Towards Plug-and-Play Visual Graph Query Interfaces: Data-driven Canned Pattern Selection for Large Networks

[Technical Report]

Zifeng Yuan §,‡ Huey Eng Chua § Sourav S Bhowmick ‡ Zekun Ye §,‡ Wook-Shin Han♠
Byron Choi ‡

‡School of Computer Science and Engineering, Nanyang Technological University, Singapore
‡School of Computer Science, Fudan University, China
*POSTECH, South Korea

hechua@assourav@ntu.edu.sg, wshan@dblab.postech.ac.kr, bchoi@comp.hkbu.edu.hk, zfyuan16/zkye16@fudan.edu.cn

ABSTRACT

Canned patterns (i.e., small subgraph patterns) in visual graph query interfaces (a.k.a GUI) facilitate efficient query formulation by enabling pattern-at-a-time construction mode. However, existing GUIs for querying large networks either do not expose any canned patterns or if they do then they are typically selected manually based on domain knowledge. Unfortunately, manual generation of canned patterns is not only labor intensive but may also lack diversity for supporting efficient visual formulation of a wide range of sub-graph queries. In this paper, we present a novel generic and extensible framework called TATTOO that takes a data-driven approach to automatically selecting canned patterns for a GUI from large networks. Specifically, it first decomposes the underlying network into truss-infested and truss-oblivious regions. Then candidate canned patterns capturing different real-world query topologies are generated from these regions. Canned patterns based on a user-specified plug are then selected for the GUI from these candidates by maximizing coverage and diversity, and by minimizing the cognitive load of the pattern set. Experimental studies with real-world datasets demonstrate the benefits of TATTOO. Importantly, this work takes a concrete step towards realizing plug-and-play visual graph query interfaces for large networks.

1 INTRODUCTION

A recent survey [39] revealed that graph query languages and usability are considered as some of the top challenges for graph processing. A common starting point for addressing these challenges is the deployment of a visual query interface (a.k.a GUI) that can enable an end user to draw a graph query interactively by utilizing direct-manipulation [41] and visualize the result matches effectively [1, 36]. A useful component of such a GUI is a panel containing a set of canned patterns (i.e., small subgraphs) which is beneficial to visual querying in at least three possible ways [9, 26, 27]. First, it can potentially decrease the time taken to visually construct a query by facilitating pattern-at-a-time query mode (i.e., construct multiple nodes and edges by performing a single click-and-drag action) in lieu of edge-at-a-time mode. Second, it can facilitate "bottom-up" search when a user does not have upfront knowledge of what to search for. Third, canned patterns (patterns for brevity) may alleviate user frustration of repeated edge construction especially for larger queries.

Example 1.1. Consider the real-world subgraph query in Figure 1 from BSBM [2] (Query Q12). Suppose Wei, a non-programmer, wishes to formulate it using a GUI containing a set of canned patterns (a subset of patterns is shown). Specifically, he may drag and drop P2 and P3 on the Query Canvas, merge the yellow vertex of P3 with the center vertex of P2, add a vertex and connect it with the grey vertex of P2. Finally, Wei can assign appropriate vertex labels. Observe that it requires five steps to construct the topology. On the other hand, if Wei takes an edge-at-a-time approach to construct the query, it would require 23 steps. Clearly, canned patterns enable more efficient (i.e., fewer number of steps or lesser time) formulation of the query.

It is worth noting that Wei may not necessarily have the complete query structure “in his head” during query formulation. He may find P3 interesting while browsing the pattern set, which may initiate his bottom-up search leading to the query. Clearly, without the existence of a pattern set, such bottom-up search would be infeasible in practice.

Data-driven selection of relevant canned patterns for a GUI (e.g., P1, P2, P3 in Fig. 1) is important to facilitate efficient query formulation [9, 26]. In particular, data-driven selection paves the way...
for plug-and-play visual graph query interfaces, which are like a plug-and-play device that can be plugged into any kind of socket (i.e., graph data) and used. A plug-and-play GUI is dynamically built from a high-level specification of canned pattern properties known as the plug (detailed in Section 3). Specifically, given a network \( G \) and a plug \( h \), the GUI is automatically constructed by populating its various components (e.g., node/edge attributes, canned patterns) from \( G \) without the need for manual GUI coding. This enhances portability and maintainability of GUIs across different data sources [9].

In this paper, we present a novel framework called TATTOO (daTa-driven CAnnOmed PRattErn selecTIOn from netwOrks) that takes a data-driven approach to the canned pattern selection (CPS) problem for large networks. Given a network \( G \), a user-specified plug specification \( h \) which is the number of canned patterns to display and their minimum and maximum permissible sizes, TATTOO automatically selects canned patterns from \( G \) that satisfy \( h \).

The CPS problem is technically challenging. First, it is a NP-hard problem [26]. Second, the availability of query logs can facilitate the selection of relevant patterns as they provide rich information of past queries. In practice, however, such information is often publicly unavailable (e.g., none of the networks in SNAP [5] reveal query logs) due to privacy and legal reasons. Hence, we cannot realistically assume the availability of query logs to select patterns. Furthermore, users may demand a gui prior to querying a network. Hence, there may not exist any query log prior to the creation of a visual query interface. Third, it is paramount to find unlabeled patterns (e.g., Example 1.1) that are potentially useful for query formulation (detailed in Sec. 4). However, the selection of such patterns is challenging as there is an exponential number of them in a large network. Fourth, these selected patterns should not only be topologically diverse so that they are useful for a wide variety of queries but they should also impose low cognitive load (i.e., mental load to visually interpret a pattern’s edge relationships to determine if it is useful for a query) on users. In particular, large graphs overload the human perception and cognitive systems, resulting in poor performance of tasks such as identifying edge relationships [25, 46].

At this point, a keen reader may wonder why building blocks of real-world networks (e.g., paths of length 6, triangle patterns) [33, 45] cannot be simply utilized as canned patterns since they have high coverage and low cognitive load. However, it may take a larger number of steps to formulate a variety of queries using these patterns due to their small size. For instance, reconsider Example 1.1. Suppose the pattern set consists of an edge, a path of length 2 (i.e., 2-path), a triangle, and a rectangle. In this case, \( Q_{12} \) may be formulated by dragging and dropping the rectangle once, the 2-path three times, construction of a single node and two edges, along with three node mergers. That is, it takes 10 steps altogether, which is more than using the patterns in Figure 1. Furthermore, these patterns do not expose “interesting” substructures to facilitate bottom-up search as they occur in almost all large real-world networks.

TATTOO addresses the aforementioned challenges as follows. It exploits a recent analysis of real-world query logs [12] to classify topologies of canned patterns into categories that are consistent with the topologies of real-world queries (detailed in Section 5). This enables us to reach a middle ground where TATTOO does not need to be restricted by the availability of query logs but yet exploit topological characteristics of real-world queries to guide the selection process. Next, it realizes a novel and efficient candidate canned pattern generation technique based on the classified topologies to identify potentially useful patterns. Lastly, canned patterns are selected from these candidates for display on the gui based on a novel pattern set score that is sensitive to coverage, diversity, and cognitive load of patterns. Specifically, we leverage recent progress in the algorithm community to propose a selection algorithm that guarantees \( \frac{1}{2} \)-approximation [13]. Figure 2 depicts an overview of the TATTOO framework. Experiments with several real-world large networks and users reveal that TATTOO can select canned patterns within few minutes. Importantly, these patterns can reduce the number of steps taken to formulate a subgraph query and query formulation time by up to 9.7X and 18X, respectively, compared to several baseline strategies.

In summary, this paper makes the following contributions: (1) We describe TATTOO, an end-to-end canned pattern selection framework for any plug-and-play visual graph query interface for large networks independent of domains and data sources. A video of a plug-and-play interface that incorporates TATTOO can be viewed at https://youtu.be/sL0yHV1eEPw. (2) We formally introduce the CPS problem for large networks (Sec. 4) and present a novel categorization of potentially useful canned patterns in Section 5. (3) We present an efficient solution to select canned patterns for a GUI (Sec. 6 - 7). Specifically, we present a novel candidate pattern generation framework that is grounded on topologies of real-world subgraph queries. Furthermore, for the first time in graph querying literature, we utilize the recent technique in [13] from the algorithm community to select canned patterns with good theoretical quality guarantees. (4) Using real-world networks, we show the superiority of our proposed framework compared to several baselines (Sec. 8).

Proofs of theorems and lemmas are provided in Appendix A.

2 RELATED WORK

Most germane to our work is our prior efforts on data-driven construction of visual graph query interfaces in [10, 26, 48]. The work in [27] focuses on the maintenance of canned patterns for evolving data graphs. Our work differs from these efforts in the following ways. First, we focus on selecting unlabeled canned patterns from large networks in contrast to labelled patterns from a collection of small- or medium-sized data graphs in [10, 26, 27, 48]. Specifically, existing efforts such as CataPult [26] first partitions a collection of data graphs into a set of clusters and summarizes each cluster.
to a cluster summary graph (CSG). Then, it selects the canned patterns with the aforementioned characteristics from these CSGs using a weighted random walk approach. This clustering-based approach is prohibitively expensive for large networks as detailed in Sec. 8. Second, these approaches do not exploit characteristics of real-world subgraph queries for selecting canned patterns. In contrast, we utilize topological characteristics of real-world queries to guide our solution design. Third, we present a novel real-world query topology-aware candidate pattern generation technique and a selection technique that provides quality guarantee. No theoretical guarantee is provided in [10, 26, 48] for selecting canned patterns. Lastly, as detailed in Sec. 7, the computation of pattern score to assess the quality of canned patterns is different as the computation of cognitive load and diversity is different from [26] due to the nature of large networks. Furthermore, in this work we provide a theoretical analysis of the pattern score.

Motif discovery techniques [21, 33] do not consider diversity and cognitive load. Sizes of these motifs are generally bounded in the range of [3-7] in real applications [21, 33]. For the same reason, it is difficult to use graphlets [7, 23, 38] as patterns. Also, frequent subgraphs [17] may not constitute good canned patterns [9] and are prohibitively expensive to compute for large networks (detailed in Sec. 8).

3 BACKGROUND

We first introduce several graph terminologies that we shall be using subsequently. Next, we formally define the notion of plugs. Finally, we briefly describe the desirable characteristics of canned patterns as introduced in [26].

3.1 Terminology

We denote a graph or network as $G = (V,E)$, where $V$ is a set of nodes/vertices and $E \subseteq V \times V$ is a set of edges. Vertices and edges can have labels as attributes. The size of $G$ is defined as $|G| = |E|$. The degree of a vertex $v \in V$ is denoted as $\text{deg}(v)$. In this paper, we assume that $G$ is an undirected, unweighted graph with labeled vertices.

A triangle is a cycle of length 3 in $G$. The support of an edge $e = (u,v) \in E$ (denoted by $\text{sup}(e)$) is the number of triangles in $G$ containing $u$ and $v$. $G_S = (V_S, E_S)$ is a subgraph of $G$ (denoted by $G_S \subseteq G$) if $V_S \subseteq V$ and $E_S \subseteq E$. Consider another graph $G' = (V', E')$ where $|V| = |V'|$. $G$ and $G'$ are isomorphic if there exists a bijection $f : V \rightarrow V'$ such that $(u,v) \in E$ iff $(f(u), f(v)) \in E'$. Further, there exists a subgraph isomorphism from $G$ to a graph $Q$ if $G$ contains a subgraph $G_S$ that is isomorphic to $Q$. We refer to $G_S$ as the embedding of $Q$ in $G$.

Given $G$, the $k$-truss of $G$ is the largest subgraph $G' = (V', E')$ of $G$ in which every edge $e \in E'$ is contained in at least $k-2$ triangles within the subgraph. A $2$-truss is simply $G$ itself. We define the trussness of an edge $e$ as $t(e) = \max\{|e| \in E_T\}$ where $T_k = (V_T, E_T)$ is the $k$-truss in $G$. Further, $k_{\text{max}}$ denotes the maximum trussness.

3.2 Plugs

Recall that data-driven selection of canned patterns facilitates the construction of a plug-and-play visual graph query interface. A plug is a high-level specification of the patterns in a GUI. Given the specification, TATTOO dynamically generates the canned patterns satisfying it from the underlying network. Formally, it is defined as follows.

Definition 3.1. [Plug] Given a network $G$ and a GUI $I$, a plug $b = (\eta_{\text{min}}, \eta_{\text{max}}, \gamma)$ where $\eta_{\text{min}} > 2$ (resp. $\eta_{\text{max}}$) is the minimum (resp. maximum) size of a pattern, $\gamma > 0$ is the number of patterns to be displayed on $I$.

