally as sampling increases from 1% to 10%—demonstrating that even small samples effectively capture the database's structural patterns. In contrast to ANPI's sensitivity to query size, IncE exhibits strong stability across all sampling ratios, consistently maintaining values around 3.0 or higher, indicating that nodes added per accepted suggestion is largely independent of sampling ratio. As shown in Table VII, response time increases with sampling ratio, rising from 5.89 seconds at 1% to 81.17 seconds at 10%. This trade-off between response time and suggestion quality allows practitioners to choose appropriate sampling ratios based on their latency requirements.

Scalability with Fixed Sampling Ratio (Fig. 9): Using a 10% sampling ratio, we evaluated GUIDE on PubChem subsets ranging from 1K to 1M graphs. ANPI exhibits a weak negative correlation with database size, showing only modest decline from approximately 2.5-3.0 on 1K graphs to around 2.1-2.9 on 1M graphs. This gentle degradation indicates that

TABLE VII: Average Response Time with Sampling

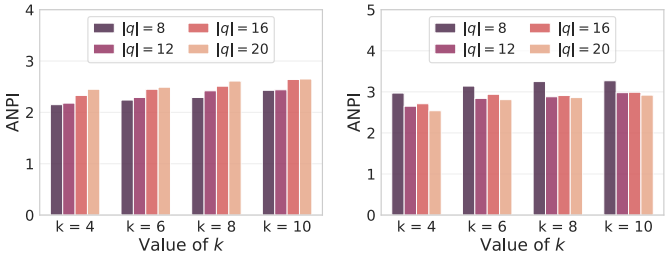
Sampling Ratio	**1%**	**2%**	**5%**	**10%**
Average Response Time	5.89	10.81	37.23	81.17

TABLE VIII: Default Pattern Evaluation

Dataset	**AP (%)**	**AvgNodes**	**EffNodes**
AIDS	94.68	5.67	5.37
eMolecules	80.62	6.43	5.18
PubChem23238	90.42	6.69	6.03

our sampling strategy effectively maintains node expansion capabilities even as the database scales. In contrast, IncE shows virtually no correlation with database size, consistently maintaining values above 3.0 across all scales from 1K to 1M graphs. This stability demonstrates that nodes added per accepted suggestion is largely unaffected by database size, as our suggestion selection mechanism ensures rich structural suggestions regardless of the underlying database scale.

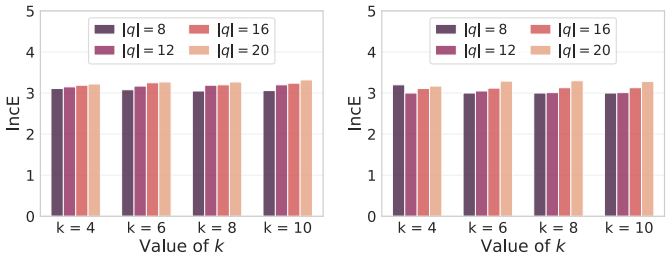
Exp 6. Default Pattern Effectiveness. To validate the utility of pre-computed default patterns in accelerating query construction, we evaluated their effectiveness across three datasets, as shown in Table VIII. The results demonstrate high adoption rates (AP) exceeding 80% across all datasets and reaching 94.68% for AIDS, confirming that these pre-computed structures serve as highly relevant starting points for real-world queries. Each pattern contributes 5.67-6.69 nodes (AvgNodes) to the initial query, effectively eliminating multiple manual drawing steps. The EffNodes metric (EffNodes $=$ AP $\cdot$ AvgNodes), combining adoption rate with structural contribution, yields values of 5.18-6.03, confirming that default patterns substantially accelerate query construction by providing 5-6 nodes that users would otherwise draw manually.



(a) AIDS (b) eMolecules

Fig. 10: Effect of parameter k on ANPI.

Exp 7. Impact of Parameter k on Suggestion Quality. We investigated how the number of suggestions k affects GUIDE's performance by varying k from 4 to 10 on AIDS and eMolecules datasets, measuring both ANPI and IncE across different query sizes. As shown in Fig. 10, ANPI exhibits a gradual upward trend as k increases from 4 to 10 across both datasets. For AIDS, ANPI improves from 2.1-2.4 at $k = 4$ to 2.4-2.7 at $k = 10$, representing a 10-15% im-



(a) AIDS (b) eMolecules

Fig. 11: Effect of parameter k on IncE.

provement. eMolecules exhibits a similar pattern, with ANPI values increasing from 2.5-2.9 at $k = 4$ to 2.9-3.2 at $k = 10$. This pattern suggests that additional suggestions provide more options for node expansion. In contrast, IncE (Fig. 11) remains quite stable across all k values, maintaining around 3.0-3.3 for both datasets, demonstrating that nodes added per accepted suggestion depends primarily on the quality of top-ranked suggestions rather than quantity.

