Envisage: Towards Expressive Visual Graph Querying

Xiaolin Wen, Qishuang Fu, Shuangyue Han, Yichen Guo, Joseph K. Liu, and Yong Wang

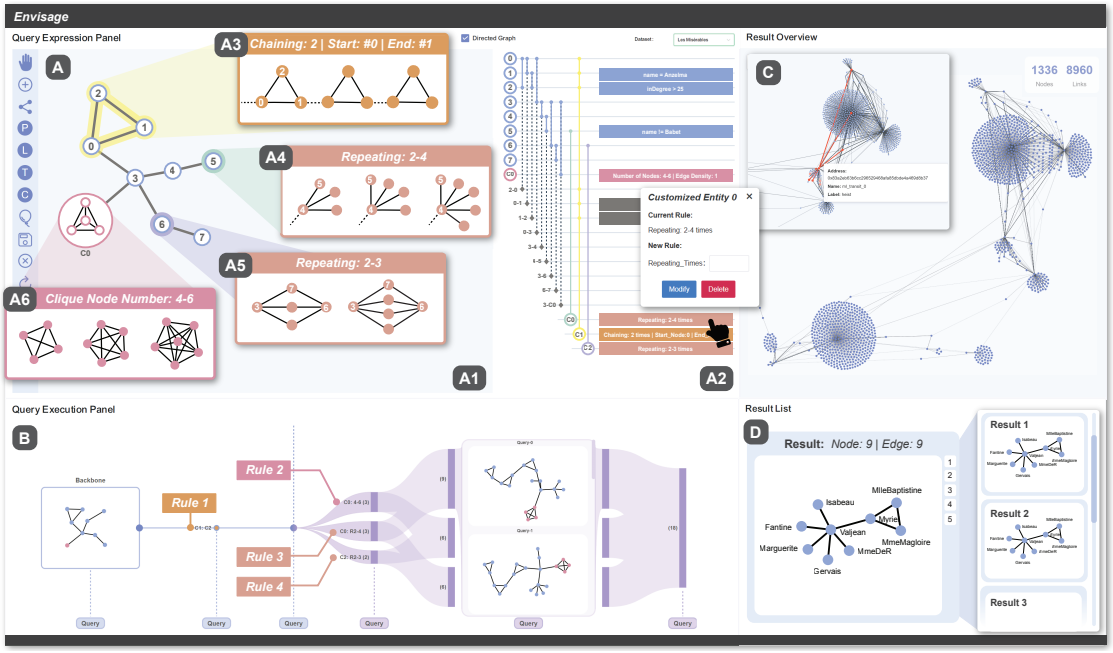

Fig. 1: The Interface of *Envisage* includes four components: *Query Expression Panel* (A), *Query Execution Panel* (B), *Result Overview* (C), and *Result List* (D). Users can flexibly perform visual graph querying by quickly constructing a graph structure (A1), applying various rules (A2), verifying and executing the generated query instances (B), and analyzing the results (C and D).

Abstract—Graph querying is the process of retrieving information from graph data using specialized languages (e.g., Cypher), often requiring programming expertise. Visual Graph Querying (VGQ) streamlines this process by enabling users to construct and execute queries via an interactive interface without resorting to complex coding. However, current VGQ tools only allow users to construct simple and specific query graphs, limiting users' ability to interactively express their query intent, especially for underspecified query intent. To address these limitations, we propose *Envisage*, an interactive visual graph querying system to enhance the expressiveness of VGQ in complex query scenarios by supporting intuitive graph structure construction and flexible parameterized rule specification. Specifically, *Envisage* comprises four stages: *Query Expression* allows users to interactively construct graph queries through intuitive operations; *Query Verification* enables the validation of constructed queries via rule verification and query instantiation; *Progressive Query Execution* can progressively execute queries to ensure meaningful querying results; and *Result Analysis* facilitates result exploration and interpretation. To evaluate *Envisage*, we conducted two case studies and in-depth user interviews with 14 graph analysts. The results demonstrate its effectiveness and usability in constructing, verifying, and executing complex graph queries.

Index Terms—Visual graph querying, interactive query construction, graph data

1 Introduction

Graph data represents the relationships among entities and has become widely used in various application domains, including social network analysis [18], bioinformatics [55], and financial fraud detection [59]. Graph querying refers to the process of retrieving relevant information (e.g., nodes, edges, and subgraphs) from graph data according to user-defined rules [62]. It is a fundamental operation for many graph exploration tasks, such as node labeling [34], anomaly detection [36], and pattern analysis [53]. Traditional graph querying methods typically require users to construct text-based queries formulated in specialized graph query languages such as Cypher [17] and GraphQL [23], which are subsequently executed on graph databases like Neo4j [39]. However, composing such queries using graph query languages typically requires strong programming skills [6]. To address this problem, several studies have proposed leveraging visualization and human-computer interaction (HCI) techniques to simplify the query construction process [7, 11, 13, 43, 55], which is referred to as ***Visual Graph Querying (VGQ)***. Specifically, VGQ enables users to interactively construct graph queries by dragging and linking nodes [7, 11, 44], selecting predefined subgraph examples [51], and imposing specific constraints on graph nodes and edges [55]. Nevertheless, existing VGQ approaches primarily support simple and fully-specified graph queries, such as small, fixed subgraphs with attribute constraints. This **limited expressiveness** restricts users' ability to construct and refine queries flexibly using such VGQ systems, particularly when addressing underspecified query

- *Xiaolin Wen, Shuangyue Han, Yichen Guo, and Yong Wang are with Nanyang Technological University. E-mail: {xiaolin004, shuangyu001, yguo017}@e.ntu.edu.sg, yong-wang@ntu.edu.sg.*
- *Qishuang Fu and Joseph K. Liu are with Monash University. E-mail: {qishuang.fu, joseph.liu}@monash.edu.*
- *Yong Wang is the co-corresponding author.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

intent, which is essential in many real-world scenarios [6]. In this paper, underspecified query intent refers to a loosely defined graph pattern that can match multiple possible queries, similar to how a regular expression can match a variety of string patterns [54]. For example, a user may want to query a "connector" subgraph (i.e., two nodes connected to a common set of intermediary nodes [15], as shown in Fig. 1A5), without specifying the exact number of intermediary nodes.

We surveyed existing research on VGQ [5, 6, 37, 44, 51] and conducted a preliminary study (Sec. 4.1) with four graph analysis experts to identify the challenges in expressing graph query intent via existing VGQ systems. Specifically, there are four major challenges (**C1-C4**) in achieving expressive visual graph querying. **C1. Full Expression of Underspecified Graph Query Intent.** Most existing VGQ tools require users to define a concrete query graph structure. However, user intent is often underspecified and cannot be fully captured by a fixed graph structure. For example, a user may want to query a subgraph with two loops connected by a path, where the number of nodes in each component can be different. Such query intentions are common and can be expressed using operators like the *Kleene star* in some graph query languages [16], but are difficult to express using existing VGQ tools. **C2. Fast Specification of Repetitive Graph Structure.** Users' query intent may involve repetitive substructures, but existing VGQ tools require users to manually construct each substructure, which is inefficient and tedious. For the aforementioned "connector" pattern, it would be highly time-consuming to manually create a large number of intermediary nodes and connect them accordingly. This becomes even more challenging when the repeated substructures are complex. **C3. Flexible Configuration of Node/Edge Attribute Constraints in Graph Queries.** Specifying attribute constraints (e.g., node labels or edge attribute values) is essential for constructing graph queries [44]. However, existing tools like SIERRA [37] and Visage [44] typically require users to edit these constraints for each node or edge individually, which is laborious and time-consuming. As the number of necessary constraints increases, it becomes increasingly difficult to review, verify, and modify graph queries. **C4. Effective Verification and Execution of Query Instances.** Even when users can express the aforementioned underspecified query intent, the possible number of concrete query instances reflecting their intended patterns can be enormous, which demands an effective way to verify, select, and execute appropriate graph query instances aligned with their query intent.

To address these challenges, we propose a novel expressive visual graph querying framework and further develop ***Envisage***, an interactive visual analytics system to **en**hance the expressiveness of **visu**a**l** **g**raph qu**e**rying. *Envisage* enables users to conduct visual graph querying through four stages: *Query Expression*, *Query Verification*, *Progressive Query Execution*, and *Result Analysis*. Users begin by expressing their query intent, including underspecified intent (**C1**), through intuitive operations such as customized motif configuration and parameterized rule specification, where two types of rules (i.e., repeating and chaining) help users efficiently define repetitive graph patterns (**C2**). *Envisage* further enables quick configurations of attribute constraints for different entities by specifying corresponding attribute rules (**C3**). Then, users can check and confirm that their constructed graph query matches their intended query by verifying their defined rules and the generated query instances (**C4**). Users can also progressively execute query instances to identify and rectify issues in the current graph query (**C4**). The query results are displayed as a list of subgraphs (Fig. 1D), with their distribution revealed in the input graph (Fig. 1C). To evaluate *Envisage*, we conducted two case studies and in-depth user interviews with 14 graph analysts. The results demonstrate that *Envisage* enables expressive visual graph querying, allowing users to effectively explore graph data. Our main contributions are as follows:

- We formulate design requirements for flexibly expressing query intent through visual graph querying, based on a preliminary study with four graph analysis experts, and propose an interactive visual graph querying framework to meet the requirements.
- We introduce *Envisage*, an interactive visual graph querying system based on our proposed framework to help users express, verify, and execute graph queries in a flexible and expressive way.
- We present two case studies and conduct in-depth user interviews with 14 graph analysts to demonstrate the effectiveness and usefulness of *Envisage*.