Essentially a plug is a collection of attribute-value pairs that specifies the high-level content of a canned pattern panel in a GUI. For example, $b = (3, 15, 30)$ is a plug. Accordingly, the minimum and maximum sizes of patterns in $I$ are 3 and 15, respectively, and the total number of patterns to be displayed is 30. Observe that there can be multiple plugs for $G$ as well. Similarly, the same plug can be used for different $G$. Hence, different guis can be constructed by different plug specifications.

A plug should possess the following properties. (a) Data independence - A plug should not depend upon a specific network (i.e., socket). The specification of plug enables this by not admitting any network-specific information. Observe that this property is important for plug-and-play interfaces as a plug can be used on different network data across different application domains. (b) Able to select canned patterns with the required specifications - The resulting canned pattern selection mechanism should select patterns exactly as specified by the plug.

3.3 Characteristics of Canned Patterns

Since it is impractical to display a large number of patterns in a visual graph query interface $I$, the number of patterns should be small and satisfy certain desirable characteristics as introduced in [26].

High coverage. A pattern $p \in \mathcal{P}$ covers $G$ if $G$ contains a subgraph $s$ that is isomorphic to $p$. Since $p$ may have many embeddings in $G$, the pattern set $\mathcal{P}$ should ideally cover as large portion of $G$ as possible. Then a large number of subgraph queries on $G$ can be constructed by utilizing $\mathcal{P}$.

High diversity. High coverage of patterns is insufficient to facilitate efficient visual query formulation [26]. In order to make efficient use of the limited display space on $I$, $\mathcal{P}$ should be structurally diverse to serve a variety of queries. This also facilitates bottom-up search where a user gets a bird’s-eye view of the diverse substructures in $G$.

Low cognitive load. Cognitive load refers to the memory demand or mental effort required to perform a given task [25]. A topologically complex pattern may demand substantial cognitive effort from an end user to decide if it can aid in her query formulation [26]. Hence, it is desirable for the canned patterns in $\mathcal{P}$ to impose low cognitive load on an end user to make browsing and selecting relevant patterns cognitively efficient during visual query formulation.

4 THE CPS PROBLEM

Given a data graph or network $G = (V,E)$, a visual graph query interface $I$ and a user-specified plug $b$, the goal of the canned pattern
The \textsc{CPS} problem is to select a set of unlabelled patterns $\mathcal{P}$ for display on $\mathcal{I}$, which satisfies the specifications in $b$ and optimizes coverage, diversity and cognitive load of $\mathcal{P}$.

Observe that our \textsc{CPS} problem differs from [26] in two key ways. First, we focus on a single large network instead of a large collection of small- or medium-sized data graphs. Second, we select unlabelled patterns instead of labelled ones. In large networks, a subgraph query may not always contain labels on its vertices or edges. Specifically, unlabelled query graphs are formulated in the subgraph enumeration problem [6] whereas query graphs are labelled in the subgraph matching problem [42]. Hence, by selecting unlabelled patterns \textsc{Tattoo} facilitates visual formulation of both these categories of queries. In particular, one may simply drag-and-drop specific vertex/edge labels from the \textit{Attribute} panel of a \textsc{gui} to add labels to the vertices/edges of a pattern (e.g., Example 1).

We now formally define the \textsc{CPS} problem addressed in this paper. We begin by introducing coverage, diversity, and cognitive load of canned patterns. Let $S(p) = \{s_1, \ldots, s_n\}$ be a bag of subgraphs in $G$ isomorphic to $p$ (i.e., embeddings of $p$) where vertex labels in $G = (V, E)$ and $p = (V_p, E_p)$ are assumed to be the same and $s_i = (V_i, E_i)$. We say an edge $e \in E_i$ is covered by $p$. The coverage of $p$ is given as $\text{cov}(p) = \left| \bigcup_{i \in S(p)} E_i \right| / |E|$. Similarly, $\text{cov}(\mathcal{P}) = \left| \bigcup_{p \in \mathcal{P}} E_p \right| / |E|$ (i.e., $f_{\text{cov}}(\mathcal{P})$) where every $e \in E^\prime$ is covered by at least one $p \in \mathcal{P}$. Since $|E|$ is constant for a given $G$, coverage can be rewritten as $\text{cov}(p) = \left| \bigcup_{i \in S(p)} E_i \right|$ and $\text{cov}(\mathcal{P}) = \left| E^\prime \right|$. The diversity of $p$ w.r.t to $\mathcal{P}$ is the inverse of similarity of $p$. In particular, the similarity of a set of canned patterns $\mathcal{P}$ is denoted as $f_{\text{sim}}(\mathcal{P}) = \sum_{(p_i, p_j) \in \mathcal{P} \times \mathcal{P}} \text{sim}(p_i, p_j)$ where $\text{sim}(p_i, p_j)$ is the similarity between patterns $p_i$ and $p_j$ (detailed in Sec. 7). Finally, we measure cognitive load of $p$ (denoted by $\text{cog}(p)$) based on the size, density, and edge crossings in $p$ (detailed in Sec. 7) as a user tends to spend more time identifying relationships between vertices in denser graphs with more edge crossings [24, 25, 46]. The cognitive load of $\mathcal{P}$ (i.e., $f_{\text{cog}}(\mathcal{P})$) is given as $\sum_{p \in \mathcal{P}} \text{cog}(p)$. Definition 4.1. [\textsc{CPS} Problem] Given a network $G$, a \textsc{gui} $\mathcal{I}$, and a plug $b = (\eta_{\text{min}}, \eta_{\text{max}}, \gamma)$, the goal of the \textit{canned pattern selection (CPS) problem} is to find a set of unlabelled canned patterns $\mathcal{P}$ from $G$ that satisfies

$$\max_{\mathcal{P}} f_{\text{cov}}(\mathcal{P}), -f_{\text{sim}}(\mathcal{P}), -f_{\text{cog}}(\mathcal{P})$$

subject to $|\mathcal{P}| = \gamma, \mathcal{P} \in \mathcal{U}$ (1)

where $\mathcal{P}$ is the solution; $\mathcal{U}$ is the feasible set of canned pattern sets in $G$; $f_{\text{cov}}(\mathcal{P}), f_{\text{sim}}(\mathcal{P})$ and $f_{\text{cog}}(\mathcal{P})$ are the coverage, similarity, and cognitive load of $\mathcal{P}$, respectively.

Remark. Observe that $\text{cns}$ is a multi-objective optimization problem as our goal is to maximize coverage and diversity (i.e., minimize similarity) of canned patterns while minimizing their cognitive load. Hence, we address it by converting $\text{cns}$ into a single-objective optimization problem using a \textit{pattern score} (detailed in Section 7). Also, observe that we aim to find patterns of size greater than 2 (i.e., $\eta_{\text{min}} > 2$). Small-size patterns that are basic building blocks of networks [33, 45] (e.g., edge, 2-path, triangle) are provided by default for all datasets (i.e., default patterns).

The $\text{cns}$ problem is shown to be NP-hard in [26] by reducing it from the classical maximum coverage problem.

Theorem 4.2. The $\text{cns}$ problem is NP-hard.

5 CATEGORIES OF CANNED PATTERNS

In theory, numerous different patterns can be selected from a given network. Which of these are “useful” for subgraph query formulation in practice? In this section, we provide an answer to this question.

5.1 Topologies of Real-world Queries

Although basic building blocks of networks [33, 45] are presented as default patterns in our \textsc{gui}, as remarked earlier, they are insufficient as they do not expose to a user more domain-specific and larger patterns in the underlying data. Such larger substructures not only facilitate more efficient construction of subgraph queries but also guide users for bottom-up search by exposing substructures that are network-specific. However, which topologies of these substructures should be considered for canned patterns? Ideally, real-world subgraph query logs can provide guidance to resolve this challenge. However, as remarked in Section 1, such data may be unavailable. Hence, we leverage results from a recent study [12] that analysed a large volume of real-world \textsc{sparql} query logs. It revealed that topologies of many real-world subgraph queries map to chains, trees, stars, cycles, petals, and flowers$^2$ [12]. Figure 3 depicts examples of these topologies in real-world subgraph queries extracted from BigRDFBench [40], BSBM [2], Rapid [4], and DBpedia [18]. Consequently, canned patterns in any \textsc{gui} should facilitate efficient construction of these substructures.

5.2 Topologies of Canned Patterns

We consider the following types of topological structures of canned patterns in order to facilitate construction of the above query substructures.

Path and cycle patterns. A subgraph query may contain paths of different lengths (i.e., chain) and/or cycles. Figure 3 depicts some examples. Hence, our canned patterns should expose representative $k$-paths and $k$-cycles in the underlying data. Given a graph $G = (V, E)$, a $k$-path, denoted as $p_k = (V_k, E_k)$, is a walk of length $k$ containing a sequence of vertices $v_1, v_2, \ldots, v_k$ where $E_k \subseteq E$, $V_k \subseteq V$ such that all vertices in $V_k$ are distinct. A $k$-cycle is simply a closed $(k - 1)$-path where $k \geq 3$.

$^2$A petal is a graph consisting of a source node $s$, target node $t$ and a set of at least 2 node-disjoint paths from $s$ to $t$. A flower is a graph consisting of a node $x$ with three types of attachments: chains (stamens), trees that are not chains (the stems); and petals. A flower set is a graph in which every connected component is a flower.
Table 1: No. of steps for constructing queries.

| ID  | Edge-at-a-time | Default patterns | Canned patterns |
|-----|----------------|------------------|-----------------|
| (6) | 17             | 6 \{2-path + 1 square - 1 edge + 1 edge + 1 merge\} | 5 \{4-path + 2 2-path + 2 merge\} |
| (7) | 25             | 1 \{5-path + 1 node + 2 edge + 3 merge\} | 3 \{5-star + 2-path + 1 node + 2 edge\} |
| (8) | 18             | 8 \{2-path + 1 edge + 3 merge\} | 4 \{5-cycle + 4-star + 1 merge\} |
| (9) | 23             | 10 \{square + 3 2-path + 1 node + 2 edge + 3 merge\} | 3 \{4-path + 3-star + 1 node + 1 edge + 1 merge\} |

and $E_{ck} = \{(u, v)\} \cup V'_{ck}$ where $V'_{ck} = \{(w_i : 0 \leq i \leq k - 2) \}$ and $E'_{ck} = \{(u, w_i), (w_i, v) : 0 \leq i \leq k - 2\}$. $k$-CP can be considered as a building block of $k$-trusses since it is found with respect to each edge in a given $k$-truss. Examples of $k$-CPs ($4$-CP and $5$-CP) are illustrated in Figure 4. We refer to the edge in a $k$-chord pattern that is involved in $(k-2)$-trangles as a truss edge and the remaining edges as non-truss edges. For example, in Figure 4, edges $(A_1, B_1)$ and $(A_2, B_2)$ are truss edges whereas $(A_1, C_1)$ and $(B_2, D_2)$ are non-truss edges. Correspondingly, vertices of a truss edge (e.g., $A_1, B_1, A_2, B_2$) are referred to as truss vertices. Observe that we can formulate a simple petal query in two steps by selecting the $4$-CP pattern and deleting the truss edge.

To select larger canned patterns with greater structural diversity, we combine $k$-CPs to yield additional composite chord patterns ( CCP ) that occur in the underlying network. Observe that combining a set of $k$-CPs in different ways results in different patterns as demonstrated in Figure 4. However, this is an overkill as they are not only expensive to compute but also may generate patterns with higher density (higher cognitive load) or are larger than $\eta_{max}$. Hence, we focus on the CCP generated by merging a single edge of two $k$-CPs as it not only reduces the complexity of CCP generation, but also produces CCPs with lower density.

Unique small graph patterns. Lastly, we find small connected subgraphs that do not fall under above categories but occur multiple times in the underlying network.

Table 1 reports the number of steps taken by various modes of query construction of selected query topologies in Fig. 3. Observe that query construction using canned patterns often takes fewer number of steps compared to construction using only default patterns, emphasizing the need for patterns beyond the default ones. One can also formulate a specific query following multiple alternatives, i.e., using multiple sets of patterns (canned and default). This gives users the flexibility to formulate a query using these patterns in many ways, all of which often take fewer steps compared to the edge-at-a-time or default pattern-based modes.

6 CANDIDATE PATTERNS GENERATION

In the preceding section, we classified the topologies of canned patterns broadly into “$k$-truss-like” and “non-$k$-truss-like” structures. In this section, we describe how candidate canned patterns conforming to these topological categories are extracted from the underlying network $G$.