VI. Related Work

This section reviews prior work on graph database query autocompletion and related techniques.

A. Graph Query Autocompletion Systems

As detailed in Section II, AutoG [1] and GFocus [2] represent the current state-of-the-art in graph query autocompletion. While these systems pioneered the field, they suffer from fundamental limitations including extensive preprocessing requirements and minimal incremental suggestions. GUIDE addresses these limitations through an online, preprocessing-free approach that generates arbitrarily-sized suggestions.

B. Visual Query Interfaces

Visual query interfaces have gained significant attention for making graph querying more accessible. Systems like QUBLE [25] enables visual subgraph query formulation integrated with query processing on large networks. VIIQ [26] provides an auto-suggestion enabled visual interface that suggests individual nodes and edges during query construction. AURORA [24] presents a data-driven approach for automatically constructing visual graph query interfaces based on query logs. While these systems improve query accessibility through visual metaphors, they lack sophisticated autocompletion capabilities that can suggest complete structural patterns. GUIDE builds upon these visual interfaces by integrating progressive autocompletion that suggests meaningful subgraph structures rather than individual elements.

C. Graph Indexing and Pattern Mining

Traditional graph indexing techniques focus on accelerating query execution rather than query formulation. Systems like gIndex [13] and FG-index [10] index frequent substructures to filter non-answers efficiently during query processing. Similarly, graph mining algorithms such as gSpan [6] extract frequent patterns from graph databases but are designed for offline pattern discovery rather than interactive query suggestion. These approaches require extensive preprocessing and cannot adapt to database changes without complete reindexing. GUIDE adapts pattern mining concepts through its CSGenerate algorithm but performs online pattern extraction focused on query-relevant structures, eliminating preprocessing overhead.

D. Query Reformulation and Exploration

Query reformulation approaches help users explore graph data by suggesting alternative queries. Mottin et al. [14] proposed a graph query reformulation framework that generates queries maximizing result coverage and diversity. However, these methods assume users have already formulated complete queries and seek alternatives, rather than assisting during initial query construction. Exploratory search paradigms [15] advocate for iterative refinement in information seeking but have not been effectively adapted for graph query construction. GUIDE integrates exploratory principles by enabling users to progressively build queries through meaningful structural additions, supporting both query construction and exploration simultaneously.

E. XML Query Autocompletion

The XML community has extensively studied query autocompletion, with systems providing fuzzy type-ahead search [16] and position-aware suggestions [17]. However, XML's hierarchical tree structure differs fundamentally from arbitrary graph topologies. While XML autocompletion suggests elements along tree paths, graph queries must handle complex topological relationships including cycles, multiple paths between nodes, and diverse structural patterns. The techniques developed for XML are thus not directly applicable to general graph databases.

VII. Conclusion

In this paper, we presented GUIDE, a novel framework for progressive visual query autocompletion in graph databases that eliminates the limitations of existing index-heavy approaches. Our key innovations include: (1) an online mechanism that operates directly on databases without preprocessing overhead, enabling instant adaptation to data changes; (2) progressive multi-step suggestions of arbitrary size that drastically reduce user interactions compared to existing systems' minimal edge increments; and (3) guaranteed validity of all suggestions as actual database substructures. Extensive experiments demonstrate GUIDE's superiority, doubling structural progress per iteration compared to existing methods while maintaining interactive response times on databases with up to one million graphs. By making graph query formulation more efficient and intuitive, GUIDE represents a significant step toward democratizing access to graph databases for domain experts without extensive technical training.

REFERENCES

[1] P. Yi, Choi, B., Bhowmick, S.S. et al. "AutoG: a visual query autocompletion framework for graph databases." The VLDB Journal 26, 347–372 (2017). https://doi.org/10.1007/s00778-017-0454-9

[2] P. Yi, B. Choi, Z. Zhang, S. S. Bhowmick and J. Xu, "GFocus: User Focus-Based Graph Query Autocompletion," in IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 4, pp. 1788-1802, 1 April 2022, doi: 10.1109/TKDE.2020.3002934.

[3] J. Zeng, L. H. U, X. Yan, M. Han and B. Tang, "Fast Core-based Top-k Frequent Pattern Discovery in Knowledge Graphs," 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 2021, pp. 936-947.

