CATAPULT: Data-driven Selection of Canned Patterns for Efficient Visual Graph Query Formulation

Kai Huang [§,‡] Huey Eng Chua[‡] Sourav S Bhowmick[‡] Byron Choi[†] Shuigeng Zhou[§]

[§]Shanghai Key Lab of Intelligent Information Processing, Sch. of Computer Science, Fudan University, China
[‡]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[†]Department of Computer Science, Hong Kong Baptist University, Hong Kong
hechua|assourav@ntu.edu.sg,bchoi@comp.hkbu.edu.hk,khuang14|sgzhou@fudan.edu.cn

ABSTRACT

Visual graph query interfaces (a.k.a GUI) widen the reach of graph querying frameworks across different users by enabling non-programmers to use them. Consequently, several commercial and academic frameworks for querying a large collection of small- or medium-sized data graphs (*e.g.,* chemical compounds) provide such visual interfaces. Majority of these interfaces expose a fixed set of *canned patterns* (*i.e.,* small subgraph patterns) to expedite query formulation by enabling *pattern-at-a-time* in lieu of *edge-at-a-time* construction mode. Canned patterns to be displayed on a GUI are typically selected manually based on domain knowledge. However, manual generation of canned patterns is labour intensive. Furthermore, these patterns may not sufficiently cover the underlying data graphs to expedite visual formulation of a wide range of subgraph queries. In this paper, we present a generic and extensible framework called CATAPULT to address these limitations. CATAPULT takes a data-driven approach to *automatically* select canned patterns, thereby taking a concrete step towards the vision of data-driven construction of visual query interfaces. Specifically, it first *clusters* the underlying data graphs based on their topological similarities and then *summarize* each cluster to create a *cluster summary graph* (CSG). The canned patterns within a user-specified *pattern budget* are then generated from these CSGs by maximizing *coverage* and *diversity*, and minimizing *cognitive load* of the patterns. Experimental study with real-world datasets and visual graph interfaces demonstrates the superiority of CATAPULT compared to traditional techniques.

ACM Reference Format:
Kai Huang, Huey Eng Chua, Sourav S Bhowmick, Byron Choi, Shuigeng Zhou. 2019. CATAPULT: Data-driven Selection of Canned Patterns for Efficient Visual Graph Query Formulation. In 2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands. ACM, NewYork, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300072

1 INTRODUCTION

Large collections of small- or medium-sized data graphs are prevalent nowadays in a variety of domains such as cheminformatics, bioinformatics, drug discovery, and computer vision. For example, more than a million chemical compounds and drugs are publicly available from sources such as *DrugBank* (https://www.drugbank.ca/), *eMolecules* (https://www.emolecules.com/), and *PubChem* (https://pubchem.ncbi.nlm.nih.gov/). *Subgraph search* (a.k.a *substructure search*) is a popular query primitive for querying these data graphs. In this search paradigm, a set of data graphs containing *exact* or *approximate* match of a user-specified query graph (*i.e.,* subgraph query) is retrieved from the underlying data graphs [36]. As such data graph repositories continue to grow rapidly in size, frameworks to support the formulation and processing of subgraph queries have become increasingly important. In this paper, we focus on the subgraph query formulation problem.

A number of declarative query languages have been proposed for graphs (*e.g.,* SPARQL, *Cypher*) which can be utilized to formulate subgraph queries in textual form. All these languages assume that a user has programming and debugging expertise to formulate queries correctly. Unfortunately, this assumption makes it harder for non-programmers to take advantage of a graph querying framework. A popular approach to alleviate this problem is to provide a user-friendly

SIGMOD'19, June 30–July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300072

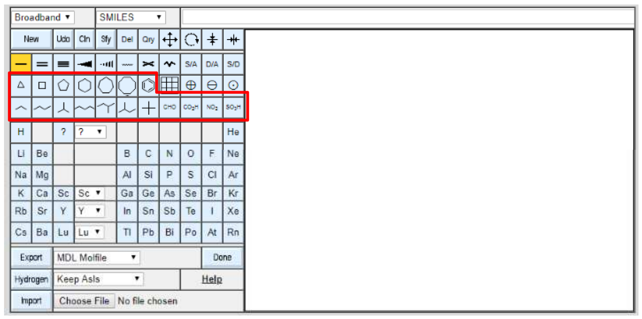

Figure 1: Visual graph query interface in *PubChem*.