We begin by providing an overview of the Tattoo algorithm. Algorithm 1 outlines the procedure. It first decomposes $G$ into truss-infested and truss-oblivious regions (Line 1) and then generates “$k$-truss-like” and “non-$k$-truss-like” candidate patterns from these regions, respectively (Lines 2-8). Finally, it selects the canned pattern.
Algorithm 1 The Tattoo algorithm.

Require: Data graph $G$, plug $b = (\eta_{\text{min}}, \eta_{\text{max}}, \lambda)$.
Ensure: Canned pattern set $P^*.$

1. $G_T$, $G_O \leftarrow \text{GraphDecomposition}(G)$
2. $P_{cep}, \text{freq}(P_{cep}) \leftarrow \text{GenChordPatterns}(G_T, t(e)) / \text{Alg. 2/}$
3. $P_{cep}, \text{freq}(P_{cep}) \leftarrow \text{GenComChordPatterns}(G_T, t(e)) / \text{Alg. 3/}$
4. $P_T, \text{freq}(P_T) \leftarrow \text{GenStarPatterns}(G_O) / \text{Alg. 4/}$
5. $G_T \leftarrow \text{RemovesAttPattEdges}(G_T, P_T)$
6. $P_T, \text{freq}(P_T) \leftarrow \text{GenSmallPatterns}(G_T, b) / \text{Alg. 5/}$
7. $P_{canal} \leftarrow P_T \cup \text{freq}(P_T) \cup P_T$
8. $\text{freq}(P_{canal}) \leftarrow \text{freq}(P_T) \cup \text{freq}(P_{canal})$
9. $P^* \leftarrow \text{SelectCannedPatterns}(P_{canal}, \text{freq}(P_{canal}), b)$

Table 2: TIR and TOR graphs in real networks.

| Data | Name | $|V|$ | $|E|$ | $\%(|E|)$ | $\%(O(|E|))$
|------|------|------|------|----------|----------|
| RK   | Inc-NeighborNet | 28K  | 218K | 67.2 | 32.7 |
| GO   | Inc-Gowalla | 19K | 99B | 78.2 | 21.8 |
| DM   | com-DG | 17K | 170M | 93 | 7 |
| AM   | com-Amazon | 31K | 92B | 77.2 | 22.8 |
| RJ   | RoadNet-PA | 115M | 134M | 12.7 | 57.3 |
| RJ   | com-Twbible | 115M | 299M | 46.8 | 22.2 |
| RJ   | RoadNet-VK | 175M | 172M | 12.5 | 57.5 |
| SK   | as-Attiler | 17M | 13M | 78.1 | 20.9 |
| RC   | RoadNet-CA | 175M | 277M | 12.6 | 57.4 |
| RJ   | com-LiveJournal | 851 | 314M | 83.2 | 16.9 |

set from these candidate patterns based on the plug specification (Line 9). We discuss the decomposition of $G$ and candidate pattern generation in turn. In the next section, we shall elaborate on the selection of canned patterns from the candidate patterns.

6.1 Truss-based Graph Decomposition

In order to extract “non-k-truss-like” and “k-truss-like” structures as candidate patterns, we first decompose a network $G$ into sparse (containing non-trusses) and dense (containing trusses) regions. The latter region is referred to as truss-infested region (TIR) graph and the former truss-oblivious region (TOR) graph, and are denoted by $G_T$ and $G_O$, respectively. Table 2 reports the sizes of $G_T$ and $G_O$ in several real-world networks measured as the percentage of the total number of edges. We observe $G_T$ basically consists of relatively large connected subgraphs that comprise multiple $k$-trusses. On the other hand, $G_O$ mainly consists of chains (i.e., paths), stars, cycles, and small connected components. Furthermore, although some networks have small $G_O$ (e.g., com-DBLP), there are networks where $G_O$ is large (e.g., RoadNet-CA), encompassing up to 87.5% of the total number of edges. Consequently, by decomposing a network into $G_T$ and $G_O$, we can improve efficiency by limiting the search for $k$-truss-like patterns in $G_T$ instead of the entire network and extract non-truss-like patterns from $G_O$. Additionally, generating candidate patterns of aforementioned topological categories from both TIR and TOR graphs enables us to select a holistic collection of patterns having higher coverage and diversity. Cognitive load of the pattern set is often reduced when patterns from both regions are considered due to the sparse structure of TOR.

Tattoo utilizes the state-of-the-art truss decomposition approach in [44] to decompose $G$ into $G_T$ and $G_O$. Briefly, this approach identifies $k$-trusses ($k \in [2, \eta_{\text{max}}]$) in $G$ iteratively by removing edges with support less than $k - 2$ from $G$. Hence, our graph decomposition algorithm adapts it to assign 2-trusses as $G_O$ and the remaining $k$-trusses as $G_T$.

We keep track of the edge trussness (denoted as $t(e)$) in $G_T$. Since the goal is to select canned patterns with maximum size

Figure 5: $k$-CCP node and edge neighbourhoods.

$\eta_{\text{max}}$: the upper bound of edge trussness is set to this value. The algorithm first identifies the support of each edge. Then, regions of the data graph are iteratively extracted by removing edges with the lowest support, starting from the sparsest (i.e., $sup(e) = 0$) to the densest. In particular, Tattoo considers all edges with $sup(e) = 0$ as sparse regions and these edges form the TOR graph $G_O$. The remaining edges form the TIR graph $G_T$.

In summary, the above approach makes the following two simple modifications to the truss decomposition technique in [44]: (1) instead of storing each $k$-truss as a separate graph, it stores 2-truss as $G_O$ and the remaining $k$-trusses are combined as a single graph $G_T$; (2) it assigns a trussness value $t(e)$ to every edge in $G_T$ and $G_O$. The worst-case time and space complexities of this algorithm are $O(|E|^{1.5})$ and $O(|V| + |E|)$, respectively [44].

6.2 Patterns from a TIR Graph

Next, we generate $k$-cpts and CCPs as candidate patterns from a TIR graph. For each pattern we also compute its frequency as it will be used subsequently to measure its coverage. We discuss them in turn.

Generation of $k$-chord patterns. Algorithm 2 describes generation of the $k$-cpts. In particular, we can find $k$-cpts with respect to each edge in a given $k$-truss. For instance, every edge in a 4-truss and a 5-truss is part of at least 2 and 3 triangles, respectively. Observe that the 2-chord pattern of an edge $e$ is simply the edge itself. Hence, Tattoo generates $k$-cpts for $k \geq 3$. The frequency of a $k$-cpt is measured by the frequency of the pattern occurring in $G_T$, which is essentially the number of edges having trussness greater than or equals to $k$ (Lines 13 to 20). Formally, given a TIR graph $G_T = (V_T, E_T)$ and a $k$-chord pattern $C_k = (V_{ck}, E_{ck})$, the frequency of $C_k$ is defined as $freq(C_k) = |\{e \in E_T | t(e) \geq k\}|$. Then, the set of $k$-cpts of a $G_T$ is simply the set of patterns $C_k$ whose frequency is greater than 0. We first generate $k$-chord patterns in $G_T$ and then compute their frequencies using edge trussness.

Lemma 6.1. The worst-case time and space complexities of $k$-cpt generation are $O(k_{\max}|E_T|^{1.5})$ and $O(|V_T| + |E_T|)$, respectively.

Generation of composite chord patterns. Next, we generate the CCPs. Specifically, we generate the following categories of CCPs based on different ways of merging truss and non-truss edges.

Definition 6.2. Let $C_{ck} = (V_{ck}, E_{ck})$ and $C_{ck} = (V_{ck}, E_{ck})$ be two $k$-chord patterns where $s, t \in V_{ck}$ and $u, v \in V_{ck}$ are truss vertices. Then, we can generate the following categories of composite chord patterns of $C_{ck}$ and $C_{ck}$ by merging $C_{ck}$ and $C_{ck}$ as follows:

1. $CCP_{\text{tr}}(k_1, k_2)$: merge the truss edge of $C_{ck}$ with a non-truss edge of $C_{ck}$.
2. $CCP_{\text{tr}}(k_1, k_2)$: merge the truss edge of $C_{ck}$ with a non-truss edge of $C_{ck}$.
3. $CCP_{\text{ntr}}(k_1, k_2)$: merge a non-truss edge of $C_{ck}$ with a non-truss edge of $C_{ck}$ such that there is an overlapping truss vertex.
4. $CCP_{\text{mtr}}(k_1, k_2)$: merge a non-truss edge of $C_{ck}$ with a non-truss edge of $C_{ck}$ such that there is no overlapping truss vertex.
Figure 4 depicts examples of these four categories of CCPs. When the context is clear, we shall simply refer to a CCP as CCP. A keen reader may observe that it is possible to create another CCP by merging the truss edge of $C_k$ with the truss edge of $C_{k'}$. However, this CCP is in fact a $k$-CP where $k = k_1 + k_2 - 2$. For instance, when $C_4$ and $C_5$ in Figure 4 are merged on their truss edges, the resultant pattern is a 7-CP. Also, combining two 3-CPs always yields a 4-CP (Lemma 6.3). Since 3-CPs have already been handled earlier, these combinations are ignored.

**Lemma 6.3.** Two 3-CPs always yield a CCP that is 4-CP.

We now elaborate on how the CCPs and their frequencies are computed in Tattoo. We shall introduce two terminologies related to node and edge neighbourhoods of a CCP to facilitate exposition. Given an edge $e = (u, v)$ in a $k$-truss, the $k'$-CCP node neighbourhood (denoted as $NB_{CCP}(k', e)$) of $e$ is a set of vertices $W$ adjacent to $u$ and $v$ such that $\forall w \in W, t((u, w)) \geq k'$ and $t((w, v)) \geq k'$ where $k' \leq k$. The $k'$-CCP edge neighbourhood (denoted as $EB_{CCP}(k', e)$) of $e$ is the set of edges $S$ adjacent to $e$ such that $S((u, x_1), (x_2, v)) \in S$, $x_1, x_2 \in NB_{CCP}(k', e)$ where $k' \leq k$. Figure 5 illustrates examples of $k'$-CCP node and edge neighbourhoods. For instance, $NB_{CCP}(4, e)$ consists of $v_3$ since $t((v_1, v_3)) \geq 4$ and $t((v_2, v_3)) \geq 4$.

**Lemma 6.4.** Given a truss edge $e$, there is at least a $k$-chord pattern $C_k$ on $e$ if $|NB_{CCP}(k', e)| \geq (k - 2)$.

**Frequencies of CCP$_n(k_1, k_2)$ and CCP$_{pat}(k_1, k_2).** Consider two different $k$-CPs. CCP$_n$ and CCP$_{pat}$ involve merger of truss edges belonging to one $k$-CP with a non-truss edge belonging to another $k$-CP. Given two $k$-CPs $C_{k_1}$ and $C_{k_2}$, let edges $e_1$ and $e_2$ be the truss edges of $C_{k_1}$ and $C_{k_2}$, respectively. Intuitively, a pattern is a CCP$_n(k_1, k_2)$ if it contains an embedding of $C_{k_1}$ and of $C_{k_2}$ whereby there is an edge $e_{k_1}$ in the pattern that belongs to the two embeddings such that $e_{k_1}$ is a truss edge of $C_{k_1}$’s embedding and is a non-truss edge of $C_{k_2}$’s embedding, respectively. In other words, $C_{k_1}$ and $C_{k_2}$ can form a CCP (CCP$_n(k_1, k_2)$) by merging a truss edge $e_1$ from $C_{k_1}$ with a non-truss edge from $C_{k_2}$, if the following conditions are satisfied: (a) *Condition 1:* There is a $C_{k_1}$ pattern on $e_1$ containing $e_2$. (b) *Condition 2:* There is a $C_{k_2}$ pattern on $e_2$ where $e_2 \neq e_1$. Note that due to Lemma 6.4, *Condition 1* holds if $\left|NB_{CCP}(k_2, e_2) \setminus \{u, v\}\right| \geq (k_2 - 2)$ where $e_1 = (u, v)$. Further, if $\left|NB_{CCP}(k_1, e_1) \cup NB_{CCP}(k_2, e_2) \setminus \{u, v\}\right| \geq (k_1 - 2) + (k_2 - 2)$, then the pattern CCP$_{pat}(k_1, k_2)$ must exist. Hence, Tattoo checks the conditions iteratively on decreasing $k_2$ and skips checks for $k'_2 < k_2$ if the conditions are satisfied for $k_2$. The frequency of CCP$_{pat}(k_1, k_2)$ is simply the number of such $e_1$ edges. For CCP$_{pat}(k_1, k_2)$, the approach is the same by swapping $C_{k_1}$ with $C_{k_2}$.