2 Related Work

This work is related to prior research on *visual graph querying* and *subgraph matching*.

2.1 Visual Graph Querying

Given our focus on effectively expressing query intent, we comprehensively review the query construction processes of existing VGQ methods. First, most VGQ systems, such as Graphite [11], Prague [28], VOGUE [7], ViSual [8], and VIMO [55], require users to manually build query graphs by interactively specifying each node and edge [6]. One exception is VIGOR [43], which accepts text-based queries and focuses on visualizing both queries and results. Second, some systems adopt a "*query-by-template*" approach [44], enabling users to construct queries from predefined graph templates quickly. For instance, VisualNeo [25] enables users to construct a subgraph query with multiple nodes and edges by performing a single click-and-drag action. Likewise, Song et al. [51] enable users to conduct example-based graph querying by allowing them to select a graph instance to construct the query graph. Third, specifying constraints on attributes of nodes and edges is essential for meaningful graph querying [11, 44, 51]. SIERRA [37] introduces visual abstraction, called labeled composite graph, to represent attribute constraints in a visual graph query. Also, some tools support interactions tailored to specific domain tasks and graph types, such as knowledge graphs [57], multilayer networks [13], bipartite networks [45], and hierarchical graphs [35].

Existing methods require users to construct concrete and fully-specified query graphs, which is time-consuming and has no support for underspecified query intent that are common in real-world applications. *Envisage* allows users to configure customized motifs and specify parameterized rules to flexibly express their queries. Users can quickly construct repetitive graph patterns and specify attribute constraints by combining multiple rules. Furthermore, *Envisage* enables users to verify query instances derived from underspecified query intent, ensuring that query execution aligns with their expectations.

2.2 Subgraph Matching

Subgraph matching aims to find all subgraphs g in a data graph G that match a given query graph pattern q. Existing subgraph matching algorithms can be categorized into three groups: backtracking-based, join-based, and neural network-based approaches. **Backtracking-based approaches** optimize depth-first search through filtering and pruning techniques. Notable methods include VF2 [12], VF2++ [31], and DP-iso [21], which reduces the search space via preprocessing. Many graph databases integrate these techniques [22, 23, 46, 47, 50, 65], with GraphQL [23] improving efficiency through local pruning and global refinement. **Join-based approaches** decompose queries into smaller sub-patterns and then combine intermediate results through join operations, such as binary joins or generalized multi-way joins [1,3,33,38,52]. Cypher [23], Neo4j's core query language, applies this approach to property graphs. **Neural network-based Approaches** utilize graph neural networks (GNNs) for approximate matching. GraphSAGE [20] improves scalability, while Neuralign [51] builds upon it and enhances dynamic graph alignment. GNN-PE [63] introduces exact matching with path-dominance embeddings but is computationally expensive.

User-defined queries in *Envisage* are ultimately represented as individual subgraphs with attribute constraints, which can be readily translated into query languages that support fundamental features like subgraph matching, attribute filtering, and variable-length path matching. *Envisage* adopts Cypher via Neo4j [40] for its user-friendly syntax and robust support for property graph querying.

3 Background

This section introduces background information, including *graph querying*, *graph definitions*, and *motif definitions*.

3.1 Graph Querying

Graph querying refers to the entire process of interacting with a graph database, from formulating high-level queries to retrieving results. Typically, users define the desired patterns using a specialized graph query language. While graph databases support other query types, such as traversal queries (e.g., "find the shortest path between two nodes"), our work focuses specifically on pattern-matching queries. Depending on the database (e.g., Neo4j [40], Amazon Neptune [2], ArangoDB [4]), users may use different query languages such as Cypher [17], Gremlin [49], or SPARQL [42] to express these patterns, which are collectively referred to as ***graph queries***. The graph database then parses and executes the graph queries, returning results in formats such as JSON, tables, or graph visualizations. ***Visual graph querying*** replaces the need to write queries in a graph query language with the construction of ***query graphs*** through interactions with a graphical interface. These query graphs, composed of nodes and edges with attribute constraints, enable users to express their desired patterns. The query graphs are then translated into graph query language statements and executed by the underlying graph database. In this work, users can construct a ***graph query representation*** to express their underspecified query intent, which may correspond to multiple specific query graphs. We refer to these derived, concrete query graphs as ***query instances***.

3.2 Graph Definitions

In this work, we focus on visual graph querying over directed and undirected *multivariate graphs*, where both nodes and edges can have associated attributes. A multivariate graph is defined as a tuple $G = (V, E, A_V, A_E, \Sigma)$, where V is a finite set of nodes, E is a set of edges, and Σ is a finite set of edge labels that allow multiple labeled edges between the same node pairs. The functions $A_V : V \to \mathscr{D}_V$ and $A_E : E \to \mathscr{D}_E$ assign attributes to nodes and edges, where the attribute values are denoted as $\mathscr{D}_V$ and $\mathscr{D}_E$ respectively. In a *directed multivariate graph*, $E \subseteq V \times V \times \Sigma$, where each edge is a tuple (u, v, ℓ) representing a directed edge from node u to node v with an optional label $\ell \in \Sigma$. In an *undirected multivariate graph*, $E \subseteq \{\{u, v\} \mid u, v \in V\} \times \Sigma$, where each edge connects an unordered node pair with label $\ell \in \Sigma$.

3.3 Motif Definitions

A ***motif*** refers to a small subgraph pattern that captures meaningful structural configurations within graphs. Such patterns have been extensively studied for their importance in understanding graph structures [67]. In this work, we introduce a quick configuration feature for four commonly used motif types (i.e., *path*, *loop*, *tree*, and *clique*) to help users easily construct their graph queries. We selected these four motif types due to their frequent use in existing motif-based network visualization research [10, 26, 30, 58], aligning with our focus on visual graph querying. The four motif types are described below:

- **Path:** A path P is defined as a sequence of vertices $v_1, v_2, \ldots, v_k$ (with $k \geq 2$) such that: (1) each consecutive pair of vertices v_i and v_{i+1} connected by an edge $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$; (2) in undirected graphs, the edge may also be $(v_{i+1}, v_i) \in E$); and (3) all vertices are distinct, i.e., $v_i \neq v_j$ for all $i \neq j$. In our approach, users can specify a desired range for the number of vertices in the path to support flexible query construction.
- **Loop:** A loop L is a closed path in graph G, defined by a sequence of vertices $v_1, v_2, \ldots, v_k$ (with $k \geq 3$) such that: (1) $v_1 = v_k$; (2) all intermediate vertices are distinct, i.e., $v_i \neq v_j$ for all $i \neq j$, except $i = 1, j = k$; and (3) each consecutive pair is connected by an edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$. In undirected graphs, edges may also be $(v_{i+1}, v_i) \in E$. Users can specify a range for the number of vertices in the loop.
- **Tree:** A tree T is a connected, acyclic subgraph with a designated root node r, defined over a set of vertices $v_1, v_2, \ldots, v_k$ (with $k \geq 2$) such that: (1) a unique simple path exists between any pair of vertices $u, v \in V_T$; and (2) the number of edges satisfies $|E_T| = |V_T| - 1$. In directed graphs (i.e., directed trees), all edges must follow a consistent direction from parent to child (i.e., $(v_i, v_j) \in E$ such that v_i is the parent of v_j). Users can specify a range for the number of vertices in the tree, as well as the width and depth.
- **Clique:** A clique C is a complete subgraph defined over a set of vertices $v_1, v_2, \ldots, v_k$ (with $k \geq 2$) such that: (1) for every pair of distinct vertices $u, v \in V_C$, an edge $(u, v) \in E$ exists; (2) in undirected graphs, edges are symmetric, i.e., $(u, v) \in E$ or $(v, u) \in E$; (3) in directed graphs, both edges $(u, v) \in E$ and $(v, u) \in E$ must exist for every pair. Users can specify a desired range for the number of vertices in the clique.

4 Informing the Design

This section presents our preliminary study and design requirements.

4.1 Preliminary Study

We conducted a preliminary study to comprehensively gather users' requirements regarding the expressiveness of visual graph querying. The details of the participants and procedures are as follows:

Participants: We invited four graph analysis experts (**E1–E4**) as participants. All of them are researchers from universities, each with over three years of experience in graph analysis across various domains, where retrieving subgraphs from graph data is a routine task. **E1** and **E2** possess expertise in querying using graph databases, while **E3** and **E4** analyze graphs through programming and were unfamiliar with graph query languages. Their feedback is also valuable because our goal is to enable users to conduct graph querying interactively without using complex query languages.

Procedures: The preliminary study was conducted in two sessions. In the first session, we conducted open-ended interviews with each participant. Initially, participants were asked to describe typical requirements for querying patterns within graph data in their routine research activities. We then introduced existing graph querying methods, including both query language-based and interactive visualization approaches. Participants were encouraged to describe any current or potential challenges in applying these methods to express and execute their previously-described requirements. In response to these challenges, we formulated initial design requirements and proposed an interactive framework for editing and executing graph queries. In the second session, participants were invited to evaluate these design requirements and the proposed framework. Their feedback guided us in finalizing the design requirements and refining the interactive framework.

The challenges (C1-C4) presented in Section 1 are summarized from the challenges proposed by participants. The design requirements and interactive framework are detailed in Section 4.2 and Section 5.