visual query interface (a.k.a GUI) for interactive construction of queries. For instance, *PubChem* provides a visual query interface for structure-based chemical compound search (Figure 1). Similarly, *DrugBank* allows end users to draw a subgraph search query using a visual interface[1]. *Interestingly, both these data sources do not expose interfaces to formulate textual queries using a graph query language*, highlighting the reluctance of end users to use such programming languages.

A core component of many real-world visual subgraph query interfaces is a panel containing a set of *canned patterns* (*i.e.,* small subgraph patterns) to facilitate fast query formulation [6]. Specifically, a canned pattern (pattern for brevity) enables a user to construct multiple nodes and edges in a subgraph query by performing a *single* click-and-drag action (*i.e., pattern-at-a-time* mode) in lieu of iterative construction of edges one-at-a-time (*i.e., edge-at-a-time* mode). For example, the GUI in Figure 1 provides a library of canned patterns such as benzene ring and carboxyl group. Observe that exposition of canned patterns on a GUI potentially decrease the time taken to finish a visual query formulation task by reducing the number of formulation steps. Consequently, this enhances the usability of graph querying frameworks.

A recent study revealed that the selection of canned patterns for visual interfaces is typically performed manually based on domain knowledge [6]. Consequently, they suffer from two main drawbacks. First, manual selection of canned patterns is labour-intensive especially for a large graph repository. Second, it is very challenging even for a domain expert to garner a comprehensive knowledge of different topological structures in the underlying data graphs for canned pattern selection. As a result, the selected patterns may not be topologically *diverse* enough to expedite formulation of a wide range of subgraph queries. Consider the following example scenario that highlights these limitations.

Example 1.1. Urea derivatives such as those in Figure 2 have a wide range of applications. For instance, DCMU is an *algicide* whereas TMAD is an agent for *Mitsunobu reaction*[2].

[1]https://www.drugbank.ca/structures/search/small_molecule_drugs/structure

[2]*Mitsunobu reaction* is an important organic reaction that converts an alcohol into various functional groups [8].

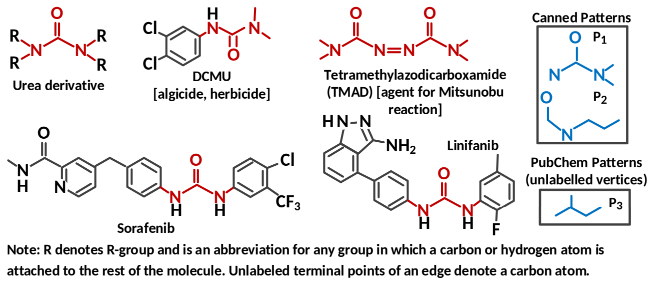
Urea derivative
DCMU
[algicide, herbicide]
Tetramethylazodicarboxamide
(TMAD) [agent for Mitsunobu reaction]
Canned Patterns
P_1
P_2
PubChem Patterns
(unlabelled vertices)
P_3
Sorafenib
Linifanib
Note: R denotes R-group and is an abbreviation for any group in which a carbon or hydrogen atom is attached to the rest of the molecule. Unlabeled terminal points of an edge denote a carbon atom.

Figure 2: Skeletal structures of some urea derivatives and related canned patterns (best viewed in color).

Diaryl urea derivatives such as sorafenib and linifanib are important cancer drugs that act as EGFR inhibitors [22].

Consider the canned patterns P_1 and P_2 in Figure 2, which are structurally similar to the urea functional group (red highlighted parts in urea derivative). Note that in these patterns single and double bonds are both represented as unweighted edges. Observe that P_1 can be used in its entirety to partially construct a subgraph query for retrieving data graphs containing DCMU and TMAD whereas P_2 can be utilized for constructing queries seeking information related to sorafenib, linifanib, and DCMU. For instance, consider a subgraph query for TMAD. It can be visually constructed by undertaking the following three steps.