[4] Huang K, Hu H, Ye Q, Tian K, Zheng B, Zhou X. "TED: Towards Discovering Top-k Edge-Diversified Patterns in a Graph Database" in *Proceedings of SIGMOD*, 2023.

[5] L. P. Cordella, P. Foggia, C. Sansone and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct. 2004, doi: 10.1109/TPAMI.2004.75.

[6] Xifeng Yan and Jiawei Han, "gSpan: graph-based substructure pattern mining," 2002 IEEE International Conference on Data Mining, 2002. Proceedings., Maebashi City, Japan, 2002, pp. 721-724, doi: 10.1109/ICDM.2002.1184038.

[7] 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)

[8] S. Kim *et al.*, "PubChem 2023 update," *Nucleic Acids Res.*, vol. 51, no. D1, pp. D1373–D1380, Jan. 2023.

[9] National Cancer Institute, "AIDS antiviral screen dataset," DTP Developmental Therapeutics Program, 1999. [Online]. Available: https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data

[10] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases", in Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Beijing, China, 2007, pp. 857–872.

[11] S. Zhang, S. Li, and J. Yang, "GADDI: distance index based subgraph matching in biological networks", in Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, Saint Petersburg, Russia, 2009, pp. 192–203.

[12] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism", Proc. VLDB Endow., vol. 1, no. 1, pp. 364–375, Aug. 2008.

[13] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach", in Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, 2004, pp. 335–346.

[14] D. Mottin, F. Bonchi, and F. Gullo, "Graph Query Reformulation with Diversity", in Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, 2015, pp. 825–834.

[15] G. Marchionini, "Exploratory search: from finding to understanding", Commun. ACM, vol. 49, no. 4, pp. 41–46, Apr. 2006.

[16] J. Feng and G. Li, "Efficient Fuzzy Type-Ahead Search in XML Data," in IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 5, pp. 882-895, May 2012, doi: 10.1109/TKDE.2010.264.

[17] C. Lin, J. Lu, T. W. Ling and B. Cautis, "LotusX: A Position-Aware XML Graphical Search System with Auto-Completion," 2012 IEEE 28th International Conference on Data Engineering, Arlington, VA, USA, 2012, pp. 1265-1268, doi: 10.1109/ICDE.2012.123.

[18] S. Comai, E. Damiani, and P. Fraternali, "Computing graphical queries over XML data", ACM Trans. Inf. Syst., vol. 19, no. 4, pp. 371–430, Oct. 2001.

[19] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos, "QURSED: querying and reporting semistructured data", in Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, 2002, pp. 192–203.

[20] D. Braga, A. Campi, and S. Ceri, "XQBE (XQuery By Example): A visual interface to the standard XML query language", ACM Trans. Database Syst., vol. 30, no. 2, pp. 398–443, Jun. 2005.

[21] S. S. Bhowmick, H. E. Chua, B. Thian and B. Choi, "ViSual: An HCI-inspired simulator for blending visual subgraph query construction and processing," 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea (South), 2015, pp. 1480-1483, doi: 10.1109/ICDE.2015.7113406.

[22] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan and R. Elmasri, "GQBE: Querying knowledge graphs by example entity tuples," 2014 IEEE 30th International Conference on Data Engineering, Chicago, IL, USA, 2014, pp. 1250-1253, doi: 10.1109/ICDE.2014.6816753.

[23] G. Nemhauser, L. Wolsey, and M. Fisher, "An Analysis of Approximations for Maximizing Submodular Set Functions—I", Mathematical Programming, vol. 14, pp. 265–294, 12 1978.

[24] S. S. Bhowmick, K. Huang, H. E. Chua, Z. Yuan, B. Choi, and S. Zhou, "AURORA: Data-driven Construction of Visual Graph Query Interfaces for Graph Databases", in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 2020, pp. 2689–2692

[25] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou, "QUBLE: blending visual subgraph query formulation with query processing on large networks," in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 2013, pp. 1097–1100.

[26] N. Jayaram, S. Goyal, and C. Li, "VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation," Proceedings of the VLDB Endowment, vol. 8, no. 12, pp. 1940–1951, 2015.