4.2 Design Requirements

We derived six design requirements (**R1-R6**) and grouped them into three categories: Query Expression (Expression), Query Verification (Verification), and Query Execution (Execution).

- **R1** Expression **Enable users to specify graph structure efficiently.** All experts (**E1-E4**) noted that constructing query graphs with numerous nodes and edges is time-consuming and could be simplified. As discussed in C1, user query intent may involve motifs (e.g, loops and paths) with underspecified parameters (e.g., the number of nodes), so our approach should allow users to specify such motifs efficiently. In addition, as manually creating repetitive structures is tedious (C2), our approach should also support expressing repetitive substructures quickly.
- **R2** Expression **Support underspecified query intent.** All experts (**E1-E4**) mentioned that a fixed query graph is insufficient to specify graph patterns that they wish to query (C1). For instance, in anomaly detection tasks, anomaly patterns often lack precise and concrete structures, and are typically expressed as soft structure constraints. The ideal query results are diverse and may vary in aspects such as node count or topology. Our approach should allow users to express such underspecified query intent, e.g., specifying motifs by approximate node counts and defining repetition ranges for particular structures.

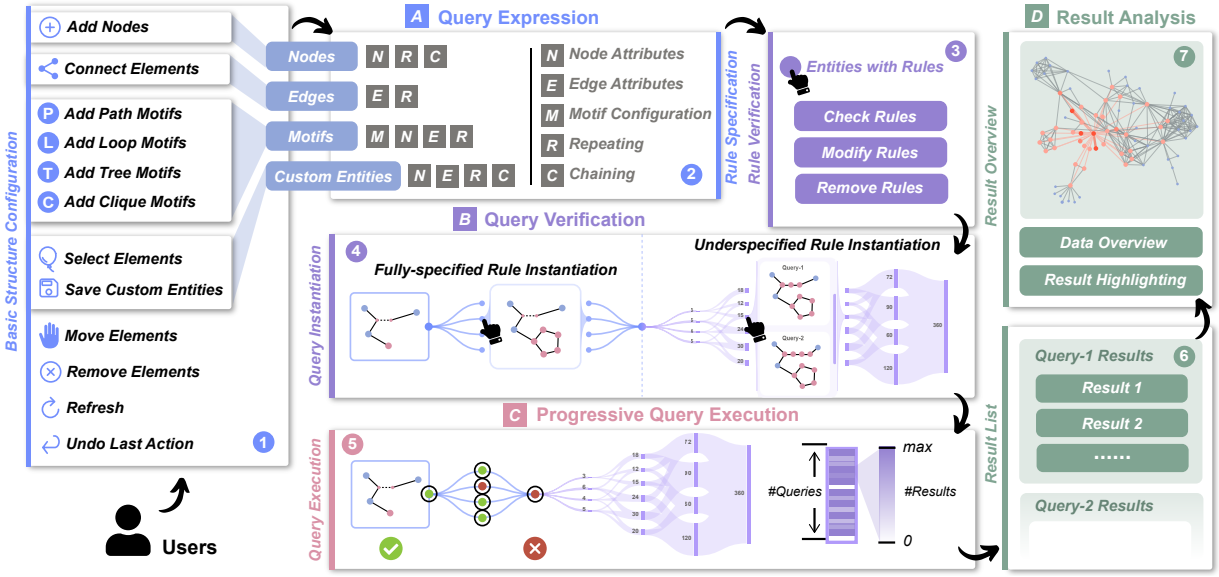

Fig. 2: The *Envisage* framework consists of four stages: (A) Query Expression, where users construct the basic structure of the query (1) and specify rules (2); (B) Query Verification, which allows users to inspect applied rules (3) and preview query instantiations (4); (C) Progressive Query Execution, where queries are progressively executed and refined based on intermediate results (5); and (D) Result Analysis, which presents the query outcomes through a detailed result list (6) and an overview visualization (7).

- **R3** Expression **Facilitate the expression and review of attribute constraints.** Attribute constraints are crucial in querying multivariate graphs [37], and all experts (**E1–E4**) agreed on the importance of specifying attribute constraints in queries. However, existing methods require users to assign constraints individually to nodes and edges, making batch specification difficult (C3). Our approach should allow users to easily specify constraints across multiple elements. Additionally, **E3** suggested providing an overview to facilitate constraint review and modification.
- **R4** Verification **Provide detailed query instances for verification.** Three experts (**E1-E3**) expected that an effective approach should allow them to instantiate detailed query graphs for verification (C4). **E1** emphasized that replacing specific query graphs with underspecified query expressions might result in queries that do not align with user expectations. Therefore, our approach should instantiate user-defined expressions into concrete graphs for an easy verification by users.
- **R5** Verification **Offer guidance for query modification.** A user-friendly visual query system should guide users in constructing correct queries [6]. Unexpected query instances or empty query results may be caused by issues in certain parts of a query expression. **E1** and **E2** suggested that our approach should offer a clear guidance to identify which parts deviate from their expectations, facilitating easy modifications.
- **R6** Execution **Execute user-defined queries and display the results.** All experts (**E1-E4**) emphasized that our approach should promptly execute them and display the results (C4). **E1** and **E2** suggested transforming user-defined query expressions into graph query language formats and executing them through a graph database to support precise subgraph matching. Additionally, **E4** noted that presenting query results within the context of the entire graph data helps users better understand result distributions and gain deeper insights for further query refinement.

5 *Envisage*

Informed by the above design requirements, we propose an interactive visual graph querying framework (Fig. 2) that supports efficient expression, verification, and execution of graph queries, which comprises four stages: *Query Expression*, *Query Verification*, *Progressive Query Execution*, and *Result Analysis*. We implemented this framework in a prototype system, *Envisage*[1] (Fig. 1), using JavaScript (Vue.js and D3.js) and backed by a Neo4j graph database. This section outlines each stage and the corresponding user interactions in *Envisage*.

5.1 Query Expression

The *Query Expression* stage (Fig. 2A) enables users to express their query intent through two phases: *basic structure configuration* and *rule specification*. In addition to standard graph construction, it supports efficient expression of underspecified query intent via advanced functions like customized motif configuration and parameterized rules.

Basic Structure Configuration: Initially, users can configure the basic structure of a query graph through a set of interactions (as shown in Fig. 2(1)), including four types of entities: nodes, edges, motifs, and customized entities. Motifs represent four common graph structures described in Section 3.3, allowing users to quickly build queries with specific patterns. For clarity and consistency, all external edges connected to motifs, except paths, are treated as connecting to a representative node within the motif (e.g., the root of a tree). Paths, however, expose both a head and a tail node for connection. A customized entity comprises a group of nodes, edges, and motifs marked by users, enabling batch operations such as setting attribute constraints.

Rule Specification: To support efficient query expression, we introduce five types of parameterized rules that can be applied to entities defined during the basic structure configuration stage. These rules enable users to specify structural or semantic constraints on nodes, edges, motifs, and customized entities within the query graph. As illustrated in Fig. 2(2), users can interactively assign rules to relevant entities to express specific query requirements. The definitions and functionalities of the five rule types are described in detail below.

- **Node Attribute Rules** specify attribute constraints for nodes within an entity, such as requiring a numerical attribute to exceed a threshold. These rules apply to individual nodes as well as to nodes within motifs or customized entities.
- **Edge Attribute Rules** specify attribute constraints for edges within an entity, such as requiring edge weights to fall within a certain range. These rules apply to individual edges as well as to edges within motifs or customized entities.

[1]Video demo available at https://youtu.be/d1KR5McgWPA

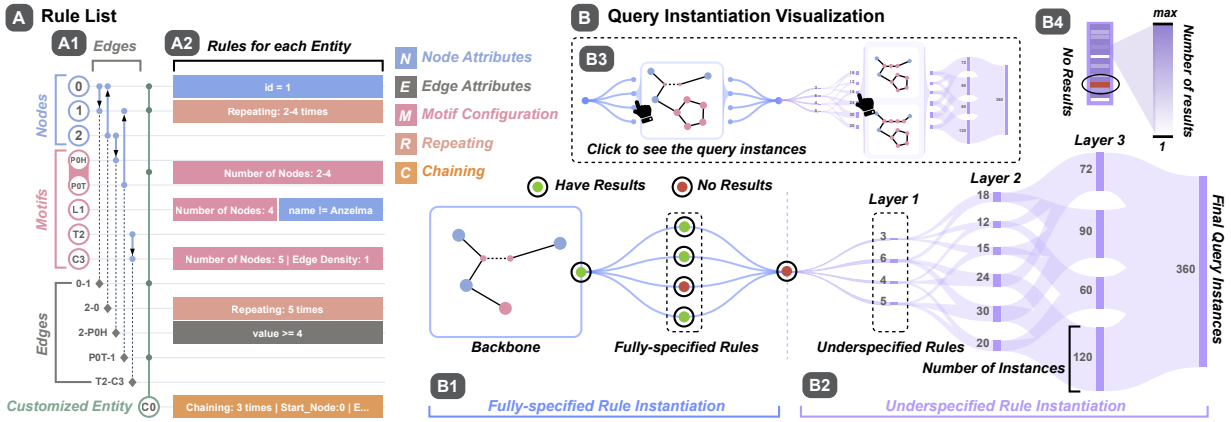

Fig. 3: The visual design of (A) the *Rule List* and (B) the *Query Instantiation Visualization*. The *Rule List* displays all user-defined entities along the y-axis (A1), with their corresponding rules listed alongside each entity (A2). The *Query Instantiation Visualization* illustrates how the graph query representation is instantiated into specific query instances based on both fully-specified rules (B1) and underspecified rules (B2). Users can interactively explore each query instance (B3), and the corresponding query results can be displayed within this view (B4).