- *Step 1*: Select and drag P_1 to the query construction canvas (QCC) in the GUI (P_{1a}).
- *Step 2*: Select and drag P_1 to QCC (P_{1b}).
- *Step 3*: Construct an edge between P_{1a} and P_{1b}.

Unfortunately, the canned pattern set of *PubChem* (Figure 1) does not contain patterns related to urea functional group[3]. In fact, it exposes a very limited and fixed set of canned patterns on its GUI for query formulation. Furthermore, vertices in these patterns are unlabeled. Consequently, in order to visually formulate the above query in the *PubChem* GUI, one may need to undertake the following steps.

- *Step 1:* Select and drag P_3 to QCC (P_{3a}).
- *Steps 2-6:* Label all vertices in P_{3a}.
- *Step 7:* Select and drag P_3 (P_{3b}) to connect to the vertex N in P_{3a}.
- *Steps 8-11:* Label all remaining vertices in P_{3b}.
- *Steps 12-17:* Select and drag three vertices labelled C to QCC. Connect these vertices to the partially constructed TMAD.

Notice that it takes 17 steps instead of 3 steps to visually formulate the aforementioned query in *PubChem* due to the lack of availability of P_1 in the canned pattern set. This naturally increases the task complexity of the *PubChem* interface as it may take a longer time to formulate a query. ∎

[3]This is also the case for the *DrugBank* GUI.

Is it possible to automatically select a superior collection of canned patterns that can expedite formulation of a wide variety of visual subgraph queries? In this paper, we answer to this question affirmatively by presenting a novel framework called CATAPULT (**C**anned p**A**ttern selec**T**ion for f**A**st gra**P**h q**U**ery formu**L**a**T**ion) that takes a *data-driven* approach to the canned pattern selection problem. Given a graph database D, a visual graph query interface $\mathbb{I}$, and a *pattern budget* b (*i.e.,* minimum and maximum size of canned patterns, number of patterns to display on $\mathbb{I}$), CATAPULT automatically selects canned patterns from D *satisfying* b by ensuring that these patterns not only have high *coverage* of D but also are highly *diverse.* Furthermore, it preferentially selects patterns that have potentially low *cognitive load* on end users. This is important as research in the information visualization community reveal that large and dense graphs overload the human perception and cognitive systems, resulting in poor performance of relatively complex tasks such as identifying relationship in graphs [20]. Comprehension of such relationships in canned patterns is a critical step during visual query formulation as a user needs to visually search these patterns to select relevant ones. Consequently, a pattern with high cognitive load may adversely impact the visual search time.

CATAPULT emphasizes the need to strive a balance between coverage, diversity, and cognitive load for selecting canned patterns. We advocate that this is important to the goal of efficient visual query formulation. To elaborate further, a single unlabeled edge as a canned pattern has complete coverage of any graph repository. However, such a canned pattern may not facilitate efficient visual query formulation due to edge-at-a-time construction mode. On the other hand, the pattern P_1 in Figure 2 has significantly lower coverage and higher structural diversity than an unlabelled edge. It also enables efficient formulation of the subgraph query in Example 1.1.

The canned pattern selection problem is NP-hard. Figure 3 depicts the architecture of CATAPULT to tackle it. Briefly, it comprises of three key components, namely *small graph clustering*, *cluster summary graph (CSG) generator*, and *canned pattern selector.* Given D, the *small graph clustering* module performs clustering of the data graphs using features such as *frequent subtrees* [10] and *maximum (connected) common subgraphs* [36]. Then, each cluster is *summarized* as a *cluster summary graph* (CSG) by "integrating" all data graphs in that cluster. Finally, the *canned pattern selector* component greedily generates candidate patterns from summarized CSGs in lieu of the underlying data graphs as the number of CSGs is significantly smaller than the number of data graphs. In particular, a *pattern score* that is sensitive to the coverage, diversity, and cognitive load of patterns is employed to select suitable patterns within the budget b for display on a GUI. CATAPULT also has a *sampler* component (*eager* and *lazy sampler*) to tackle very large repositories. As we shall see