**Frequencies of CCP$_n(k_1, k_2)$ and CCP$_{pat}(k_1, k_2).** Recall that (Def. 6.2) a single-edge merge can also involve the merger of two non-truss edges, each from a different $k$-CP. Each non-truss edge contains a truss vertex. There are two ways in which two non-truss edges can merge as shown in Figures 6(b) and (c). In the former (resp. latter), vertex pairs $(w_1, w_2)$ (resp. $(w_1, u_1)$) and $(u_2, w_2)$ (resp. $(w_1, u_2)$) are merged. Hence, a pattern is a CCP$_n$ if it contains at least one embedding of a structure shown in Figure 6(b) which we refer to as the skeleton structure of CCP$_n$ (denoted as $S_{nn}$). Hence, we can follow the $S_{nn}$ of a CCP$_n$ in a TAR graph to compute its occurrence and frequency. Specifically, a CCP$_n$ can be obtained if the following conditions are satisfied: (a) *Condition 1:* There is a $C_{k_2}$ pattern on its truss edge $e_1 = (u_1, u_2, v)$ which contains $e_2 = (u_1, u_2, w_1)$. (b) *Condition 2:* There is a $C_{k_2}$ pattern on its truss edge $e_3 = (w_1, w_2, x)$ which contains $e_2$.

Note that *Condition 1* holds if $\left|NB_{CCP}(k_1, e_1) \setminus \{u_1, w_1, w_2\}\right| \geq (k_1 - 3)$ (Lemma 6.4). Similarly, *Condition 2* holds if $\left|NB_{CCP}(k_2, e_2) \setminus \{u_2, w_1, w_2\}\right| \geq (k_2 - 3)$. Further, if $\left|NB_{CCP}(k_1, e_1) \cup NB_{CCP}(k_2, e_2) \setminus \{u_1, u_2, w_1, w_2\}\right| \geq (k_1 - 3) + (k_2 - 3)$, then the pattern CCP$_{pat}$ must exist. The frequency of a CCP$_n$ is simply the number of skeleton structures $S_{nn}$ in a TAR graph.

CCP$_{pat}$ is very similar to CCP$_n$ except that the truss vertices of the merged edges are not combined during the merger. Figure 6(c) illustrates the skeleton structure of a CCP$_{pat}$ (S$_{pat}$), which occurs in all CCP$_{pat}$. The frequency of a CCP$_{pat}$ is the number of skeleton structures $S_{pat}$.

Observe that $freq(CC_P_{pat}(k_1, k_2)) = freq(CC_P_{pat}(k_2, k_1))$ since $k_1$ and $k_2$ can be swapped. The same is true for CCP$_n$ and CCP$_{pat}$. Hence, when combining two $k$-CPs, we only consider the case when $k_1 \geq k_2$.

**Algorithm.** Putting the above strategies together (outlined in Algorithm 3), the CCPs are computed as follows. For each edge in $G_T$, compute the $k_1$-CCP node and edge neighbourhoods (Lines 4-5). Next, it computes the four types of CCPs (Lines 6-13) based on the aforementioned strategies. Note that the smallest CCP generated is a CCP(3,4) due to Lemma 6.3. Also, we only compute CCP$_n(k_1, k_2)$ instead of both CCP$_n(k_1, k_2)$ and CCP$_{pat}(k_1, k_2)$ as CCP$_{pat}(k_1, k_2)$ is covered when $k_2$ and $k_1$ are swapped.
Algorithm 3 GenCombinatorialPatterns.

Require: tin graph $G_T = (V_T, E_T)$, transmissiveness of all edges $T(e)$.
Ensure: Composite chord patterns $P = (CCP_m \cup CCP_n \cup CCP_t \cup CCP_e)$ and frequency $freq(P)$ where $CCP_m = [CCP_m(k_1, k_2) | 3 \leq k_1 \leq k_2 \leq \max(k_1, k_2)]$, $CCP_n = [CCP_n(k_1, k_2) | 3 \leq k_1 \leq k_2 \leq \max(k_1, k_2)]$, $CCP_t = [CCP_t(k_1, k_2) | 3 \leq k_1 \leq k_2 \leq \max(k_1, k_2)]$ and $CCP_e = [CCP_e(k_1, k_2) | 3 \leq k_1 \leq k_2 \leq \max(k_1, k_2)]$.

1. $CCP_m \rightarrow \phi$, $CCP_n \rightarrow \phi$, $CCP_t \rightarrow \phi$, $CCP_e \rightarrow \phi$.
2. for $e_i \in E_T$ do
3.  $k_1 \leftarrow i(e_i)$.
4.  Compute $N_{BC}(k_1, e_i)$ / compute k-ccp node neighbourhood.
5.  Compute $El_{cc}(k_1, e_i)$ / compute k-ccp edge neighbourhood.
6.  while $k_1 \geq k$/do
7.  for $e_j \in E_{BC}(k_1, e_i)$ do
8.   $k_2 \leftarrow \max(1, e_{j}, \max - k_1)$.
9.  $CCP_m, freq(CCP_m)$ / GetTN($G_T, e_1, k_1, k_2$) $\leftarrow$ GetNN($G_T, e_1, k_1, k_2$).
10. $CCP_n, freq(CCP_n)$ / GetNn($G_T, e_2, k_1, N_{BC}(k_1, e_2)$).
11. $CCP_t, freq(CCP_t)$ / GetNN($G_T, e_2, k_1, k_2$).
12. $CCP_e, freq(CCP_e)$ / GetNN($G_T, e_2, k_1, k_2$).
13. end while
14. end for.

Theorem 6.5. The worst-case time and space complexities of the CCP generation technique are $O(k_{\max}^2 |E_T|) + O(\max |E_T| + |V_T|)$, respectively.

6.3 Patterns from a TOR Graph

Generation of candidates from a TOR graph consists of two phases: star pattern extraction and small pattern extraction. The former extracts star and asterism patterns. Subsequently, the edges involved in these patterns are removed from $G_T$ resulting in further decomposition of the TOR graph. The resultant graph is referred to as the remainder graph ($G_R$). Then, the second phase extracts paths, cycles, and small connected subgraphs from $G_R$.

Extraction of star and asterism patterns. The frequencies of these patterns can be derived directly from their definitions (Sec. 5.2). Specifically, $freq(S_k) = \{v \in V_{O}, deg(v) = k\}$ and $freq(A_S) = \{v \in V_{O}, deg(v) = k\}$ where $v \in V_{O}, \{k, k_1\} \geq \epsilon, deg(r_1) = k_1$ and $deg(r_1) = k_1 + 1$. Algorithm 4 outlines the procedure. The star and asterism patterns are extracted in Lines 2 to 22 and Lines 6 to 20, respectively. Briefly, asterism patterns are found using breadth-first search (BFS). A vector of vertices is kept to track star centers in an asterism pattern. We “grow” the pattern by adding a neighbouring vertex $z$ of the current star center being considered only if $deg(z) \geq \epsilon$ and when the size of the grown pattern is less than or equals to $T_{max}$.

Lemma 6.6. The worst-case time and space complexities of star and asterism pattern extraction are $O(|V_{O}|^2)$ and $O(|E_{O}| + |V_{O}|)$, respectively.

Extraction of small patterns. The remainder graph $G_R$ is primarily composed of small connected components such as paths, cycles, and subgraphs with unique topology. Algorithm 5 outlines the extraction of these small patterns and we denote $k$-cycle as $V_k$ and subgraphs with unique topology as $U$. Given a graph $G = (V, E)$, a $k$-path, denoted as $P_k = (V_k, E_k)$, is a walk of length $k$ containing a sequence of vertices $v_1, v_2, \ldots, v_k, v_k, v_{k+1}$ where $E_k \subseteq E$, $V_k \subseteq V$ such that all vertices in $V_k$ are distinct. A $k$-cycle is simply a closed ($k - 1$)-path where $k \geq 3$. We refer to small subgraph patterns as connected components in $G_R$ that are neither $k$-paths nor $k$-cycles. Note that 1-path, 2-path, 3-cycle and 4-cycle are basic building blocks of real-world networks [33]. Recall that in Task-Too, we consider them as default patterns and they are not part of the candidate canned pattern set. Hence, we extract all $k$-paths for $k > 2$ (Lines 19-21) and $k$-cycles for $k > 4$ (Lines 22-24) and their frequencies. After that, small connected subgraphs and their corresponding frequencies are extracted.

Lemma 6.7. Worst-case time and space complexities to find small patterns are $O(\max |V_{R}|) + O(|E_{R}||V_{R}|)$, respectively.

Remark. Exponential time complexity of the small pattern extraction phase is due to the isomorphism check. The time cost is small in practice due to the small size of candidate patterns and their number is typically small in $G_R$.

7 Selection of Canned Patterns

In this section, we describe the algorithm to select canned pattern set $\mathcal{P}$ from the generated candidate patterns. We begin by presenting the theoretical underpinning that influences the design of our algorithm.

7.1 Theoretical Analysis

Due to the hardness of the CPS problem, we design an approximation algorithm to address it. We draw on insights from a related problem, team formation problem (TFP) [11, 14], which aims to hire a team of individuals $T$ from a group of experts $S$ for a specific project where $T \subseteq S$. Bhownik et al. [11] proposed that several aspects should be considered in TFP, namely, skill coverage (skill), social compatibility (social), teaming cost (team) and miscellaneous aspects such as redundant skills avoidance (red) and inclusion of selected experts (exp). The formulation of TFP is given as $s(T) = \alpha_{skil} \cdot \alpha_{skill}(T) - \alpha_{social} \cdot \alpha_{social}(T') + \alpha_{team} \cdot \alpha_{team}(T') - \alpha_{red} \cdot \alpha_{red}(T') + \alpha_{exp} \cdot \alpha_{exp}(T')$. These are skill, social, team, red and exp are non-negative coefficients that represent the relative importance of each aspect of team formation [11]. The goal is to find a team $T' \subseteq S$ where the non-negative and non-monotone function $s(T')$ is maximized. According to [11], this formulation can
Algorithm 5 GenSmallPatterns.

Require: Remainder graph $G_R = (V_R, E_R)$, pattern budget $b = (\eta_{min}, \eta_{max}, Y)$

Ensure: Small patterns $P_T = \{P \cup Y \cup U\}$ and frequency $freq(P_T)$

where $P = \{P_k | k \geq 3\}$, $Y = \{Y_k | k \geq 5\}$;

1: $P_T \leftarrow \phi$, $P \leftarrow \phi$, $Y \leftarrow \phi$, $U \leftarrow \phi$
2: $UC_{maxID} \leftarrow 0$ /maximum ID of UC {
3: for $v \in V_R$ do
4: set $v$ as unvisited
5: end for
6: for $v \in V_R$ do
7: if $v$ is not visited then
8: find component $C = (V_C, E_C)$ containing $v$
9: $n_{deg1} \leftarrow 0$ /num of nodes with deg=1 + /
10: $n_{deg2} \leftarrow 0$ /num of nodes with deg=2 + /
11: for $u \in V_C$ do
12: if $deg(u) = 1$ then
13: $n_{deg1} \leftarrow n_{deg1} + 1$
14: else if $deg(u) = 2$ then
15: $n_{deg2} \leftarrow n_{deg2} + 1$
16: end if
17: set $u$ as visited
18: end for
19: if $n_{deg1} = 2$ and $n_{deg2} = |V_C| - 2$ and $|V_C| \neq 2$ and $|V_C| \neq 3$ then
20: $P \leftarrow P \cup P_{\{C\}}$
21: $freq(P_{\{C\}}) \leftarrow freq(P_{\{C\}}) + 1$
22: else if $n_{deg1} = 0$ and $n_{deg2} = |V_C|$ and $|V_C| \neq 3$ and $|V_C| \neq 4$ then
23: $Y \leftarrow Y \cup Y_{\{C\}}$
24: $freq(Y_{\{C\}}) \leftarrow freq(Y_{\{C\}}) + 1$
25: else if $|E_C| \geq b_{\eta_{min}}$ and $|E_C| \leq b_{\eta_{max}}$ then
26: if isIsomorphic($C, U)$ is true then
27: $UC_{currID} \leftarrow GetID(C, U)$
28: $freq(U_{\{currID\}}) \leftarrow freq(U_{\{currID\}}) + 1$
29: else
30: $U \leftarrow U \cup (C, U_{\{maxID\}})$
31: $freq(U_{\{maxID\}}) \leftarrow 1$
32: $U_{\{maxID\}} \leftarrow U_{\{maxID\}} + 1$
33: end if
34: end if
35: end if
36: end for

be posed as an unconstrained submodular function maximization problem which is NP-hard for arbitrary submodular functions.