- **Motif Configuration Rules** define the structure of motifs using parameters (e.g., number of nodes). To express the underspecified intent, users can assign value ranges to these parameters. Fig. 1A6 illustrates several subgraph instances that conform to a clique motif configuration with a specified node range of 4 to 6.
- **Repeating Rules** define how a substructure is repeated within the query graph by specifying parameters such as the number of repetitions and the entities to repeat. These rules are applicable to all entity types. When applied to a customized entity, the rule duplicates all included nodes, edges, and motifs and reconnects duplicated components to the same external entities as the original, thereby preserving the structural context. Similarly, repeating rules for nodes and motifs duplicate them while preserving external connections, whereas edge repetition duplicates only the edges themselves without duplicating the connected nodes. Fig. 1A4 and A5 showcase the use of repeating rules on a single node to express a star structure and a connector pattern, respectively.
- **Chaining Rules** define how a substructure is duplicated and connected into a chain-like pattern, which applies to both nodes and customized entities. Users can configure chaining rules by selecting a customized entity or node and specifying parameters, such as the start node, end node, number of chaining iterations, and chaining mode. Chaining rules duplicate the selected entities and connect the duplicated entities sequentially. The *chaining mode* determines how each duplication is connected: either by linking the start node of the new duplication to the end node of the previous duplication (Fig. 1A3), or by using the end node of each duplication as the start node for the next iteration (Fig. 4B1).

Node Attribute Rules and *Edge Attribute Rules* are fundamental features in some existing VGQ tools [44, 55]. However, our approach allows users to batch-apply these rules to customized entities and motifs (**R3**). *Motif Configuration Rules* enable users to efficiently construct query graphs with specific structures (**R1**) and express the underspecified intent regarding the size and shape of motifs (**R2**). *Repeating Rules* and *Chaining Rules* facilitate the rapid specification of the query graphs containing repetitive patterns (**R1**), and also allow users to express underspecified intent by specifying a range for the parameters (**R2**). To distinguish the users' query expressions (i.e., basic structures with parameterized rules) from concrete query instances, we refer to them as ***graph query representations***, as mentioned in Section 3.1.

Interactions with *Envisage*: *Envisage* provides a *Query Expression Panel* (Fig. 1A) that allows users to construct their graph query representations. The panel consists of two main components: the *Query Editor* (Fig. 1A1) and the *Rule List* (Fig. 1A2). In the *Query Editor*, users can add nodes and motifs by clicking buttons and reposition them via drag-and-drop. By clicking the "*Connect Elements*" button, users can draw edges between two entities. To create customized entities, users can select multiple entities by clicking on them and then finalize the selection by clicking the "*Save*" button. For rule specification, users can right-click on individual entities to open a pop-up menu, where rule parameters can be configured. The *Rule List* displays all entities and the corresponding rules applied to them, which will be described in detail in Section 5.2. For entities that are difficult to select directly in the *Query Editor* (e.g., customized entities), users can instead right-click on their labels in the *Rule List* to access the same pop-up menu.

5.2 Query Verification

Query Verification (Fig. 2B) is designed to help users confirm whether their query graph representations align with their expectations, which consists of *Rule Verification* and *Query Instantiation Verification*.

Rule Verification: Before executing queries, users can check whether the applied rules accurately reflect their query intent, and then modify or remove any incorrect or unintended rules as needed (Fig. 2③), using the *Rule List* (Fig. 3A). To facilitate rule verification, all entities are listed along the y-axis and grouped by type, while the rules applied to each entity are displayed as colored blocks with text, arranged horizontally to the right (Fig. 3A2). Different colors indicate different rule types, making it easier for users to distinguish and review them. By right-clicking a colored block, users can open a pop-up menu to modify and remove the corresponding rules. To help users identify which entities are connected by edges or included in customized entities, we offset the label positions of edges and customized entities, draw vertical lines linking them to their corresponding entities, and add arrows to edges to indicate the direction from source to target. This design is inspired by *Massive Sequence View* [56] and *BioFabric* [19], which effectively encodes graph structural information in node-list layouts. Additionally, users can hover over an entity to highlight it and its related entities in both *Query Editor* and *Rule List*, establishing visual connections between the two components.

Query Instantiation Verification: In line with **R4**, we generate all valid query instances based on user-defined query representations for verification. To help users understand how these instances are derived and identify any misalignment with their query intent (**R5**), we visually present the *query instantiation* process (Fig. 2④) based on user-defined rules, which includes two phases: ***fully-specified rule instantiation*** and ***underspecified rule instantiation***. *Fully-specified rules* are those in which all parameters are fixed values, resulting in a single query instance. In contrast, *underspecified rules* include at least one parameter defined as a value range, capturing flexible user intent and producing multiple possible query instances. The above definitions only relate to three of the four rule types (i.e., motif configuration, repeating, and chaining). Attribute rules do not affect instance variation and are directly assigned to the relevant nodes and edges in query instances. The details of both phases are discussed below:

- **Fully-specified Rule Instantiation:** We first construct a *backbone* based on users' query representations, where entities with repeating and chaining rules appear only once. Motifs other than paths are abstracted as single nodes, while paths are represented by two nodes (a head node and a tail node) connected by a path. This abstraction preserves the fundamental structure of user-defined queries, making them easier to understand as a starting point for instantiation. We begin by expanding the backbone using each fully-specified rule, then apply all these rules collectively, and finally visualize the resulting query instances for users. This visualization helps users understand the final fully-specified query instance while allowing them to inspect individual rules to guide modifications (**R5**).
- **Underspecified Rule Instantiation:** Each underspecified rule can generate multiple possible query instances, and combining several such rules can lead to a large number of combinations. To manage this complexity, we start from the final fully-specified query instance and incrementally generate all possible combinations of underspecified rules and their corresponding instances. This process follows a layered strategy: each layer represents instances generated by applying a specific number of underspecified rules. For example, Layer 2 includes all instances formed by applying any two underspecified rules to the fully specified base. The final layer includes instances that satisfy all user-defined rules. This layer-by-layer organization allows users to efficiently review the query generation process (**R4**) and identify rule combinations that cause unintended outcomes for further modification (**R5**).

During the *query instantiation* process, motifs are expanded into specific nodes and edges, while entities with repeating or chaining rules are duplicated according to the rule descriptions in Section 5.1. The query instances consist solely of nodes and edges, with their *node attribute rules* and *edge attribute rules* accordingly.

Interactions with *Envisage*: For *rule verification*, users can begin by browsing all applied rules (colored blocks with text) associated with each entity. By hovering over an entity of interest, users can highlight the corresponding rules and related entities in both the *Rule List* and the *Query Editor*, enhancing the clarity. When finding unintended rules, users can right-click on the colored blocks and edit or remove them in the pop-up menu (Fig. 1A2). For *query instantiation verification*, *Envisage* provides a ***query instantiation visualization*** (Fig. 3B) to illustrate the instantiation process. On the left, a node-link diagram shows the backbone structure (Fig. 3B1) linked to several circles, with each circle representing a fully-specified rule. Clicking a circle shows the corresponding query instance in another node-link diagram (Fig. 3B3), allowing users to verify whether each rule expands the backbone as intended. These circles converge into a single circle representing the final fully-specified query instance, which can also be clicked to review and confirm whether all fully-specified rules are correctly applied. This final instance is further linked to a Sankey diagram-based design to display the *underspecified rule instantiation* (Fig. 3B2). The x-axis layer index indicates the number of underspecified rules selected in each combination. For example, Layer 1 includes individual rules, Layer 3 includes all combinations of three rules from the full rule set, and the final layer represents the full combination of all rules. Each rectangle within a layer represents a rule combination, with its height indicating the number of query instances generated by that combination. The flows between adjacent layers indicate that each combination in the next layer includes multiple combinations from the previous layer. For example, the combination [0,1,2] in Layer 3 contains the subsets [0,1], [0,2], and [1,2]. By clicking these rectangles, users can review and compare the query instances generated by different rule combinations to identify which underspecified rules produce unintended results (Fig. 3B3). Additionally, rule descriptions and the number of instances are displayed in text alongside corresponding visual elements for clarity.

5.3 Progressive Query Execution

The generated query instances exist as graph structures with attribute constraints, rather than in graph query language statements that are directly executable by the graph database. To support execution, we introduce a *query translator* that converts these instances into graph query language statements. However, executing all final query instances at once can be time-consuming. If the results are unavailable or unexpected, users must iteratively modify the queries and re-execute them, making it difficult to diagnose issues and guide modifications (**R5**). To ensure smooth execution of user-defined query instances (**R6**), we introduce a *progressive execution* process (Fig. 2C). The *query instantiation visualization* contains multiple intermediate steps: the backbone, individual fully-specified rules, the final specific instance, and all layers of underspecified rule combinations. Each step builds on the results of the previous one, forming a structured and traceable execution flow. By executing queries progressively, users can easily identify which rules cause failures. If a step returns no results, all subsequent instances depending on it are also guaranteed to fail (Fig. 2⑤). To speed up this process, users can initially limit the number of returned results and execute the steps incrementally, allowing them to quickly identify and adjust problematic rules. Once the queries are refined, users can execute the final queries to retrieve the complete set of results.