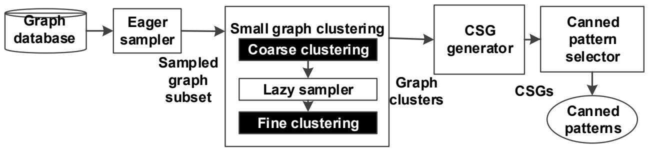

Figure 3: The CATAPULT framework.

in Section 6, our experimental study with real-world visual graph query interfaces reveals that CATAPULT can reduce the number of steps taken for query formulation by up to 85.7% and as a result makes query formulation more efficient.

It is worth noting that the benefit of CATAPULT is not limited to faster query formulation time. Due to its generic and extensible architecture, it can be easily utilized to automatically generate canned patterns for *any* domain-specific graph querying application (*e.g.,* drug discovery, computer vision) centered around a collection of small- or medium-sized data graphs. This naturally enhances the portability of CATAPULT across different visual graph query interfaces. Furthermore, it can be *extended* to support incremental maintenance of canned patterns as the underlying data graphs evolve.

In summary, this paper makes the following contributions. (a) We describe CATAPULT, an end-to-end canned pattern selection framework that can be utilized to populate the set of canned patterns in any visual graph query interface independent of domains and data sources. (b) To the best of our knowledge, we are the first to formally propose the novel *canned pattern selection problem* by analyzing the desired *properties* of a "good" canned pattern set and propose a novel system to mine them from the underlying data graphs. (c) Using real-world data graph repositories and visual graph query interfaces, we show the superiority and applicability of our proposed framework compared to state-of-the-art canned pattern selection techniques.

The rest of the paper is organized as follows. Section 2 introduces preliminary concepts. We formally define the *canned pattern selection problem* in Section 3. We present details of the CATAPULT framework in Sections 4 and 5. Section 6 details the experimental results. Related research is discussed in Section 7. Section 8 concludes the paper. Formal proofs of theorems and lemmas are given in Appendix A.

2 BACKGROUND

We denote a graph as $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of (directed or undirected) edges. Vertices and edges can have labels as attributes. Let l be the mapping function of G for labels of vertices or edges. That is, $l(v)$ and $l(u, v)$ are the labels of vertex $v \in V$ and edge $(u, v) \in E$, respectively. In this paper, we assume that G (data or query graph) is a connected graph with at least one edge. The *size* of G is defined as $|G| = |E|$. For ease of

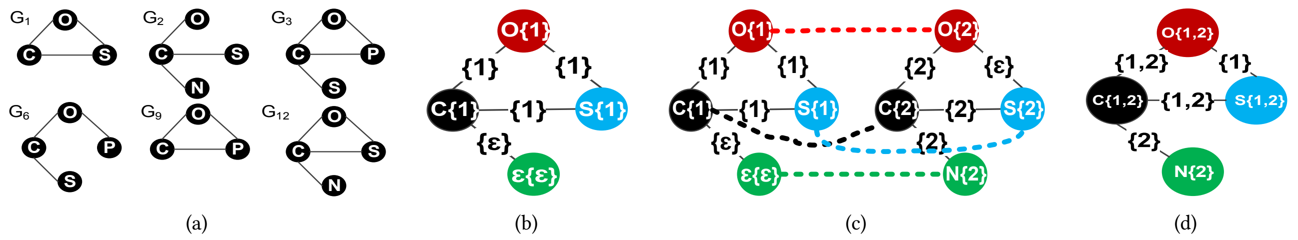

Figure 4: (a)A cluster containing six graphs ($G_1, G_2, G_3, G_6, G_9, G_{12}$); (b) extended graph of G_1; (c) mapping of extended graphs of G_1 and G_2 (dotted line denotes mapping of the vertex pair); (d) closure graph of G_1 and G_2.

presentation, we assume data graphs and visual subgraph queries as undirected simple graphs with labeled vertices.