Selecting a set of canned patterns in CPS is akin to hiring a team of individuals in TFP where $f_{kill} = f_{red} = f_{team}$ correspond to $f_{cov}$, $f_{sim}$, and $f_{cog}$, respectively. Hence, CPS can be formulated in the form $s(P) = \alpha_{f_{cov}} f_{cov}(P') - \alpha_{f_{sim}} f_{sim}(P') - \alpha_{f_{cog}} f_{cog}(P')$ (Definition 7.1) where $P'$ is the set of candidate patterns which yields an optimized $s(P')$.

Definition 7.1. [Pattern Set Score] Given a pattern set $P'$, the score of $P'$ is $s(P') = \sum_{P \in P' \{P\}} (f_{cov}(P') - f_{sim}(P') - f_{cog}(P') + 2|P|)$ where $f_{cov}$, $f_{sim}$, and $f_{cog}$ are the coverage, similarity and cognitive load of $P'$, respectively.

Definition 7.2. [Good Candidate Pattern] Given a pattern set $P'$ and two candidate patterns $p_1$ and $p_2$, $p_1$ is considered a good candidate pattern if $s(P' \cup p_1) > s(P' \cup p_2)$ and is added to $P'$ instead of $p_2$.

Note that Definition 7.2 can be utilized for determining inclusion of a candidate pattern in $P'$. Next, we analyze the properties of $f_{cov}$, $f_{sim}$, and $f_{cog}$ and the pattern score.

Lemma 7.3. Coverage of a pattern set $P'$, $f_{cov}(P')$, is submodular.

Lemma 7.4. The similarity (resp. cognitive load) of a pattern set $P'$, $f_{sim}(P')$, (resp. $f_{cog}(P')$), is supermodular.

Theorem 7.5. The pattern set score $s(P')$ in Definition 7.1 is a non-negative and non-monotone submodular function.

Similar to $s(T')$ in TFP, $s(P')$ in CPS is non-negative and non-monotone. However, unlike TFP, CPS imposes a cardinality constraint where $|P'|$ is at most $\gamma$. Thus, CPS can be posed instead as a maximization of submodular function problem subject to cardinality constraint [13].

7.2 Coverage, Cognitive Load, and Similarity

Next, we quantify the coverage, cognitive load, and similarity measures used in the pattern score $s(P')$.

Coverage. Recall from Section 4, we can compute the coverage of a pattern $p$ as $cov_p = |U_i \in |S(p)| \in E_i|$. Since the edge sets of $G_T = (V_T, E_T)$ and $G_O = (V_O, E_O)$ are mutually exclusive, we further modify $cov_p$ to include a weight factor to account for effects exerted by the sizes of $G_T$ and $G_O$. Specifically, $cov_p = |U_i \in |S(p)| \in E_i| \frac{|G_k|}{|E|}$, where $G_k \in \{G_T, G_O\}$ for patterns obtained from $G_x$. However, exact computation of coverage for each candidate pattern is prohibitively expensive. Hence, we approximate $cov_p$ as follows: $cov_{ub}(p) = |E_p| \times freq(p) \times \frac{|G_k|}{|E|}$. Observe that $cov_{ub}(p)$ is in fact the upper bound of $cov_p$ when no isomorphic instances of $p$ in $G$ overlap. Any superior upper bound that can be computed efficiently can be incorporated. Unlike $cov_p$, computation of $cov_{ub}(p)$ requires only $freq(p)$, which is significantly more efficient.

The order of pattern extraction in $G_O$ (e.g., extracting stars and asterisms before small patterns) may affect the frequency of the extracted patterns. Hence, normalization of $cov_{ub}$ is performed for each class of patterns ($k$-CP, CCP, star, asterism, and small pattern) as follows:

$$cov_{ub}(p) = \frac{cov'_{ub}(p) - Min(cov'_{ub}(P_l)) + 1}{Max(cov'_{ub}(P_l)) - Min(cov'_{ub}(P_l)) + 1}$$

where $t \in \{k - CP, CCP, star, asterism, small\}$ represents a class of pattern. Specifically, we compute $k$-CPs and CCPs in $G_T$. Stars, asterisms and small patterns are computed in $G_O$. The normalized $cov_{ub}$ is in [0-1].

Cognitive Load. [26, 27] measure cognitive load based on size and density only, ignoring edge crossings. Since it is designed for a collection of small- or medium-sized data graphs, it is a reasonable measure as in many applications such data graphs have very few edge crossings (e.g., chemical compounds), if any. In contrast, edge crossings occur frequently in large networks and hence cannot be ignored in our context. In fact, Huang and colleagues examined the effect of edge crossings on mental load of users and found that cognitive load displays a relationship with edge crossings that resembles the logistic curve $\{24\} f(x) = \frac{1}{1 + e^{-k(x-x_0)}}$ where $L$ is the
curve’s maximum value, $x_0$ is the $x$ value of sigmoid’s midpoint and $k$ is the logistic growth rate [47].

**Lemma 7.6.** The crossing number (i.e., number of edge crossings) of any simple graph $G = (V, E)$ with at least 3 vertices satisfies $cr \geq |E| - 3|V| + 6$.

Hence cognitive load of a pattern $p$ is computed based on the size ($s_p = |E_p|$), density ($d_p = |E_p|/(V_p - 1)$) and edge crossing ($cr_p$), $cr_p = 0$ if $p$ is planar. Otherwise, it is $cr_p = |E_p| - 3|V_p| + 6$. We modelled the normalized cognitive load function in TATTOO according to the logistic curve:

$$\text{cog}_p = \frac{1}{1 + e^{-0.5(s_p + d_p + cr_p - 10)}}$$  (3)

Parameters of $\text{cog}_p$ are set empirically to ensure even distribution within the range of $[0, 1]$.

**Similarity.** Given a partial pattern set $\mathcal{P}'$ and two candidate patterns $p_1$ and $p_2$, TATTOO selects $p_1$ preferentially to add to $\mathcal{P}'$ if $\text{max}_{p \in \mathcal{P}'} \text{sim}(p_1, p) < \text{max}_{p \in \mathcal{P}'} \text{sim}(p_2, p)$. To this end, we utilize NetSimile, a size-independent graph similarity approach based on distance between feature vectors [8]. It is scalable with runtime complexity linear to the number of edges.

### 7.3 CPS-Randomized Greedy Algorithm

The canned pattern selection algorithm is as follows. First, it retrieves the default pattern set (1-path, 2-path, 3-cycle and 4-cycle). Next, it prunes candidate patterns whose sizes do not satisfy the plug specification or are “nearly-unique” (i.e., $\text{freq}(p) < \delta$ where $\delta$ is a pre-defined threshold). Note that the latter patterns have very low occurrences in $G$ and are unlikely to be as useful for query construction in their entirety. Then, it selects $\mathcal{P}$ from the remaining candidates.

Recall from Section 7.1, the cps problem can be cast as a maximization of submodular function problem subject to cardinality constraint. Recently, the algorithm community has proposed a technique with quality guarantee in [13] to address it. We exploit this approach, referred to as **CPS-Randomized Greedy** (CPS-R-Greedy, Algorithm 6), in our cps problem. To the best of our knowledge, this approach has not been utilized for graph querying.

In particular, CPS-R-Greedy extends the discrete greedy algorithm [34] using a randomized approach. At every step, a random candidate pattern is chosen from a set of “reasonably good” candidates (Lines 19-22). Intuitively, these candidates should have very few edge crossings, good coverage and are different from patterns already in $\mathcal{P}$. These candidates are identified as follows. For every candidate pattern $p$, we compute the pattern set score (Definition 7.1) assuming $p$ is added to the canned pattern set. A “good” candidate $p$ improves on the score of the set when it is added (Definition 7.2). Note that $\text{cog}_i$, $\text{cog}_p$ and $\text{sim}$ change as $\mathcal{P}$ changes. Hence, we recompute them at every iteration. Then, we randomly select a “good” candidate and assign it to $\mathcal{P}$. The algorithm terminates either when the set contains the desired number of patterns or when there exists no more good candidates. The following quality guarantee can be derived from [13].

**Theorem 7.7.** CPS-R-Greedy achieves $\frac{1}{2}$-approximation of CPS.

---

In the case, a user is interested in patterns with low coverage, $\delta$ can be set to 0 along with the reduction in $\text{freq}(\mathcal{P})$ in $\text{cog}(\mathcal{P})$ (Defn. 7.3).
Algorithm 6 CPS-R-Greedy.

Require: Candidate pattern set \( I_{all} \) and its frequency \( f(q/p) \); pattern budget \( b = (\eta_{min}, \eta_{max}) \).
Ensure: Pattern set \( \mathcal{P} \).

1. \( s_{best} \leftarrow 0 \)
2. while \( y > 0 \) do
3. \( P_{best} \leftarrow \phi \)
4. \( s_{map} \leftarrow \phi \)
5. \( C \leftarrow \phi \) (list of good candidates) //
6. for \( p \in I_{all} \) do
7. \( f_{good}(p) \leftarrow \text{GetCover}(p, \mathcal{P}, f(q/p)) \)
8. \( f_{freq}(p) \leftarrow \text{GetCognitiveLoad}(p, \mathcal{P}) \)
9. \( f_{sim}(p) \leftarrow \text{GetSimilarty}(p, \mathcal{P}) \)
10. \( s \leftarrow f_{good}(p) - f_{sim}(p) - f_{freq}(p) + 2 \)
11. if \( s > s_{best} \) and \( |\mathcal{P}| = 0 \) then
12. \( s_{best} \leftarrow s \)
13. \( P_{best} \leftarrow P \)
14. else if \( s > s_{best} \) then
15. \( s_{map} \leftarrow \text{UpdateScore}(s_{map}, s, p) \)
16. \( C \leftarrow C \cup \{p\} \)
17. end if
18. end for
19. if \( |C| > 0 \) then
20. \( P_{best} \leftarrow \text{RandomChoose}(C) \)
21. \( s_{best} \leftarrow \text{GetScore}(s_{map}, P_{best}) \)
22. end if
23. if \( P_{best} \neq \phi \) then
24. \( \mathcal{P} \leftarrow \mathcal{P} \cup \{P_{best}\} \)
25. \( I_{all} \leftarrow I_{all} \setminus \{P_{best}\} \)
26. \( y \leftarrow y - 1 \)
27. end else
28. break
29. end if
30. end while

Frequent subgraphs are generated using Peregrine[29] (downloaded from [3]). These subgraphs are considered as candidates from which the canned patterns are selected using our algorithm in Section 7.3.

Query sets and GUI. We use different query sets for the user study and automated performance study. We shall elaborate on them in respective sections. The GUI used for these studies is available at https://youtu.be/sL0yHV1eEPw.

Parameter settings. Unless specified otherwise, we set \( \eta_{min} = 3 \), \( \eta_{max} = 15 \), \( y = 30 \), \( \delta = 3 \), and \( \epsilon = 5 \).

Performance measures. We measure the performance of TATTOO using the following.

(1) Time: Execution time of TATTOO.
(2) Memory requirement (MB): Peak memory usage when using TATTOO.
(3) Reduction ratio (denoted as \( \mu \)): Given a subgraph query \( Q \), \( \mu = \frac{\text{steps}}{\text{stepPotal}} \), where \( \text{steps} \) is the minimum number of steps required to construct \( Q \) when \( \mathcal{P} \) is used and \( \text{stepTotal} \) is the total number of steps needed when edge-at-a-time approach is used. Note that the number of steps excludes vertex label assignments which is a constant for a given \( Q \) regardless of the approach. For simplicity in automated performance study, we follow the same assumptions in [26]: (1) a canned pattern \( p \in \mathcal{P} \) can be used in \( Q \) if \( p \subseteq Q \); (2) when multiple patterns are used to construct \( Q \), their corresponding isomorphic subgraphs in \( Q \) do not overlap. In this user study, we shall jettison these assumptions by allowing users to modify the canned patterns and no restrictions are imposed (i.e., \( \text{stepPotal} \) does not need to be minimum). Smaller values of \( div \) imply better pattern diversity. For ease of comparison, the diversity plots are based on the inverse of \( div \).

8.2 User Study

We undertake a user study to demonstrate the benefits of using our framework from a user’s perspective. 27 unpaid volunteers (ages from 20 to 35), who were students of, or, researchers within different majors took part in the user study. None of them has used our GUI prior to the study. First, we presented a 10-min scripted tutorial of our GUI describing how to visually formulate queries. Then, we allowed the subjects to play with the tool for 15 min.