Interactions with *Envisage*: The progressive query execution workflow is integrated into the *Query Execution Panel* via *query buttons* (Fig. 1B). Each intermediate step has a query button placed below it, enabling users to execute queries and view the results. When users click the query button for an individual query instance (represented by circles), *Envisage* translates it into query language through the query translator and executes it in Neo4j database [40]. The result is visually encoded: a green circle indicates a successful match, while a red circle indicates no results (Fig. 3B1). The number of results is displayed alongside the circle as text. In the Sankey diagram-based design, each rectangle represents a combination of multiple underspecified rules and may include several query instances, each with its own result set. Since the height of the rectangle reflects the number of query instances, we subdivide it into smaller vertically-arranged rectangles, each representing a single query instance. To enhance interpretability, we use a gradient purple color to visually encode the number of results for each query instance. The instances with no results are shown in red (Fig. 3B4). Users can selectively execute query instances at any intermediate step with a limited number of returned results, enabling them to quickly assess whether queries are functioning as intended and iteratively refine rules to ensure meaningful results.

5.4 Result Analysis

In *progressive query execution*, users can determine whether query instances produce results but still require an in-depth analysis of each instance's results (**R6**). The Result Analysis stage (Fig. 2D) aims to present results through both an overview and a detailed examination. In the *result overview* (Fig. 2⑦), we first compute the frequency of all nodes and edges based on the results that users wish to analyze. These frequencies are then visualized and highlighted within the context of the entire graph data. This overview enables users to identify result distributions across the entire graph and detect potential issues, such as an excessive concentration of results within a single subgraph. Additionally, users can examine individual query results in detail, including the graph structure and its associated properties (Fig. 2⑥).

Interactions with *Envisage*: *Envisage* provides a *Result Overview* (Fig. 1C) and a *Result List* (Fig. 1D) that display the corresponding query results when users click on circles or rectangles representing query instances that have returned results. Specifically, the *Result Overview* presents the entire graph as a node-link diagram, with result frequencies highlighted using a gradient red color scheme. The *Result List* displays information cards, each representing the results of a single query instance. Each card includes statistics (e.g., node count), a node-link diagram showing one result, and a sidebar for switching between results of that instance. Only one result is shown at a time, as all results for a given instance share the same graph structure but differ in labels. Users can browse these labels via the sidebar (Fig. 1D) and navigate between different instances using a scroll bar.

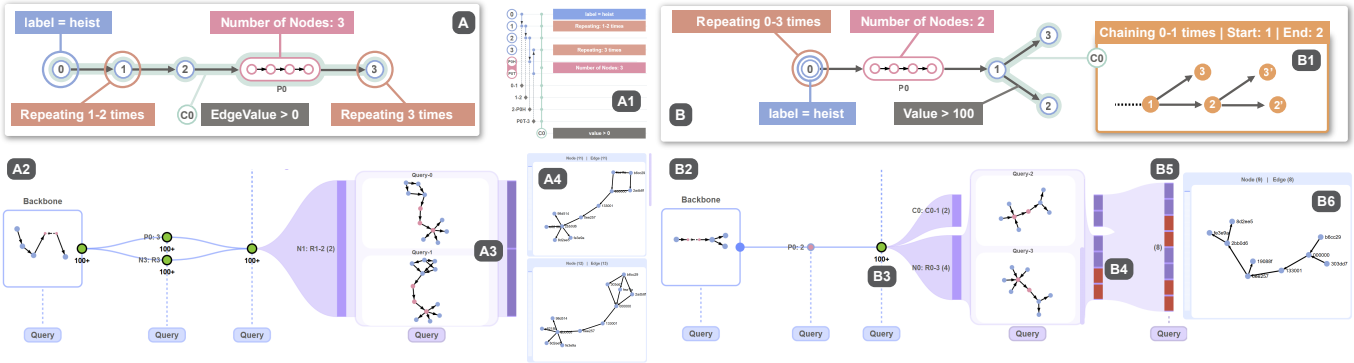

Fig. 4: Money laundering pattern detection using *Envisage*. (A) The user defined a dispersion–convergence–dispersion pattern by applying multiple rules (A1), executed the query progressively (A2–A3), and successfully retrieved matching results (A4). (B) She constructed a chain-shaped pattern using chaining and repeating rules (B1) and used progressive execution (B2–B4) to identify over-repetition issues as shown in the results (B5-B6).

6 Case Study

This section presents two case studies demonstrating the effectiveness of *Envisage*. To evaluate *Envisage*, we conducted user interviews with 14 graph analysts (**U1-U14**), which will be discussed in Section 7. The cases in this section illustrate how **U1** and **U9** utilized *Envisage* for specific querying tasks. To assess the generalizability of *Envisage*, we incorporated three graph datasets of varying scales and from different domains for user selection to conduct the graph querying. The details of these datasets are described as follows:

- **Les Misérables Co-occurrence Network (LMCN):** This dataset involves co-occurrence relationships between characters in Victor Hugo's novel Les Misérables [60]. It is a classical undirected graph dataset for graph analysis [32], consisting of 77 nodes and 254 edges. Each node corresponds to a character, and an edge connects two characters if they appear in the same chapter.
- **Function Call Graph (FCG):** We select four samples from the public function call dataset, *Malicious Webshell Family (MWF)* [66], to construct this dataset, which comprises 305 nodes and 4,516 edges. Nodes represent functions, while edges denote function calls with multiple attributes, including the caller function, callee function, parameters, and return values.
- **Money Laundering Network (MLN):** This dataset captures an Ethereum-based money laundering event, named *eazyfihacker* [48], selected from the public Ethereum Money Laundering dataset, *EthereumHeist* [61]. It contains 1,335 nodes and 8,960 edges, where nodes represent blockchain accounts and edges denote transactions between them.

6.1 Case 1: In-depth Analysis of Money Laundering Patterns

U1 expressed a particular interest in the money laundering dataset (MLN) and aimed to leverage *Envisage* to identify characteristic patterns associated with money laundering activities.

Layering–Integration–Layering: As a first step, **U1** sought to investigate the presence of Layering–Integration–Layering patterns within the network, which means that illicit funds are initially dispersed across multiple accounts (layering), consolidated into a single account (integration), and then dispersed again (layering). To analyze this pattern, she constructed a query representation, as illustrated in Fig. 4A. Specifically, she applied a node attribute rule "*label=heist*" (*heist* label marks the source node in MLN dataset) to the initial node (*Node 0*) to mark it as the suspected origin of the money laundering flow. She then added two additional nodes (*Node 1* and *Node 2*), connecting them sequentially to model the initial money flow. To express the Layering–Integration structure, she applied a repeating rule to *Node 1*, allowing it to repeat 1–2 times. To capture subsequent layering, she appended a directed path (*P0*) consisting of three nodes and added *Node 3* to represent the further flow of laundered funds. A repeating rule was then applied to *Node 3* to express the final layering process. She then grouped all entities into a customized entity (*C0*) and applied an edge attribute rule, "*value>0*", to ensure only transactions with positive monetary value were considered. After constructing the query, **U1** reviewed the defined rules (Fig. 4A1) and the generated query instances (Fig. 4A3) to verify that the structure aligned with her intended pattern. She then executed intermediate steps and the final query instances by interacting with the query interface (Fig. 4A2). The appearance of green circles and purple blocks indicated that each step returned results successfully. Finally, in the *Result List* (Fig. 4A4), she confirmed that the dataset did indeed contain instances matching the Layering–Integration–Layering pattern of money laundering behavior.

Chain-shaped Money Laundering: **U1** then turned her attention to investigating a chain-shaped money laundering pattern and constructed a corresponding query representation, as shown in Fig. 4B. Similar to the previous pattern, she designated *Node 0* as the source node and added a directed path (*P0*) to represent the primary flow of illicit funds. To simulate branching behavior within the laundering process, she added three additional nodes (*Node 1*, *Node 2*, and *Node 3*), thereby creating two diverging branches from the main path. She grouped these nodes into a customized entity (*C0*) and applied a chaining rule, specifying *Node 1* as the start node and *Node 2* as the end node, to expand the query in a chain-like structure, as illustrated in Fig. 4B1. Then, she applied an attribute rule, "*value>100*", to the edge from *Node 0* to *Node 1*, marking it as the main branch to differentiate it from auxiliary paths. Additionally, **U1** hypothesized that multiple source nodes might initiate transactions along this chain-shaped pattern, and she applied a repeating rule to the source node (*Node 0*). After executing the final query instances, she observed that half of them returned results (Fig. 4B6), while the other half returned no results, indicated by red rectangles in Fig. 4B5. To diagnose the issue, **U1** progressively executed the intermediate steps (Fig. 4B3 and B4) and discovered that the red rectangles appeared specifically in the instances generated by the repeating rule applied to the *source* node. This indicated that while patterns where the heist node repeated 0 or 1 times produced valid results, those involving 2–3 repetitions did not yield any matches.

Using *Envisage*, **U1** successfully expressed and verified her desired money laundering patterns within the dataset.

6.2 Case 2: Rapid Character Relationship Querying

U9 has read the novel *Les Misérables* and wanted to explore the relationships between its main characters by querying the character co-occurrence network (LMCN).