A graph $G = (V, E)$ is a *subgraph* of another graph $G' = (V', E')$ if there exists a subgraph isomorphism from G to G', denoted by $G \subseteq G'$. We may simply say that G' *contains* G.

We focus on a graph database or repository containing a set of small- or medium-sized data graphs. Given such a database D, we assign a unique *index* (*i.e.,* id) to each data graph. A data graph G with index i is denoted as G_i.

Maximum (Connected) Common Subgraph. Graph similarity can be assessed using *feature-based* or *structure-based* measures. The former cannot capture the global structural information of graphs. Hence, structure-based measures such as those based on *maximum common subgraph* (MCS) [7] and *maximum connected common subgraph* (MCCS) [36] are considered superior alternatives. Given two graphs G_1 and G_2, G is a *common subgraph* of G_1 and G_2 if $G \subseteq G_1$, $G \subseteq G_2$. G is an MCS if there exists no other common subgraph of G_1 and G_2 larger than G.

Since MCS does not require a common subgraph to be connected, it is possible to have poor similarity match where vertices in one graph are mapped to those in another that are positioned very distant from each other [36]. The *maximum connected common subgraph* (MCCS) addresses this by imposing an additional constraint that an MCS must be connected.

Given two graphs G_1 and G_2, let G_{MCS} (resp. G_{MCCS}) be the MCS (resp. MCCS) of G_1 and G_2. The *maximum common subgraph similarity* (resp. *maximum connected common subgraph similarity*) between G_1 and G_2 is defined as $\omega_{\text{MCS}}(G_1, G_2) = \frac{|G_{\text{MCS}}|}{Min(|G_1|, |G_2|)}$ (resp. $\omega_{\text{MCCS}}(G_1, G_2) = \frac{|G_{\text{MCCS}}|}{Min(|G_1|, |G_2|)}$) where $Min(.)$ is the minimum operator. It is known that MCS and MCCS computation are both NP-complete [36].

Closure Graph. A *closure graph* is a generalized graph generated by performing a *union* on the structures of a set of graphs [19]. We first review two related concepts, namely, *graph extension* and *graph mapping*. *Graph extension* allows "integration" of graphs of varying sizes into a single graph referred to as *extended graph* (denoted by $G^* = (V^*, E^*)$) by inserting dummy vertices or edges with a special label ε such that every vertex and edge is represented in G^*. For instance, consider the set of graphs in Figure 4(a). G_1 is extended to G_1^* in Figure 4(b) by adding a dummy vertex ε and an edge that connects vertex C with it. Given two extended graphs G_1^* and G_2^*, a generalized graph (Figure 4(c)) can be obtained by *mapping* their vertices and edges using the approach in [19]. Formally, given two extended graphs $G_1^* = (V_1^*, E_1^*)$ and $G_2^* = (V_2^*, E_2^*)$, the *graph mapping* between G_1^* and G_2^* is given as a bijection function $\phi : G_1^* \rightarrow G_2^*$ where (i) $\forall v \in V_1^*, \phi(v) \in V_2^*$ and at least one of v and $\phi(v)$ is not dummy, (ii) $l_1(v) = l_2(\phi(v))$ if both v and $\phi(v)$ are not dummy, and (iii) $\forall e = (v_1, v_2) \in E_1^*, \phi(e) = (\phi(v_1), \phi(v_2)) \in E_2^*$ and at least one of e or $\phi(e)$ is not dummy.