For each dataset, 5 subgraph queries with size in the range [10-28] are selected. These queries mimic topology of real-world queries containing various structures described in Section 5.2. To describe the queries to the participants, we provided printed visual sub-graph queries. A subject then draws the given query using a mouse in our GUI. The users are asked to make maximum use of the patterns to this end. Each query was formulated 5 times by different participants. We ensure the same query set is constructed in a random order (the order of the query and the approach are randomized) to counterbalance learning effects.
The canned patterns on the GUI are grouped by size and displayed using ForceAtlas2 layout [28] in different pages according to their sizes. This multi-page-based organization yields faster average query formulation time and fewer steps compared to other alternatives.

Display layout of canned patterns. We first explore three different graph layout algorithms, namely, ForceAtlas2 [28], Fruchterman Reingold (FR) [19] and Dagre [35] to determine the most suitable layout for displaying canned patterns our GUI. Ten participants were asked to construct a set of 5 queries each for the Amazon and YouTube datasets. Each participant repeated this experiment 3 times where a different graph layout was used on the canned pattern set each time. Figures 8 and 9 plot the average query formulation time and average number of steps. In general, the ForceAtlas2 layout yielded faster average QFT compared to FR and Dagre. Participants also took fewer steps using ForceAtlas2 compared to the rest. Hence, in subsequent experiments, TATTOO leverages ForceAtlas2 layout to display the canned patterns.

Learning effect. Since the same query set is used repeatedly for each approach, there may be a learning effect where volunteers start to commit the query set to memory if the study is conducted in a fixed order. Particularly, approaches that are tested last in the study may gain an unfair advantage over earlier approaches. We investigate it further with an experiment. Ten participants (U_f) were asked to construct 5 queries on the Amazon and YouTube datasets in a fixed order while another ten participants (U_r where U_f ∩ U_r = ∅) were asked to construct the same query set in a random order (the order of the query and the approach are randomized) to minimize learning effects. Figure 10 reports the average time taken for query formulation. Interestingly, formulation time is generally faster using TATTOO’s canned patterns compared to graphlets regardless of the order on the query and approach. We further examined the difference in average formulation time (i.e., t_Graphlet − t_TATTOO) across queries and datasets for these two orders. In particular, the average time difference for Amazon (resp. YouTube) is 5.5s (resp. 5.5s) and 5.9s (resp. 3.7s) for fixed order and random order, respectively. This is possibly due to the learning effect. Hence, in subsequent experiments we follow the randomized order to minimize learning effect.

Visual mapping time. In order to use canned patterns for query formulation, a user needs to browse the pattern set and visually map them to her query. We refer to this as visual mapping time (VMT). For each pattern used, we record the pattern mapping time (PMT) as the duration when the mouse cursor is in the Pattern Panel to the time a user selects and drags it to the Query Canvas. The VMT of a query is its average PMT. Intuitively, a longer VMT implies greater cognitive load on a user. Figure 11 shows the VMT of TATTOO patterns, graphlets, frequent subgraphs, and random patterns on AM and YT datasets. On average, TATTOO patterns consume the least VMT.

We investigate the effect of various GUI canned pattern layout options (i.e., single page (SP); group by size (GS); 4 per page (4P); 8 per page (8P); 16 per page (16P); sort by cognitive load (SL), diversity (SD) and sort by coverage (SC)) on VMT for AM and YT datasets. Note that SP arranges the patterns in the order that they are identified. The plug was set to b = (4, 15, 30, [3, 6]). Figure 12 shows that the average VMT for layout options with multiple pages tends to be shorter (up to 33.1%) than those in a single page (i.e., SP, SL, SD and SC). When the patterns are organized in pages, an increased number of pages reduces the need for a user to scroll and browse the patterns on a particular page and increases the need to toggle between various pages to identify useful patterns. Compared to single page options, the multi-page options (GS, 4P, 8P, 16P) achieve superior performance primarily due to the former. Hence, the multi-page-based organization is used for our user study.

Query formulation time (QFT) and number of steps. Figures 13 and 14 plot the average QFT and the average number of steps taken, respectively, for AM and YT. Note that a QFT includes the VMT and the steps include addition/deletion of nodes and edges and merger of nodes. As expected, the edge-at-a-time approach took the most
The number of patterns on a DBQ5(26)GO SK AM GO AM YT AM Others CGWe also conducted a post-study question-
RP Q5(26)RT AM Q4(21)LJ LJ 30. Increase in
and select appropriate patterns (longer
it results in two opposing effects: (1) longer time needed to b rowse
for query formulation, reducing query formulation steps. F urther,
patterns to select relevant ones. Hence, we investigate the effect of
naire to gain some qualitative feedback on
T/a.sc/t.sc/t.sc/o.sc/o.sc
is statistically significant (\(|P|<0.05\)) for 79.4% of the compar-
ions (Figure 15). In particular, it takes up to 18X, 9.3X, 6.7X, 8X, 9X, and 9X fewer steps compared to edge-at-a-time, default pat-
ttern, random patterns, graphlet, frequent patterns, and CATAPULT-generated patterns, respectively. For QFT, TATTOO is up to 9.7X, 8.6X, 9X, 6.6X, 7.1X, and 7.4X faster, respectively. The results are qualitatively similar in other datasets. Note that we can run CATAPULT only on AM for reasons discussed later.

**Effect of |P|.** The number of patterns on a gui may also impact a user cognitively as larger |P| means a user needs to browse more patterns to select relevant ones. Hence, we investigate the effect of |P| on QFT and the number of steps (Figure 16). Interestingly, QFT and steps are reduced by average of 12% and 22% (maximum reduction of 77% and 80%), respectively, when |P| is increased from 5 to 30. Increase in |P| exposes more patterns that could be leveraged for query formulation, reducing query formulation steps. Further, it results in two opposing effects: (1) longer time needed to browse and select appropriate patterns (longer vmt) and (2) potentially more and larger patterns available for query construction resulting in fewer construction steps and shorter QFT. The latter effect dominates.

Qualitative evaluation. We also conducted a post-study question-
aire to gain some qualitative feedback on TATTOO. Overall, all participants prefer to use pattern-at-a-time approach compared to edge-at-a-time approach and TATTOO’s canned patterns are rated the most useful with average rating of 4.5 on a 5-point Likert scale. Graphlets, frequent and random patterns have average ratings of 3.5, 4 and 1.75, respectively.

In addition, participants were queried using an adapted nasa-
tlx [22] questionnaire (on a 5-point Likert scale) regarding the mental demand, performance and frustration level of query formulation using canned patterns generated from various approaches. Briefly, NASA-TLX is a commonly used tool for assessing perceived workload based on user inputs in the form of a questionnaire and consists of ratings for 6 categories\(^4\), namely, mental demand, physical demand, temporal demand, performance, effort and frustration. Since we do not impose a time limit on the query formulation task, there is no temporal demand. Physical demand is also negligible in our problem setting. In particular, effort and mental demand are equivalent in this case and can be associated directly with cogni-tive load. The mental demand is 2.5, 3, 3.5 and 4.25 for TATTOO, frequent patterns, graphlets and random patterns, respectively, where larger values imply greater mental demand. In terms of performance, it is 4.25, 4, 3.75, and 3.5, respectively, where larger values are associated with better performance. The frustration level is 2.75, 2.75, 3 and 4, respectively, where larger values relates to more frus-tration. This highlights the benefits of canned patterns generated by TATTOO.

Some users elaborated on their preferences. Table 3 lists the key comments by these users. Several users highlighted that they found edge-at-a-time approach tedious to use compared to pattern-at-a-time approach due to the repetitive task of drawing vertices and edges. This is consistent with HCI research as remarked in Section 1. They also felt that canned patterns of TATTOO are more diverse and easy to map to the query graphs. Hence, using them during query construction do not require much effort, and the pat-
tterns are useful in speeding up query formulation. Lastly, several users highlighted the usefulness of default patterns in extending other larger canned patterns during query formulation.

**8.3 Automated Performance Study**

In this section, we evaluate TATTOO from the following perspec-tives. First, we compare the runtime and quality of patterns of TAT-
OO with the baseline approaches (Exp 1, 2). Second, we present re-sults that support our design decisions (Exp 3, 4, 5). To this end, we generate 1000 queries (size \([4-30]\)) for each dataset where 500 are randomly generated and remaining (evenly distributed) are path-
like, tree, star-like, cycle-like and flower-like queries.

**Exp 1: Run time.** First, we evaluate the generation time of dif-
ferent patterns types in canned pattern sets. Figure 17 (top) shows the results. In particular, generation of chord-like patterns requires significantly more time (up to 146% more for l) than other pattern types. This is primarily due to checks for different types of edge merger required for ccps. Figure 17 (bottom) reports the time taken

\(^4\)Mental demand can be interpreted as the cognitive load on the user; Physical demand assesses the amount of physical activity required for the task; Temporal demand is related to the time pressure experienced based on the pace of the task; Performance measures a user’s satisfaction with perform ance of the task; Effort can be interpreted as the overall mental and physical demand needed to perform the task; Frustration level sets out to measure user’s feelings (i.e., irritated, stressed and annoyed versus content, relaxed and complacent) during performance of the task.
Table 3: Examples of user comments.

| Index | Comment |
|-------|---------|
| 1     | I like to use canned patterns to draw a query graph because it is faster and less tedious. Constructing the query graph each vertex and each edge at a time is just too tedious and boring! |
| 2     | The random and graphlet patterns appear very cluttered and confusing! It takes me considerable time to figure out if I can use them or not for my query. |
| 3     | The default patterns are simple and easy for me to figure out. But I need to choose several of them repeatedly for a large query. The Tattoo patterns are a great complement to the default patterns to make the drawing faster. |
| 4     | I find it easy to map the Tattoo patterns to a query compared to other patterns like the random ones. It is actually much faster to construct a query using them than if I were to draw the vertices and edges one at a time. |
| 5     | I like the ease of use of the GUI. I can just use one interface to query different datasets. I don’t need to switch to different interfaces for different sources. |
| 6     | The default patterns are very useful for extending other bigger patterns like those from Tattoo and graphlets when I draw a subgraph query. |

Tattoo selects canned patterns efficiently within a few minutes. Observe that the time cost for the small pattern extraction phase is small in practice. In general, pattern selection is the most expensive phase and requires a couple of minutes or less. Results are qualitatively similar for other datasets. Figure 18 plots the memory requirement for Tattoo. It is largely dependent on the size of the dataset where the largest dataset $L_{j}$ has the greatest memory cost. Lastly, observe that Tattoo is 735X faster than Catapult, which is not designed for large networks. Except AM, other datasets either cannot be processed by Metis or fail to generate patterns in a reasonable time (within 12 hrs) due to too many possible matches of unlabelled graphs that require expensive graph edit distance computation. In the sequel, we shall omit discussions on Catapult.

Exp 2: Comparison with graphlets, frequent subgraphs, and random patterns. Next, we compare Tattoo’s patterns with those of graphlets (30 patterns derived from graphlets). Figure 19 reports the results. Observe that Tattoo’s patterns are superior to graphlets in all aspects. The results are qualitatively similar for other datasets. Note that coverage is not examined since it is 100% in all cases as all queries can be constructed using 2-node graphlet.

We compare the canned pattern set derived from frequent subgraphs generated by Peregrine (denoted as $P_{P}$) to those generated by Tattoo. We observe that Peregrine failed to extract larger size patterns (i.e., $|V| \geq 8$) within 12 hrs for all networks. Specifically, for RP, RC, and KT (resp. AM), it was able to extract frequent patterns of size $|V| \leq 7$ (resp. $|V| \leq 6$) within 2.5 hrs. For BT and DB (resp. YT, Lj, SK, and GO) it can extract upto size $|V| \leq 5$ (resp. $|V| \leq 4$) within 2.5hrs. However, it took around 39 hrs on AM to yield a meaningful number of candidate patterns (994 patterns with size $|V| \leq 7$ and $|E| \leq 21$) when the minimum threshold is set to 100. Hence, Tattoo is orders of magnitude faster than frequent pattern-based solution. Consequently, we restrict the canned pattern sets of both Tattoo and $P_{P}$ to 30 patterns with $|V| \leq 7$ and $|E| \leq 21$ for AM in our experiments for fair comparison. Consistent with our user study, Tattoo’s pattern set is superior to $P_{P}$ in most aspects. The average coverage, cognitive load, diversity and $\mu$ for Tattoo (resp. Peregrine) are 0.3 (resp. 0.27), 0.15 (resp. 0.14), 0.64 (resp. 0.59) and 0.23 (resp. 0.24), respectively.