Efficient Query Construction: **U9** aimed to explore the relationships between the protagonist, *Valjean*, and various communities (e.g., families or friend groups) by constructing a query, as shown in Fig. 5A. He began by adding *Node 0* and applying a node attribute rule, "*name=Valjean*". He then connected this node to a 5-node clique motif (*C0*), assuming five members as a reasonable size for a community. To express multiple communities, **U9** intended to repeat the clique motif but was unsure how many such groups that *Valjean* would be associated with, so he configured *C0* to repeat 0-3 times. After reviewing the instantiation visualization (Fig. 5A1), **U9** confirmed that *Envisage*

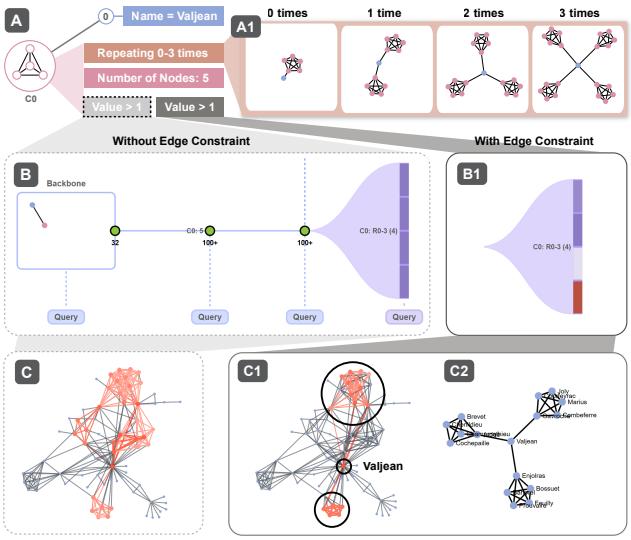

Fig. 5: Community analysis in *Les Misérables* using *Envisage*. (A) The user queried *Valjean*'s connections to multiple communities (A1). (B) All query instances returned results, while applying the edge constraint led to fewer or no results for some instances (B1). By checking the highlighted results (C, C1, C2), the user identified two distinct communities connected to Valjean, corresponding to two major plotlines in the novel.

correctly generated the query instances involving one to four communities connected to Valjean. Upon executing these instances through the *Query Execution Panel*, **U9** observed that all query instances returned results (the green circles and purple blocks), indicating that *Valjean* co-occurred with at least four distinct communities, each consisting of five or more characters. The characters in these communities are highlighted in the *Result Overview*, as shown in Fig. 5C.

Insightful Result Comparison: **U9** considered that character co-occurrences happening only once might be incidental, potentially leading to the inaccurate identification of communities. To address this concern, he applied an edge attribute rule, "*value>1*", to the clique motif (*C0*), thereby constraining all edges within *C0* to reflect stronger and more meaningful co-occurrence relationships of communities. After re-executing the query instances with this constraint, **U9** observed that the results differed from those generated without the edge value rule (Fig. 5B1), highlighting how edge strength influences community detection. Specifically, query instances where *Valjean* was connected to one to three communities still returned valid results, whereas the instance involving four communities yielded no results, as indicated by the red rectangles in Fig. 5B1. The rectangle representing Valjean with three communities was colored light purple, suggesting a low number of matching results. Upon examining the *Result Overview* (Fig. 5C1) and the *Result List* (Fig. 5C2), **U9** found that although three communities were returned, they belonged to only two distinct communities, as shown by the two circled clusters in Fig.5C1. By hovering over the nodes to view their names, **U9** discovered that the cluster in the upper circle primarily consisted of members of "*The Friends of the ABC*", a revolutionary group that once fought alongside Valjean. The cluster in the lower circle included key characters involved in a subplot of an innocent man being mistaken for *Valjean*.

With Envisage, **U9** quickly expressed and executed desired queries to explore *Valjean*'s community relationships and intuitively uncover key insights, including two main plotlines in the novel.

7 User Interview

To evaluate the effectiveness of *Envisage*, we conducted in-depth user interviews with 14 graph analysts. This section presents the participants, procedures, and key findings.

7.1 Participants and Apparatus

We recruited 14 participants (U1-U14) from universities for user interviews (6 females, 8 males, $age_{mean} = 24$, $age_{sd} = 2.95$, with normal vision and no color vision deficiency). All participants were majoring in fields related to computer science and had at least six months of experience in graph analysis. They comprised ten graduate students (U1-U3, U5-U8, and U12-U14), three undergraduates (U9-U11), and one postdoctoral researcher (U4). All participants had prior experience writing code to extract information from graph data. However, only five participants (U5, U9–U12) were familiar with data querying languages or tools: U9 with Cypher, U10 and U11 with SQL, and U5 and U12 with NetworkX [41] and Cytoscape [14]. All interviews were conducted online via Zoom. *Envisage* was deployed on a remote server, and participants accessed it using their own laptops or desktops while sharing their screens with the interviewer. Each interview lasted about one hour, and we paid $15 to each participant for compensation.

7.2 Procedures

The user interviews consisted of tutorial, task, and interview phases. In the tutorial phase, participants were first asked to access the online *Envisage* system, and we then introduced its background, workflow, visual design, and interactions. A usage scenario illustrated how *Envisage* supports graph querying in practice. We grouped participants into three groups based on their expertise and preferences, each working with a different dataset described in Section 6. In the task phase, we first introduced the assigned datasets and then guided participants through a series of instructions to perform graph querying. These instructions were designed to ensure that participants understood and used the core functions of *Envisage*. Afterward, participants were allowed to freely query specific patterns of their interest following our four-stage framework until they gained a comprehensive understanding of how *Envisage* works. The entire task phase typically lasted about 30 minutes. Finally, participants completed a post-study questionnaire, which included 14 questions (Q1-Q14), as shown in Fig. 6. Q1-Q12 were closed-ended, rated via a 7-point Likert scale [29] (1 for strongly disagree and 7 for strongly agree), and assessed *Envisage* support for query expression (Q1-Q4), verification (Q5, Q6), execution (Q7, Q8), and usability (Q9-Q12). The usability questions (Q9-Q12) were selected from System Usability Scale (SUS) Questionnaire [9].

7.3 Results

Fig. 6 shows participants' feedback for closed-ended questions (Q1-Q12), including the score distribution along with the calculated mean (M) and standard deviation (SD) for each question. The questionnaire results indicate overall positive user feedback for *Envisage* across key dimensions, while there was some negative feedback, which will be discussed in the detailed result analysis below.

Expression: Participants responded positively to *Envisage*'s ability to support expressive and flexible graph query construction. They agreed it helped specify queries efficiently (Q1: M = 5.50) and express underspecified intent (Q2: M = 6.07), with high ratings for configuring attribute constraints (Q3: M = 6.14) and overall intent expression (Q4: M = 6.00). However, some participants gave neutral responses, particularly to Q1 and Q4, primarily due to limitations in representing certain complex structures. For instance, U6 attempted to express two cliques connected by multiple edges across different nodes, but *Envisage* treated inter-clique connections as linking to the same nodes. This simplification was necessary to prevent exponential growth in the number of generated query graphs as the node count increased. Similarly, U8 wanted to construct more complex or irregular query structures beyond the current support for repeating and chaining rules. Moving forward, these limitations could be addressed by enabling users to fine-tune generated instances or by translating query intent into graph matching programs rather than relying solely on enumerated query graphs and query languages.

Verification & Execution: *Envisage* was highly rated for its support for verification, with users valuing the correctness of generated query instances (Q5: M = 6.36, SD = 0.63) and the clarity of query instantiation visualizations (Q6: M = 6.43, SD = 0.51). For execution, users appreciated the progressive execution feature, which helped guide query modifications (Q7: M = 6.00, SD = 1.18) and overall support for execution (Q8: M = 5.57, SD = 1.40). Some users gave

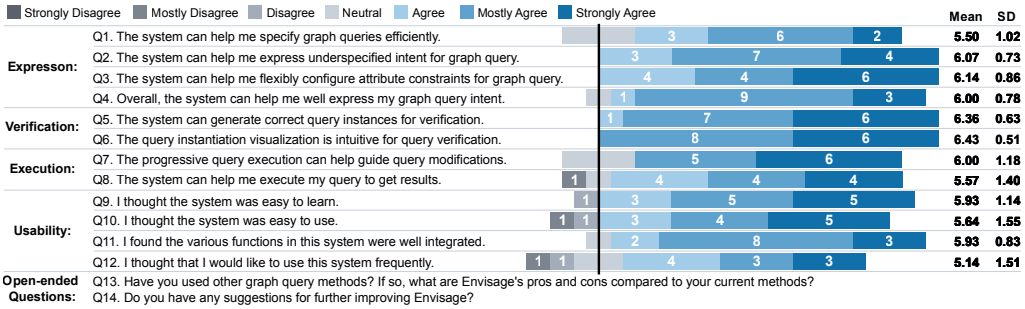

Fig. 6: The user interview questionnaire results. Q1-Q12 are closed-ended questions rated on a 7-point Likert scale. Q13, Q14 are open-ended questions to collect participants' feedback. The detailed scores of Q1-Q12 are shown in a stacked bar chart.

neutral or negative feedback on Q7 and Q8 due to slower response times when processing a large number of complex query instances, which impacted interaction smoothness. To align with underspecified query intent, the system generates many possible query instances, placing heavy demands on computational resources, particularly memory. This issue could be alleviated by deploying the system on more powerful servers or incorporating techniques such as graph neural networks [20] to accelerate querying rather than relying solely on database queries.