Given two extended graphs G_1^* and G_2^* and a mapping ϕ between them, a *vertex* and an *edge closure* can be obtained by performing an element-wise union of the attribute values of each vertex and each edge in the two graphs, respectively. Then the *closure graph* of G_1^* and G_2^* is a labelled graph $G_c = (V_c, E_c)$ where V_c is the vertex closure of V_1^* and V_2^* and E_c is the edge closure of E_1^* and E_2^*. The labels take the form of $A\{i_1, \cdots, i_n\}$ for vertices and $\{j_1, \cdots, j_m\}$ for edges where A is a vertex name, $\{i_1, \cdots, i_n\}$ is a set of indices of graphs containing vertex A, and $\{j_1, \cdots, j_m\}$ is a set of indices of graphs containing edges of connected vertex pairs. For example, in Figure 4(c), vertex C{1} in G_1^* is mapped to vertex C{2} in G_2^*. In the closure graph (Figure 4(d)), these vertices are replaced by a vertex C{1,2}. Note that these closures may contain attribute values ε corresponding to a dummy vertex or edge. We remove dummy labels from closure graphs. For example, in Figure 4(c), vertex $\varepsilon\{\varepsilon\}$ is mapped to vertex N{2}. In the closure graph (Figure 4(d)), these vertices are replaced by a vertex N{2}.

3 CANNED PATTERN SELECTION PROBLEM

In this section, we formally define the *canned pattern selection* problem and present an overview of the CATAPULT framework to address it. We begin by highlighting the desired characteristics of canned patterns.

3.1 Desired Characteristics

Intuitively, the goal of canned patterns $\mathcal{P}$ in a GUI $\mathbb{I}$ is to aid users to visually formulate queries quickly. An ideal set of canned patterns efficiently facilitates construction of a wide variety of query graphs. However, selection of $\mathcal{P}$ is challenging not only because it is hard to find an optimal solution (see Theorem 3.2) but also due to the limited availability of space in $\mathbb{I}$. It is impractical to display a large number of canned patterns as not only it will make $\mathbb{I}$ overly complex but also a user will need to search a long list of these patterns in order to determine which ones are best suited for her query. Hence, the number of canned patterns should be sufficiently small and should satisfy the following desirable characteristics.

(1) **High coverage**. A pattern $p \in \mathcal{P}$ *covers* a data graph $G \in D$ if G contains a subgraph s that is isomorphic to p. The pattern set $\mathcal{P}$ should ideally cover as many data graphs in D as possible. This ensures that a large number of subgraph queries on D can be visually constructed by utilizing $\mathcal{P}$. Furthermore, due to the small size of $\mathcal{P}$, not all vertex/edge labels of D can be made available in the patterns[4]. Nevertheless, it is desirable for $\mathcal{P}$ to have high *label coverage* of D as well.
(2) **High diversity**. As remarked in Section 1, high coverage of patterns is insufficient to support fast visual query formulation. Let Q be a subgraph query constructed by a user using $\mathcal{P}$. If $\mathcal{P}$ contains very similar patterns, then only a very few of them can be utilized to formulate Q. Given that the display space in $\mathbb{I}$ is limited, clearly $\mathcal{P}$ makes suboptimal use of this space as highly similar patterns can be replaced with structurally diverse ones serving a larger variety of queries.
(3) **Low cognitive load**. *Cognitive load* refers to the memory demand or mental effort required to perform a given task [20]. As remarked in Section 1, a large and complex pattern may demand substantial cognitive effort from a user to interpret [20, 25] and to determine if it can be used for constructing a particular query graph. Hence, it is desirable for canned patterns to impose low cognitive load on a user.

These features will guide us in our canned pattern selection process. We re-state two key differences from canned pattern selection in traditional visual graph query interfaces. First, canned patterns are manually selected in traditional interfaces. Second, they are not selected by maximizing coverage and diversity and minimizing cognitive load.