The comparison with random patterns are reported in Figure 20. We observe that Tattoo’s patterns result in higher $\mu$, and are significantly lower in cognitive load (up to 3.2X) when compared to random patterns. Tattoo’s patterns is up to 9% less diverse compared to random patterns. The greater diversity of the random pattern set is likely due to the unrestricted way in generating the random patterns as compared to pattern generation of Tattoo which are partially derived from defined structures such as trusses, paths, cycles and stars. Despite the greater diversity, random patterns are more difficult to use in practice (Section 8.2) likely due to the greater cognitive load necessary to interpret the patterns.

Exp 3: Measuring cognitive load. We now justify the choice of our proposed cognitive load measure. Specifically, we compare several ways of measuring cognitive load of a pattern $p$, namely,
Then, the graphs are ranked according to the five cognitive load quality of canned patterns in terms of number of canned patterns on a dataset produces 266 a pattern set is the average value for respective measures. Importantly, cognitive load (Figure 22). Here the cognitive load and diversity compared to stars and small patterns from \( \mathcal{G}_\text{O} \) and \( \mathcal{G}_\text{T} \) contribute to higher \( \mu \) and greater diversity over the default patterns, respectively. That is, despite 100% coverage of the latter, it is less efficient (i.e., more number of steps for query formulation) than the former. Also notice the increase in cognitive load due to the inclusion of chord-like patterns as they are likely to be denser than others. Hence, patterns from \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \) complement the default patterns by improving \( \mu \) and diversity.

**Exp 5:** Generating patterns from \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \). Lastly, we examine the (a) benefits brought by graph decomposition on pattern extraction and (b) the characteristics (i.e., average cognitive load, \( \mu \), and average diversity) of canned pattern sets generated from \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \) and compare them with the default patterns (i.e., \( \mathcal{G}_\text{T} \)- and \( \mathcal{G}_\text{O} \)-oblivious). Graph decomposition reduces the pattern extraction time for \( k \)-truss-like structures, as well as, stars and small patterns across all datasets (Figure 23). The effect is most prominent for extraction of \( k \)-truss-like structures where it is up to 81.5X faster when graph decomposition is applied. For extraction of stars and small patterns, it is up to 24.3X faster with graph decomposition. Hence, this justifies our decision to perform truss-based graph decomposition.

Figure 24 reports the characteristics of the canned pattern sets. Note that coverage is 100% for the three cases as all queries can be constructed using the default pattern \( D_1 \) (Figure 7). On the other hand, Tattoo’s canned patterns achieve 49% and 78% average coverage of \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \), respectively. Observe that patterns obtained from \( \mathcal{G}_\text{O} \) and \( \mathcal{G}_\text{T} \) contribute to higher \( \mu \) and greater diversity over the default patterns, respectively. That is, despite 100% coverage of the latter, it is less efficient (i.e., more number of steps for query formulation) than the former. Also notice the increase in cognitive load due to the inclusion of chord-like patterns as they are likely to be denser than others. Hence, patterns from \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \) complement the default patterns by improving \( \mu \) and diversity.

**Exp 4:** Chord patterns vs \( k \)-trusses. Next, we show the benefits of using \( k \)-CP/CCPS (i.e., \( k \)-truss-like structures) compared to simply utilizing \( k \)-trusses as topology for canned patterns (recall from Section 5.2). We generate 100 random queries of size \([4-30]\) from \( \mathcal{G}_\text{T} \) and these yielded 11 \( k \)-CP/CCPS and 3 \( k \)-trusses. Observe that \( k \)-CP/CCPS improve both \( \mu \) and diversity but have poorer cognitive load (Figure 22). Here the cognitive load and diversity of a pattern set is the average value for respective measures. Importantly, more \( k \)-CP/CCPS than \( k \)-trusses satisfying the plug are generated due to relaxed structure of the former. For instance, the RP dataset produces 266.67% more \( k \)-CP/CCPS due to the small size of \( \mathcal{G}_\text{T} \) (see Table 2). That is, \( k \)-trusses may not result in sufficient number of canned patterns on a GUI. Hence, chord patterns improve the quality of canned patterns in terms of \( \mu \) and diversity compared to \( k \)-trusses and yielded more candidate patterns.

**Figure 23:** Effect of graph decomposition.

**Figure 24:** Patterns from \( \mathcal{G}_\text{T} \) and \( \mathcal{G}_\text{O} \).

\[
f_{\text{cog}1} = \frac{1}{2} \sum_{x \in \{s_{zp}, d_p, cr_p\}} (1 - e^{-x}); \quad f_{\text{cog}2} = \frac{1}{(1 + e^{-0.5x(s_{zp} + d_p + cr_p - 10)})};
\]
\[
f_{\text{cog}3} = s_{zp} + d_p + cr_p; \quad f_{\text{cog}4} = s_{zp} \times d_p \quad \text{(used in [26])}; \quad \text{and} \quad f_{\text{cog}5} = cr_p \quad \text{(recall \( s_{zp}, d_p, cr_p \) from Sec. 7.2).}
\]

We select \( f_{\text{cog}2} \) as the cognitive load measure since it is in the range of \([0, 1]\) and facilitates easy formulation of a non-negative and non-monotone submodular pattern score function (Theorem 7.5).

**Case Study**

In this section, we describe the application of Tattoo for visual query formulation on the Amazon and RoadNet-TX datasets. We generate 30 canned patterns of sizes between 4 and 15 from each of these datasets. Figures 25(a) and 25(b) depict some of the patterns selected from Amazon and RoadNet-TX, respectively. Observe that the patterns are different for different datasets. Specifically, patterns in red rectangle boxes in Figures 25(a) are not found in Figures 25(b) and vice versa. This emphasizes the fact that different datasets may expose different collection of canned patterns to aid efficient query formulation. Second, observe that the patterns have low cognitive load as one can easily recognize their topology with a quick glance. In particular, patterns encapsulated by blue ellipses (solid lines) in Figures 25(a) and (b) are examples of some patterns derived from \( \mathcal{G}_\text{T} \).

We now illustrate efficient query construction using canned patterns. Suppose one is interested in making a new movie involving Ben Affleck and Matt Damon. She would like to identify other actors that have prior working experience with these two actors. She may construct a subgraph query such as the one in Figure 25(c) to query the Amazon dataset containing movie titles and associated actors. Query formulation takes 8 steps (\(~20s\)) by utilizing one canned pattern (highlighted by green ellipse with broken line in Figure 25(a)) and adding an edge. Note that the steps taken include vertex label assignment. In comparison, edge-at-a-time requires a total of 20 steps (\(~39s\)).

**10 Conclusion & Future Work**

Canned patterns play a pivotal role in supporting efficient visual subgraph query formulation using direct-manipulation interfaces.
We present work, we plan to explore the problem in a distributed setting s. [12] Angela Bonifati, Wim Martens and Thomas Timm. 2017. Analytical study of large sparql query logs. Proceedings of the VLDB Endowment 11, 2 (2017), 149-161.

Acknowledgements. The first four authors are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. Wook-Shin Han was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-01398). Byron Choi is supported by HKBU12201518.

REFERENCES
[1] 2021. Neo4j Bloom. https://neo4j.com/bloom.
[2] 2021. RAPID. https://research.csc.ncsu.edu/coul/RAPID/RAPIDAnalytics/.
[3] 2021. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.
[4] Foto N. Afrati, Dimitris Fotakis and Jeffrey D. Ullman. 2013. Enumerating subgraph instances using map-reduce. In IEEE 29th International Conference on Data Engineering. IEEE, 62-73.
[5] Nestren K. Ahmed, Jennifer Neville, Ryan A. Rossi and Nick Duffield. 2015. Efficient graphlet counting for large networks. In 2015 IEEE International Conference on Data Engineering. IEEE, 1-10.
[6] Michele Berlingerio, Danai Koutra, Tina Ealissi-Rad and Christos Faloutsos. 2013. Network similarity via multiple social theories. In Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. IEEE/ACM, 1439-1440.
[7] Sourav S. Bhownick, Byron Choi and Curtis E. Dyreson. 2016. Data-driven visual graph query interface construction and maintenance: challenges and opportunities. Proceedings of the VLDB Endowment 9, 12 (2016), 984-992.
[8] Sourav S. Bhownick, Kai Huang, Huey Eng Chua, Zifeng Yuan, Byron Choi and Shuigeng Zhou. 2020. 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. ACM, 2689-2692.
[9] Avradeep Bhownik, Vivek Borkar, Dinesh Garg and Madhavan Pallam. 2014. Submodularity in team formation problem. In Proceedings of the 2014 SIAM International Conference on Data Mining. SIAM, 895-901.
[10] Angela Bonifati, Wim Martens and Thomas Timm. 2017. Analytical study of large sparql query logs. Proceedings of the VLDB Endowment 11, 2 (2017), 149-161.

Figure 25: Case study.
A PROOFS

Proof of Theorem 4.2 (Sketch). The CPs is a multi-objective optimization problem which can be reformulated as a constrained single-objective optimization problem where the objective function is max \( f_{\text{com}} \) and the constraints are \( \min \{ f_{\text{sim}}, f_{\text{com}} \} \). This reformulated problem (i.e., max \( f_{\text{com}} \)) can be reduced from the maximum coverage problem, which is a classical NP-hard optimization problem [30]. In particular, given a number \( k \) and a collection of sets \( S \), the maximum coverage problem aims to find a set \( S' \subset S \) such that \( |S'| \leq k \) and the number of covered elements is maximized. In CPs, the collection of sets \( S \) is the set that consists of all possible subgraphs of the graph dataset \( D \). The subset \( S' \) is the pattern set and \( k \) is the size of the pattern set. The number of covered elements corresponds to the number of covered subgraphs in \( D \). Note that the reformulated optimization problem is at least as hard as the maximum coverage problem since optimizing the objective may result in solutions that are sub-optimal with regards to additional imposed constraints.

Proof of Lemma 6.1 (Sketch). In Algorithm 2, the worst-case time complexity is due to Lines 13 to 20 which computes the truss-ness of each edge \( e \in E_T (\sqrt{|V_T|}) \) [44], updates \( f_{\text{req}}(C_k) \) and stores \( C_k \) in the candidate pattern set. Hence, the worst-case time complexity is \( O(|E_T|^{1.5} + |E_T|^{k_{\text{max}}}) \) since upper bound of \( k \) is \( k_{\text{max}} \). Algorithm 2 uses \( O(|E_T| + |V_T|) \) and \( O(|E_T|) \) space to hold \( G_T \) and \( t \), respectively. Further, all possible \( k \)-chord patterns \( (3 \leq k \leq k_{\text{max}}) \) and its frequency have to be stored in the worst-case \( O(k_{\text{max}}^2) \). Hence, worst-case space complexity is \( O(|E_T| + |V_T|) \) since \( |E_T| + |V_T| \geq k_{\text{max}} \) for large graph in practice.

Proof of Lemma 6.3 (Sketch). The simple 3-truss pattern \( C_3 = (V_3, E_3) \) is simply a triangle. Hence, \( \forall e = (u, v) \in E_3 \), there is a vertex \( w \) that is adjacent to both \( u \) and \( v \). That is, all edges in \( C_3 \) have similar structure. Hence, all different types of single edge merger between two \( C_3 \) produces a pattern with a merged edge \( e_m = (x, y) \) and vertices \( x \) and \( y \) have two common adjacent vertices \( w_1 \) and \( w_2 \) which is essentially \( C_3 \) where its truss edge corresponds to the merged edge of the two \( C_3 \) (Figure 26).

Proof of Lemma 6.4 (Sketch). Observe that \( k \)-chord pattern on an edge \( e = (u, v) \) implies that \( k \)-2 triangles in the graph contains. Since \( |NB_{BC}(k, e)| \) is the set of nodes \( W \) adjacent to \( u \) and \( v \) such that \( \forall w \in W, \{ t((u, w), e) \} \geq k \) and \( t((v, w), e) \geq k \), \( |NB_{BC}(k, e)| \) is equivalent to the number of triangles around \( e \). Hence, when \( |NB_{BC}(k, e)| \leq (k - 2) \), a \( k \)-chord pattern must exist on \( e \).