Usability: Overall, participants found *Envisage* easy to learn (Q9: M = 5.93) and use (Q10: M = 5.64), with well-integrated features (Q11: M = 5.93). However, some neutral and negative feedback revealed areas for improvement. U7 noted the need for more detailed guidance, results statistics, and visual indicators during query processing, while U2 noted that repetitive basic interactions like adding nodes and edges can be simplified. The slightly lower score for intended frequent use (Q12: M = 5.14) was attributed to difficulties integrating *Envisage* into existing workflows, which often involve additional steps such as deriving new attributes or converting outputs into domain-specific formats.

Open-ended Questions: In Q13, participants compared *Envisage* with other graph query methods. Those unfamiliar with such tools shared impressions based on a brief introduction of existing systems during user interviews. Overall, participants found *Envisage* more expressive than traditional visual query tools, with features like query expansion rules helping them quickly express desired patterns. Participants familiar with graph query languages (U9 and U12) noted that these languages still offer greater expressive power, particularly for advanced operations such as subqueries, unions, and intersections. However, most participants noted that writing such queries remains time-consuming and laborious, even for experienced users. As a result, *Envisage* is well-suited for querying complex graphs involving specific (e.g., motifs) and regular patterns (e.g., repeating structures), especially for novice users. In response to Q14, U2 and U7 suggested incorporating a feature that allows users to describe their query intent in natural language. U2 further noted that while some recent studies have used large language models (LLMs) to generate Cypher queries [24], these approaches remain limited in terms of query complexity, generation accuracy, and intuitive visual design. Therefore, combining LLMs with visual graph querying could be a promising direction for future exploration. U12 also suggested making motif configuration more flexible, such as allowing customization of individual elements within a motif and enabling users to fine-tune query instances after instantiation.

8 Discussion

This section discusses key lessons learned about the expressiveness of visual graph querying and outlines our limitations.

8.1 Expressiveness of Visual Graph Querying

A primary goal of visual graph querying is to enable users to interactively construct graph queries that reflect their intent without relying on complex code or formal query languages [6]. However, user intent is often diverse and loosely defined. Existing approaches, such as "query-by-template", offer limited support for expressing more complex or underspecified queries. This work investigates how visual graph querying can support flexible query expression. Specifically, users can apply multiple rules to customized entities to construct complex query graphs aligned with their goals. Our prototype incorporates four common motifs and two additional parameterized rules (repeating and chaining) to assess the feasibility of this approach. Future extensions could support more customized motifs and rule types for broader expressiveness. User interviews revealed that *Envisage* could express the most user-desired patterns, though minor gaps remained. One potential solution is to allow users to sketch rough patterns with current functions and fine-tune them as needed. Additionally, while prior work has used large language models (LLMs) for graph query generation [24], these approaches struggle with complex patterns and are challenging to revise for users unfamiliar with query languages. Combining the intent-capturing ability of LLMs with the visual expressiveness of systems like *Envisage* presents a promising direction for more accessible and flexible graph querying.

8.2 Limitations

Envisage is not without limitations: First, user queries are represented as subgraphs with attribute constraints, which can be translated into various query languages by extending the translators. However, both translation and execution performance may vary across languages. For example, Cypher queries can become verbose and slow when expressing large patterns, potentially impacting execution speed. Some languages like Gremlin [49] offer more efficient traversal constructs, but this comes at the cost of increased translator complexity. Second, *Envisage* can benefit from integrating existing techniques such as query auto-completion [64] and auto-suggestion [27]. Third, the Result List can become difficult to navigate when a large number of results are returned, which could be mitigated by incorporating a dropdown menu or pagination. Additionally, *Envisage* could incorporate additional interactions to support the expression of temporal patterns. For example, in MLN dataset, edges represent timestamped transactions, and one participant aimed to specify temporal relationships between edges.

9 Conclusion

We presented *Envisage*, an interactive visual graph querying system designed to improve expressive power in query construction. *Envisage* adopts a four-stage framework that guides users through the processes of expressing, verifying, and executing graph queries. To evaluate it, we conducted two case studies and in-depth interviews with 14 graph analysts. The results show that *Envisage* effectively supports users in building, refining, and executing complex queries with improved scalability and flexibility in handling underspecified query intent.

In future work, we plan to allow users to customize rules when expressing graph queries, enhancing *Envisage*'s adaptability across domains. We also aim to integrate natural language with visual interaction, allowing users to express query intent in natural language and refine generated query instances through an interactive visual interface.

Acknowledgments

This project is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Proposal ID: T2EP20222-0049), and NTU Start Up Grant awarded to Yong Wang.

References

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the International Conference on Management of Data*, pp. 431–446. ACM, 2016. doi: 10.1145/2882903.2915213 2

[2] Amazon Web Services, Inc. Amazon neptune: Fully managed graph database service. https://aws.amazon.com/neptune/. [Online; accessed 24-March-2025]. 3

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using Worst-Case optimal low-memory dataflows. *arXiv preprint*, Feb. 2018. doi: 10.48550/arXiv.1802.03760 2

[4] ArangoDB GmbH. Arangodb: Multi-model database for your modern apps. https://arangodb.com/. [Online; accessed 24-March-2025]. 3

[5] S. S. Bhowmick and B. Choi. Data-driven visual query interfaces for graphs: Past, present, and (near) future. In *Proceedings of the International Conference on Management of Data*, pp. 2441–2447. ACM, Philadelphia, PA, USA, June 2022. doi: 10.1145/3514221.3522562 2

[6] S. S. Bhowmick, B. Choi, and C. Li. Graph querying meets hci: State of the art and future directions. In *Proceedings of the International Conference on Management of Data*, pp. 1731–1736. ACM, Chicago, May 2017. doi: 10.1145/3035918.3054774 1, 2, 4, 9

[7] S. S. Bhowmick, B. Choi, and S. Zhou. VOGUE: Towards a visual interaction-aware graph query processing framework. In *Proceedings of the Conference on Innovative Data Systems Research*, 2013. 1, 2

[8] S. S. Bhowmick, H. E. Chua, B. Thian, and B. Choi. ViSual: An hci-inspired simulator for blending visual subgraph query construction and processing. In *Proceedings of the 31st International Conference on Data Engineering*, pp. 1480–1483. IEEE, Seoul, Apr. 2015. doi: 10.1109/ICDE.2015.7113406 2

[9] J. Brooke. SUS: A retrospective. *Journal of usability studies*, 8(2):29–40, 2013. doi: doi/abs/10.5555/2817912.2817913 8

[10] E. Cakmak, J. Fuchs, D. Jäckle, T. Schreck, U. Brandes, and D. Keim. Motif-based visual analysis of dynamic networks. In *Proceedings of the Visualization in Data Science*, pp. 17–26. IEEE, Oct. 2022. doi: 10.1109/VDS57266.2022.00007 3

[11] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. Graphite: A visual query system for large graphs. In *Proceedings of the International Conference on Data Mining Workshops*, pp. 963–966. IEEE, 2008. doi: 10.1109/ICDMW.2008.99 1, 2

[12] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct. 2004. doi: 10.1109/TPAMI.2004.75 2

[13] E. Cuenca, A. Sallaberry, D. Ienco, and P. Poncelet. Vertigo: A visual platform for querying and exploring large multilayer networks. *IEEE Transactions on Visualization and Computer Graphics*, 28(3):1634–1647, 2021. doi: 10.1109/TVCG.2021.3067820 1, 2

[14] Cytoscape Consortium. Cytoscape: An open source platform for complex network analysis and visualization. https://cytoscape.org/, 2024. [Online; accessed: 28-March-2025]. 8

[15] C. Dunne and B. Shneiderman. Motif simplification: Improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 3247–3256. ACM, Paris, France, Apr. 2013. doi: 10.1145/2470654.2466444 2

[16] S. Egi. Loop patterns: Extension of kleene star operator for more expressive pattern matching against arbitrary data structures. *arXiv preprint*, Sept. 2018. doi: 10.48550/arXiv.1809.03252 2

[17] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1433–1445. ACM, 2018. doi: 10.1145/3183713.3190657 1, 3

[18] L. Freeman et al. The development of social network analysis. *A Study in the Sociology of Science*, 1(687):159–167, 2004. 1

[19] J. Fuchs, F. L. Dennig, M. Heinle, D. A. Keim, and S. Di Bartolomeo. Exploring the design space of biofabric visualization for multivariate network analysis. *Computer Graphics Forum*, 43(3):e15079, June 2024. doi: 10.1111/cgf.15079 5

[20] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30:1025–1035, 2017. doi: 10.5555/3294771.3294869 2, 9

[21] M. Han, H. Kim, G. Gu, K. Park, and W. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the International Conference on Management of Data*, pp. 1429–1440. ACM, Amsterdam, July 2019. doi: 10.1145/3299869.3319880 2

[22] W. Han, J. Lee, and J. Lee. TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the International Conference on Management of Data*, pp. 337–348. ACM, New York, June 2013. doi: 10.1145/2463676.2465300 2

[23] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the International Conference on Management of Data*, pp. 405–418. ACM, Vancouver, June 2008. doi: 10.1145/1376616.1376660 1, 2

[24] M. Hornsteiner, M. Kreussel, C. Steindl, F. Ebner, P. Empl, and S. Schönig. Real-time text-to-cypher query generation with large language models for graph databases. *Future Internet*, 16(12):438, Dec. 2024. doi: 10.3390/fi16120438 9