3.2 Problem Definition

Intuitively, given a graph database D, a visual graph query interface $\mathbb{I}$, and a *pattern budget b*, the canned pattern selection problem aims to select a set of patterns $\mathcal{P}$ satisfying b from D to display on $\mathbb{I}$ by maximizing *coverage* and *diversity* and minimizing *cognitive load* of $\mathcal{P}$. We consider two types of coverage, namely *subgraph coverage* and *label coverage* for patterns and vertex/edge labels, respectively. The *subgraph coverage* of a pattern $p \in \mathcal{P}$ is defined as $scov(p, D) = \frac{|\mathcal{G}_p|}{|D|}$ where $\mathcal{G}_p \subseteq D$ is a set of data graphs containing p. Consequently, the subgraph coverage of a set of canned patterns $\mathcal{P}$ is given as $scov(\mathcal{P}, D) = \frac{|\bigcup_{p \in \mathcal{P}} \mathcal{G}_p|}{|D|}$. On the other hand, labelled data graphs may contain a variety of different vertex/edge labels[5]. Let $L(e, D)$ be a set of graphs in D containing edges having same label as e and $L(E_{\mathcal{P}}, D) = \bigcup_{e_i \in E_{\mathcal{P}}} L(e_i, D)$. Then the *label coverage* of $\mathcal{P}$ w.r.t D is given as $lcov(\mathcal{P}, D) = \frac{|L(E_{\mathcal{P}}, D)|}{|D|}$.

We measure the *diversity* of a pattern by utilizing graph edit distance (GED) [32]. That is, given a candidate pattern p and a set of canned patterns $\mathcal{P}$, the *diversity* of p w.r.t $\mathcal{P}$ is defined as $div(p, \mathcal{P} \setminus p) = min\{GED(p, p_i)\}$ where $p_i \in \mathcal{P} \setminus p$ and $GED(.)$ is the graph edit distance operator.

Lastly, given a pattern $p = (V_p, E_p)$, the *cognitive load* of p is defined as $cog(p) = |E_p| \times \rho_p$ where $\rho_p = 2\frac{|E_p|}{|V_p|(|V_p|-1)}$ is the density of p. Recall that the cognitive load increases with denser graphs as users tend to spend more time to identify relationships between different vertices [20]. Kobourov et al. [25] recently demonstrated that increase in edge crossing hamper graph interpretation-related tasks in terms of both time taken and accuracy. These results inspired us to measure cognitive load using graph density as it provides an estimate on the degree of edge crossings in a pattern. Specifically, a large dense pattern is more likely to contain a larger number of edge crossings than a small sparse pattern. Experimental justification of this density-based definition is given in Appendix C (*Exp 10*).

Definition 3.1. Given a graph database D, a graph query interface $\mathbb{I}$, and a pattern budget $b = (\eta_{min}, \eta_{max}, \gamma)$ where η_{min} (resp. η_{max}) is the minimum (resp. maximum) size of a pattern and γ is the number of patterns to be displayed on $\mathbb{I}$, the goal of ***canned pattern selection problem*** *is to find a set of canned patterns $\mathcal{P}$ from D that satisfies the followings:*

$$\begin{array}{ll} max\ scov(\mathcal{P}, D) & max\ lcov(\mathcal{P}, D) \\ max\ div(p, \mathcal{P} \setminus p) & min\ cog(p) \\ s.t.\ p \in \mathcal{P}, p \subseteq G, G \in D & \end{array}$$

where $|\mathcal{P}| = \gamma$ and $\frac{\gamma}{\eta_{max} - \eta_{min} + 1}$ is the maximum number of patterns for each k-sized pattern, $k \in [\eta_{min}, \eta_{max}]$, $\eta_{min} > 2$.

Remark. Observe that our canned pattern selection problem is defined as a mixture of minimization and maximization of objective functions [12]. Also, it aims to find patterns of size greater than 2. Patterns of smaller size (*e.g.,* labelled

[4]In a GUI, there can be a separate panel that lists all unique labels of vertices and edges in a data source.

[5]In graphs where only vertices are labelled, an edge label can be considered as concatenation of labels of the end vertices.