VIII. APPENDIX

A. Proof of Lemma 1.

Proof. The GUIDE algorithm's complexity arises from three components. POLocate performs subgraph isomorphism for each graph in D, taking $2^{\mathcal{O}(|V(q_i)|)}$ per graph, yielding $\mathcal{O}(|D| \cdot 2^{\mathcal{O}(|V(q_i)|)})$. CSGenerate extends each of $|\mathcal{M}|$ matches up to $\delta_{\max}$ nodes, contributing $\mathcal{O}(|\mathcal{M}| \cdot \delta_{\max})$. QSSelect maintains a top-k heap over $|\mathcal{CS}|$ candidates, requiring $\mathcal{O}(|\mathcal{CS}| \cdot \log k)$. Summing these terms gives the total complexity. □

B. Proof of Theorem 1.

Proof. Let the current candidate suggestion be cs and final suggestions QS. Per QSSelect (Eq. (4), Section III-C), only promising candidates cs are retained in QS. Given a cs, edge coverage pruning does not filter out promising candidates, ensuring no impact on suggestion quality. □

C. Proof of Theorem 2.

Proof. We extend the VC theory to graph data, aiming to bound the deviation of empirical edge probabilities from their true values with precision.

(i) *Concentration*: For a fixed edge e, let p_e be the true occurrence probability and $\hat{p}_e$ the empirical estimate over $|D'|$ independent samples from $D' \subseteq D$. Each sample is a Bernoulli trial with p_e, confined to $[0, 1]$. Hoeffding's inequality for independent bounded variables states:

$$\mathbb{P}\left(|\hat{p}_e - \mathbb{E}[\hat{p}_e]| > \epsilon\right) \leq 2\exp\left(-2\epsilon^2|D'|/(b-a)^2\right),$$

where $[a, b] = [0, 1]$ and $\mathbb{E}[\hat{p}_e] = p_e$. This simplifies to:

$$\mathbb{P}\left(|\hat{p}_e - p_e| > \epsilon\right) \leq 2\exp(-2\epsilon^2|D'|).$$

Incorporating $|E_{\text{avg}}|$, the effective edge weight, adjusts the bound to:

$$\mathbb{P}\left(|\hat{p}_e - p_e| > \epsilon\right) \leq 2\exp(-2\epsilon^2|D'||E_{\text{avg}}|).$$

(ii) *Union Bound*: For all $|\mathcal{E}|$ edges, the probability that any edge deviates by more than ϵ is:

$$\mathbb{P}\left(\exists e : |\hat{p}_e - p_e| > \epsilon\right) \leq \sum_{e \in \mathcal{E}} \mathbb{P}\left(|\hat{p}_e - p_e| > \epsilon\right) \tag{7}$$

$$\leq 2|\mathcal{E}|\exp(-2\epsilon^2|D'||E_{\text{avg}}|). \tag{8}$$

(iii) *Sample Size*: To ensure the deviation probability is below confidence level ρ:

$$2|\mathcal{E}| \exp(-2\epsilon^2 |D'||E_{\text{avg}}|) \leq \rho.$$

Taking logarithms and rearranging:

$$\ln(2|\mathcal{E}|) - 2\epsilon^2 |D'||E_{\text{avg}}| \leq \ln(\rho).$$

Solving for $|D'|$:

$$|D'| \geq \frac{\ln(2|\mathcal{E}|/\rho)}{2\epsilon^2 |E_{\text{avg}}|} = \frac{\ln(2/\rho) + \ln|\mathcal{E}|}{2\epsilon^2 |E_{\text{avg}}|}.$$

□

D. Corollary 1 (Practical Sampling Bound)

This corollary offers a practical simplification of the sample size bound when the number of edges $|\mathcal{E}|$ is significantly smaller than the exponential term $\exp(\epsilon^2 |E_{\text{avg}}||D'|)$. This condition enables a concise expression for $|D'|$, the dataset size, essential for applications with limited edges, facilitating reliable sample size determination.

The lower sample size limit of the original theorem $|D'| \geq \frac{\ln(2/\rho)+\ln|\mathcal{E}|}{2\epsilon^2 |E_{\text{avg}}|}$ is based on the logarithmic term $\ln|\mathcal{E}|$. When the edge scale satisfies:

$$|\mathcal{E}| \leq \exp\left(\epsilon^2 |E_{\text{avg}}| \cdot |D'|\right),$$

the $\ln|\mathcal{E}|$ term is overshadowed by the exponential, simplifying the bound to depend solely on the confidence level ρ:

$$|D'| \geq \frac{\ln(2/\rho)}{2\epsilon^2 |E_{\text{avg}}|}.$$

This provides a streamlined tool for analyzing **large-scale sparse graphs** where $\ln|\mathcal{E}| \ll \epsilon^2 |E_{\text{avg}}| \cdot |D'|$.