[25] K. Huang, H. Liang, C. Yao, X. Zhao, Y. Cui, Y. Tian, R. Zhang, and X. Zhou. Visualneo: Bridging the gap between visual query interfaces and graph query engines. *Proceedings of the VLDB Endowment*, 16(12):4010–4013, Sept. 2023. doi: 10.14778/3611540.3611608 2

[26] W. Huang, C. Murray, X. Shen, L. Song, Y. X. Wu, and L. Zheng. Visualisation and analysis of network motifs. In *Proceedings of the Ninth International Conference on Information Visualisation*, pp. 697–702, 2005. doi: 10.1109/IV.2005.138 3

[27] N. Jayaram, S. Goyal, and C. Li. VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation. *Proceedings of the VLDB Endowment*, 8(12):1940–1943, Aug. 2015. doi: 10.14778/2824032.2824106 9

[28] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. Prague: Towards blending practical visual subgraph query formulation and query processing. In *Proceedings of the 28th International Conference on Data Engineering*, pp. 222–233. IEEE, Washington, Apr. 2012. doi: 10.1109/ICDE.2012.49 2

[29] A. Joshi, S. Kale, S. Chandel, and D. K. Pal. Likert scale: Explored and explained. *British Journal of Applied Science & Technology*, 7(4):396–403, 2015. doi: 10.9734/BJAST/2015/14975 8

[30] S. Jung, D. Shin, H. Jeon, K. Choe, and J. Seo. MoNetExplorer: A visual analytics system for analyzing dynamic networks with temporal network motifs. *IEEE Transactions on Visualization and Computer Graphics*, 30(10):6725–6739, Oct. 2024. doi: 10.1109/TVCG.2023.3337396 3

[31] A. Jüttner and P. Madarasi. VF2++: An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, June 2018. Computational Advances in Combinatorial Optimization. doi: 10.1016/j.dam.2018.02.018 2

[32] D. E. Knuth. The stanford graphbase: A platform for combinatorial algorithms. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 41–43. ACM/SIAM, Austin, Jan. 1993. doi: rec/conf/soda/Knuth93 7

[33] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112, 2019. doi: 10.14778/3339490.3339494 2

[34] C. Li, Y. Tang, Z. Tang, J. Cao, and Y. Zhang. Motif-based embedding label propagation algorithm for community detection. *International Journal of Intelligent Systems*, 37(3):1880–1902, 2022. doi: 10.1002/int.22759 1

[35] G. Li, H. Mi, C. H. Liu, T. Itoh, and G. Wang. HiRegEx: Interactive visual query and exploration of multivariate hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 2024. doi: 10.1109/TVCG.2024.3456389 2

[36] D. Lin, J. Wu, Y. Yu, Q. Fu, Z. Zheng, and C. Yang. Denseflow: Spotting cryptocurrency money laundering in ethereum transaction graphs. In *Proceedings of the ACM on Web Conference 2024*, pp. 4429–4438, 2024. doi: 10.1145/3589334.3645692 1

[37] J. Ma, S. S. Bhowmick, L. Tay, and B. Choi. SIERRA: A counterfactual thinking-based visual interface for property graph query construction. In *Companion of the 2024 International Conference on Management of Data*, pp. 440–443. ACM, New York, June 2024. doi: 10.1145/3626246.3654729 2, 4

[38] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment*, 12(11):1692–1704, 2019. doi: 10.14778/3342263.3342643 2

[39] J. J. Miller. Graph database applications and concepts with neo4j. In *Pro-*

ceedings of the Southern Association for Information Systems Conference, pp. 141–147. Atlanta, 2013. 1

[40] Neo4j, Inc. Neo4j graph database & analytics. https://neo4j.com/. [Online; accessed 24-March-2025]. 2, 3, 6

[41] NetworkX Developers. NetworkX: Network analysis in python. https://networkx.org/, 2024. [Online; accessed: 28-March-2025]. 8

[42] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, 2009. doi: 10.1145/1567274.1567278 3

[43] R. Pienta, F. Hohman, A. Endert, A. Tamersoy, K. Roundy, C. Gates, S. Navathe, and D. H. Chau. Vigor: Interactive visual exploration of graph query results. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):215–225, 2017. doi: 10.1109/TVCG.2017.2744898 1, 2

[44] R. Pienta, A. Tamersoy, A. Endert, S. Navathe, H. Tong, and D. H. Chau. Visage: Interactive visual graph querying. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pp. 272–279, 2016. doi: 10.1145/2909132.2909246 1, 2, 5

[45] A. Pister, C. Prieur, and J.-D. Fekete. Visual queries on bipartite multivariate dynamic social networks. In *Proceedings of the 24th Eurographics Conference on Visualization*, 2022. doi: 10.2312/evp.20221115 2

[46] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, Jan. 2015. doi: 10.14778/2735479.2735493 2

[47] C. R. Rivero and H. M. Jamil. Efficient and scalable labeled subgraph matching using sgmatch. *Knowledge and Information Systems*, 51(1):61–87, 2017. doi: 10.1007/s10115-016-0968-2 2

[48] Rob Behnke. Explained: The EasyFi Hack April 2021. https://www.halborn.com/blog/post/explained-the-easyfi-hack-april-2021, 2021. [Online; accessed 19-March-2025]. 7

[49] M. A. Rodriguez. The gremlin graph traversal machine and language. In *Proceedings of the 15th Symposium on Database Programming Languages*, pp. 1–10. ACM, Kohala Coast, Aug. 2015. doi: 10.1145/2815072.2815073 3, 9

[50] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 12 pages, Aug. 2008. doi: 10.14778/1453856.1453899 2

[51] H. Song, Z. Dai, P. Xu, and L. Ren. Interactive visual pattern search on graph data via graph representation learning. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):335–345, 2021. doi: 10.1109/TVCG.2021.3114857 1, 2

[52] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2021. doi: 10.14778/3425879.3425888 2

[53] A. Tamersoy, E. Khalil, B. Xie, S. L. Lenkey, B. R. Routledge, D. H. Chau, and S. B. Navathe. Large-scale insider trading analysis: patterns and discoveries. *Social Network Analysis and Mining*, 4:1–17, 2014. doi: 10.1007/s13278-014-0201-9 1

[54] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968. doi: 10.1145/363347.363387 2

[55] J. Troidl, S. Warchol, J. Choi, J. Matelsky, N. Dhanyasi, X. Wang, B. Wester, D. Wei, J. W. Lichtman, and H. Pfister. Vimo – visual analysis of neuronal connectivity motifs. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):748–758, 2023. doi: 10.1109/TVCG.2023.3327388 1, 2, 5

[56] S. Van Den Elzen, D. Holten, J. Blaas, and J. J. Van Wijk. Dynamic network visualization with extended massive sequence views. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1087–1099, Nov. 2013. doi: 10.1109/TVCG.2013.263 5

[57] H. Vargas, C. Buil-Aranda, A. Hogan, and C. López. A user interface for exploring and querying knowledge graphs (extended abstract). In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, article no. 666, 5 pages, 2021. doi: doi/abs/10.5555/3491440.3492106 2

[58] T. von Landesberger, M. Gorner, and T. Schreck. Visual analysis of graphs with multiple connected components. In *Proceedings of the Symposium on Visual Analytics Science and Technology*, pp. 155–162. IEEE, Oct. 2009. doi: 10.1109/VAST.2009.5333893 3

[59] X. Wen, Y. Wang, X. Yue, F. Zhu, and M. Zhu. NFTDisk: Visual detection of wash trading in nft markets. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–15. ACM, 2023. doi: 10.1145/3544548.3581466 1

[60] Wikipedia contributors. *Les Misérables* — wikipedia. https://en.wikipedia.org/w/index.php?title=Les_Mis%C3%A9rables&oldid=1006098025, Mar. 2025. [Online; accessed 19-March-2025]. 7

[61] J. Wu, D. Lin, Q. Fu, S. Yang, T. Chen, Z. Zheng, and B. Song. Toward understanding asset flows in crypto money laundering through the lenses of Ethereum heists. *IEEE Transactions on Information Forensics and Security*, 19:1994–2009, 2023. doi: 10.1109/TIFS.2023.3346276 7

[62] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment*, 7(7):565–576, 2014. doi: 10.14778/2732286.2732293 1

[63] Y. Ye, X. Lian, and M. Chen. Efficient exact subgraph matching via gnn-based path dominance embedding. *Proceedings of the VLDB Endowment*, 17(7):1628–1641, Mar. 2024. doi: 10.14778/3654621.3654630 2

[64] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. Autog: A visual query autocompletion framework for graph databases. *The VLDB Journal*, 26(3):347–372, Jan. 2017. doi: 10.1007/s00778-017-0454-9 9

[65] 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*, pp. 192–203. ACM, Mar. 2009. doi: 10.1145/1516360.1516384 2

[66] Y. Zhao, S. Lv, W. Long, Y. Fan, J. Yuan, H. Jiang, and F. Zhou. Malicious webshell family dataset for webshell multi-classification research. *Visual Informatics*, 8(1):47–55, Mar. 2024. doi: 10.1016/j.visinf.2023.06.008 7

[67] H. Zhou, P. Lai, Z. Sun, X. Chen, Y. Chen, H. Wu, and Y. Wang. AdaMotif: Graph simplification via adaptive motif design. *IEEE Transactions on Visualization and Computer Graphics*, 31(1):688–698, 11 pages, Sept. 2024. doi: 10.1109/TVCG.2024.3456321 3