Proof of Theorem 6.5 (Sketch). In Algorithm 3, for each edge \( e \in E_T \), there are \( k_1 \times |NB_{BC}(k_1, \{ t_1 \}) | \) iterations that computes the procedures \( GetTN (|O(k_{\text{max}})|) \). \( GetNN (O(k_{\text{max}} \times |E_{\text{max}}|)) \) where \( E_{\text{max}} \) is the k-CCP edge neighbourhood with the largest size. The worst-case time complexity is \( O(k_{\text{max}}^2 \times |E_T| \times |E_{\text{max}}|^2) \) since \( k_{\text{max}} \) is the upper bound of \( k_1 \). Algorithm 3 requires \( O(|V_T| + |E_T|) \) and \( O(k_{\text{max}} \times |E_T|) \) to store \( G_T \) and \( |NB_{BC}(k, e)| \), respectively. In the worst-case, all possible combinations of \( CCP_m(n(k), k) \), \( CCP_m(n(k), k) \) and \( CCP_m(n(k), k) \), and their respective frequency are stored \( O(k_{\text{max}}^2 \times |E_T|) \). The worst-case space complexity is \( O(k_{\text{max}}^2 \times |E_T|) \) since \( k_{\text{max}}^2 \times |E_T| + |V_T| \geq k_{\text{max}}^2 \times |E_T| \) for large graph in practice.
Proof of Lemma 6.6 (Sketch). In the worst-case, finding the stars and asterisk patterns requires performing BFS for each vertex in $V_0$. In the worst case, the graph is strongly connected and every other vertex in $V_0$ is visited during the BFS. Hence, the worst-case time complexity is $O(|V_0|)$. Algorithm 4 requires $O(|V_0| + |E_0|)$ space for storing $G_0$. In the worst-case, there are $deg_{max} = e + 1$ and $deg_{max} = e + 1$ possible $S_k$ and $A_S$, respectively. Since $deg_{max}$ occurs when every node $v \in V_0$ is connected to every other nodes in $V_0$, $deg_{max}$ has worst-case complexity $O(|V_0|)$. Hence, storage of $S_k$ and $A_S$ requires $O(|V_0|) + O(|V_0|^2)$, respectively and Algorithm 4 requires $O(|V_0| + |E_0|)$ space in the worst-case.

Proof of Lemma 6.7 (Sketch). In Algorithm 5, the worst-case time complexity is due to the graph isomorphism check ($O(\eta_{max}!\eta_{max})$ [16]) on Line 26 which is within a for-loop with maximum of $|V_0|$ iterations. Hence, the worst-case time complexity is $O(\eta_{max}!\eta_{max})$. Algorithm 5 requires $O(|V_0| + |E_0|)$ space for storing $G_0$. Since every $k$-path ($P_k$), $k$-cycle ($Y_k$) and subgraphs with unique topology ($U$) consists of multiple nodes, the number of possible $P_k$, $Y_k$ and $U$ is less than $|V_0|$ and the storage required will be $O(|V_0|)$. Hence, the worst-case space complexity is $O(|V_0| + |E_0|)$.

Proof of Lemma 7.3. Submodular functions satisfies the property of diminishing marginal returns. That is given a set of $n$ elements ($N$), a function $f()$ is submodular if for every $A \subseteq B \subseteq N$ and $f \notin B, f(A \cup \{j\}) - f(A) \geq f(B \cup \{j\}) - f(B)$. Given a graph $G$ and canned pattern sets $P_A$ and $P_B$ where $P_A \subseteq P_B$, let the coverage of $P_A$ and $P_B$ be $f_{co}(P_A)$ and $f_{co}(P_B)$, respectively. Observe that $P_B$ consists of $P_A$ and additional patterns (i.e., $P' = P_A \cup \{P_B\}$. For each canned pattern $P \in P'$, we let $s = \min\{|f_{co}(P)|, |f_{co}(P_A)|\}$ and $K$ denotes the overlapping set $f_{co}(P) \cap f_{co}(P_A)$. The coverage of $P$ falls under one of four possible scenarios, namely, (1) $K = f_{co}(P)$ if $s = |f_{co}(P)|$, (2) $K = f_{co}(P_A)$ if $s = |f_{co}(P_A)|$, (3) is an empty set and (4) otherwise (i.e., $0 < |f_{co}(P) \cap f_{co}(P_A)| < s$).

In the case where coverage of every $P$ falls under scenario 1, then $f_{co}(P_A) = f_{co}(P_B)$. Should any $P$ falls under scenario 2, 3 or 4, then $f_{co}(P_A) \subseteq f_{co}(P_B)$. Hence, $f_{co}(P_A) \leq f_{co}(P_B)$. Consider a canned pattern $P' \notin P_A$, let $\min\{|f_{co}(P)|, |f_{co}(P_A)|\}$.

Suppose $f_{co}(P') \cap f_{co}(P_A) = f_{co}(P')$ where $|f_{co}(P')| < |f_{co}(P_A)|$ (Scenario 1), then $f_{co}(P_A \cup \{P'\}) - f_{co}(P_A)$ is an empty set. Note that we use the minus and subset operator interchangeably in this proof. Since $f_{co}(P_A) \subseteq f_{co}(P_B)$, $f_{co}(P_B \cup \{P'\}) = f_{co}(P_B)$. Hence, $f_{co}(P_A \cup \{P'\}) = f_{co}(P_B \cup \{P'\}) - f_{co}(P_A) = f_{co}(P_B)$.

Now, consider $f_{co}(P') \cap f_{co}(P_A) = f_{co}(P')$ where $|f_{co}(P')| > |f_{co}(P_A)|$ (Scenario 2). $f_{co}(P_A \cup \{P'\}) - f_{co}(P_A) = f_{co}(P') - f_{co}(P_A) = f_{co}(P_B) \setminus f_{co}(P_A)$, where $f_{co}(P_B) \setminus f_{co}(P_A)$. Let $L$ and $M$ be $f_{co}(P') \setminus f_{co}(P_A)$ and $f_{co}(P_B) \setminus f_{co}(P_A)$, respectively. Observe that, since $L$ is finite, in order to obtain the following inequality (i.e., (1) $L$ to be fully contained in $M$ if $|L| < |M|$, (2) $M$ to be fully contained in $L$ if $|M| < |L|$, (3) $L \cap M$ to be empty or (4) otherwise (i.e., $0 < |L \cap M| < t$ where $t = \min\{|L|, |M|\}$). Hence, $|L \cap M| \in [0, t]$. When $|L \cap M| = 0, f_{co}(P_A \cup \{P'\}) - f_{co}(P_A) = f_{co}(P_B \cup \{P'\}) - f_{co}(P_B)$. Otherwise, there are some common graphs covered by $L$ and $M$, resulting in $f_{co}(P_B \cup \{P'\}) - f_{co}(P_B) = 1 \in (L \cap M)$.

Proof of Theorem 7.5 (Sketch). Consider a partial pattern set $P'$ and a candidate pattern $p$. Suppose $p$ does not improve the set coverage of $P'$ and adds a high cost in terms of cognitive load and diversity. Then, $s(P') > s(P' \cup \{p\})$. Hence, the score function $s()$ is non-monotone. Since $f_{co}(P'), f_{sim}(P'), f_{co}(P') \in [0, |P'|]$, $f_{co}(P') - f_{sim}(P') - f_{co}(P') \in \mathbb{R}$ (Definition 7.1) is in the range $[2|P'|, |P'|]$. Hence, $f_{co}(P') - f_{sim}(P') - f_{co}(P') + 2|P'|$ (Definition 7.1) is in the range $[0, |P'|]$ and is non-negative. Since submodular functions are negations of submodular functions and that non-negative weighted sum of submodular functions preserve submodularity property [20], $s(P')$ is submodular. Note that adding a constant (i.e., $\frac{1}{3}|P'|$) does not change the submodular property [11] and ensures that $s(P')$ is non-negative. The scaling factors of $a_{f_{co}} = a_{f_{sim}} = a_{f_{co}} = \frac{1}{3}|P'|$ further bounds $s(P')$ within the range $[0, |P'|].$

Proof of Lemma 7.6. Consider a graph $G = (V, E)$ with $cr$ crossings. Since each crossing can be removed by removing an edge from $G$, a graph with $|E| - cr$ edges and $|V|$ vertices containing no crossings (i.e., planar graph). Since $|E| \leq 3|V| - 6$ for planar graph (i.e., Euler’s formula), hence, $|E| - cr \leq 3|V| - 6$ for $|V| \geq 3$. Rewriting the inequality, we have $cr \geq |E| - 3|V| + 6$.

Proof of Theorem 7.7. Let $A_i$ be an event fixing all the random decisions of Greedy for every iteration $i$ and $A_i$ be the set of all possible $A_i$ events. We denote $s(P_{i-1} \cup \{p_i\}) - s(P_{i-1})$ as $s_{p_{i}}(P_{i-1})$. Further, let the desired size of $P'$ be $y$, $1 \leq i \leq y$ and $A_{i-1} \in A_i$. Unless otherwise stated, all the probabilities, expectations and random variables are implicitly conditioned on $A_{i-1}$. Consider a set $M_i$ containing the patterns of $OPT \setminus P_{i-1}$ plus enough dummy patterns to make the size of $M_i$ exactly $y$.

Note that $|E(s_{p_{i}}(P_{i-1}))| = y^2 \cdot \sum p \in M_i s_{p_{i}}(P_{i-1}) \geq y^{-1} \cdot \sum p \in M_i s_{p_{i}}(P_{i-1}) \geq \frac{y^{-1} \cdot \sum p \in OPT \setminus P_{i-1} s_{p_{i}}(P_{i-1})}{y}$ [13], where the first inequality follows from the definition of $M_i$. 
(i.e., set of “good” candidate patterns) and the second from the submodularity of \( s(\cdot) \). Unfixing the event \( A_{i-1} \) and taking an expectation over all possible such events, \( \mathbb{E}[s_p(P_{i-1})] \geq \frac{\mathbb{E}[s(OPT \cup P_{i-1})] - \mathbb{E}[s(P_{i-1})]}{\gamma} \), where the second inequality is due to observation that for every \( 0 \leq i \geq \gamma \), \( \mathbb{E}[s(OPT \cup P_i)] \geq (1 - \frac{1}{\gamma})^i \cdot s(OPT) \).

We now prove by induction that \( (\frac{1}{\gamma})^i \cdot s(OPT) \geq \frac{1}{\gamma} \cdot (1 - \frac{1}{\gamma})^{i-1} \cdot s(OPT) \). Note that this is true for \( i = 0 \) since \( s(P_0) \geq 0 = \frac{1}{\gamma} \cdot (1 - \frac{1}{\gamma})^{-1} \cdot s(OPT) \). Further, we assume that the claim holds for every \( i' < i \) and prove it for \( i > 0 \).

\[
\mathbb{E}[s_p(P_{i-1})] \geq \mathbb{E}[s(P_{i-1})] + \frac{(1 - \frac{1}{\gamma})^{i-1} \cdot s(OPT) - \mathbb{E}[s(P_{i-1})]}{\gamma} = (1 - \frac{1}{\gamma}) \cdot \mathbb{E}[s(P_{i-1})] + \frac{1}{\gamma} \cdot s(OPT) \geq (1 - \frac{1}{\gamma}) \cdot (1 - \frac{1}{\gamma})^{i-1} \cdot s(OPT) + \frac{1}{\gamma} \cdot (1 - \frac{1}{\gamma})^{i-1} \cdot s(OPT) = \frac{1}{\gamma} \cdot (1 - \frac{1}{\gamma})^{i-1} \cdot s(OPT).
\]

Hence, \( \mathbb{E}[s(P_k)] \geq \frac{1}{\gamma} \cdot (1 - \frac{1}{\gamma})^{k-1} \cdot s(OPT) \). That is, Alg. 6 achieves \( \frac{1}{\gamma} \)-approximation of CPS.

**Proof of Theorem 7.8 (Sketch).** Let \( G_{\text{max}} = (V_{\text{max}}, E_{\text{max}}) \) be the largest candidate pattern in \( P_{\text{all}} \). In the worst-case, time complexity of Algorithm 6 is \( O(|P_{\text{all}}| |V_{\text{max}}| |V_{\text{max}}|) \) since there are \( |P_{\text{all}}| \) candidate patterns and the while-loop in Algorithm 6 iterates at most \( y \) times. For each iteration, the score function requires computation of coverage, cognitive load and redundancy which requires \( O(|V_{\text{max}}| |V_{\text{max}}|) \) and \( O(|V_{\text{max}}| + |V_{\text{max}}| \log |V_{\text{max}}|) \) respectively. Note that \( |V_{\text{max}}| \log |V_{\text{max}}| \approx |E_{\text{max}}| \) in real-world graphs [8]. The space complexity is due to storage of all candidate patterns. Hence, Algorithm 6 has space complexity of \( O(|P_{\text{all}}| (|V_{\text{max}}| + |E_{\text{max}}|)) \).