Algorithm 1 Catapult.

```
Require: Graph database $D$, pattern budget $b = (\eta_{min}, \eta_{max}, \gamma)$;
Ensure: Canned pattern set $\mathcal{P}$;
1: $C_{coarse} \leftarrow CoarseClustering(D)$ /* Algorithm 2*/
2: $C_{fine} \leftarrow FineClustering(C_{coarse})$ /* Algorithm 3*/
3: $\mathbb{S} \leftarrow ClusterSummaryGraphSet(C_{fine})$
4: ELW $\leftarrow GetEdgeLabelWeight(D)$
5: CW $\leftarrow GetGraphClusterWeights(C_{fine})$
6: $\mathcal{P} \leftarrow FindCannedPatternSet($ELW, CW,$\mathbb{S}$,$b)$ /* Algorithm 4*/
```

edge[6], 2-path) are provided as *basic patterns* in our GUI [23] and is computed after the generation of canned patterns. Specifically, in our GUI we select top-*m* basic patterns based on their support and is detailed in [23].

Theorem 3.2. *The canned pattern selection problem is NP-hard.*

3.3 Overview of CATAPULT

It is prohibitively expensive to iteratively evaluate subgraphs in each data graph $G \in D$ w.r.t coverage, diversity, and cognitive load in order to compute $\mathcal{P}$. At first glance, it may seem that frequent subgraphs [31] can be utilized as canned patterns as these subgraphs may have high coverage. However, subgraph queries are not necessarily frequent in nature as users may frequently pose infrequent subgraph queries [6]. For example, in Example 1.1 patterns P_1 and P_2 are not frequent subgraphs in the underlying database.

Algorithm 1 outlines the Catapult approach to tackle this problem. It first *clusters D* based on their *feature* and *topological* similarities (Lines 1-2) and then constructs a *cluster summary graph* (CSG) $S \in \mathbb{S}$ for the data graphs in each cluster by utilizing the notion of closure graph [19] (Line 3). Next, it maintains two types of weights related to each cluster and labelled edges, namely, *cluster weight* (CW) and *edge label weight* (ELW) (Lines 4-5). The former is a ratio of the number of graphs in a cluster to that in the database and is a measure of the importance of a cluster and its CSG. A pattern that is derived from a CSG with large cluster weight is more likely to achieve a higher coverage compared to one derived from a CSG with small cluster weight. The latter measures the global occurrence of a labelled edge in the database. Lastly, Catapult automatically selects canned patterns $\mathcal{P}$ (Line 6) from CSGs within pattern budget b by first generating *weighted* CSGs using ELW, and then selecting patterns from them by considering subgraph coverage (based on CW), label coverage, diversity, and cognitive load. Intuitively, these features are utilized as follows.

Subgraph Coverage. Coverage of a canned pattern set $\mathcal{P}'$ increases if for every subsequent pattern p_i added to $\mathcal{P}'$, it is derived from graphs in D that are not yet covered by any existing pattern $p_j \in \mathcal{P}'$. For example, consider $\mathcal{P}'$ and a graph database D partitioned into mutually exclusive *graph clusters* $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ where $\forall C_i \in C$, $|C_i| = 10$ (*i.e.*, every cluster contains 10 graphs), and $D = C$. Then the cluster weight (cw) is $cw_i = \frac{|C_i|}{|D|} = \frac{10}{60}$ (identical for all 6 clusters). Suppose $\mathcal{P}'$ covers C_1, C_2, C_4 in D. Consider two candidate patterns p'_1 and p'_2 covering C_1, C_2, C_3 and C_2, C_3, C_5, respectively. Catapult preferentially selects p'_2 to be added to $\mathcal{P}'$ since it would increase the subgraph coverage to 5 clusters[7] as opposed to 4 if p'_1 is chosen.

Label Coverage. Intuitively, we add patterns in $\mathcal{P}$ that result in a higher label coverage. For example, let the set of unique edge labels of $\mathcal{P}'$ and D be $\{(0,1),(0,2),(1,3)\}$ and $\{(0,1),(0,2),(0,3),(0,4),(1,2),(1,3),(2,3),(3,4)\}$, respectively, where $(l(v_i), l(v_j))$ are labels of (v_i, v_j). Consider two candidate patterns p'_1 and p'_2 having unique edge labels $\{(0,1),(0,3),(0,4),(1,3),(1,4)\}$ and $\{(0,1),(0,2),(0,3),(1,2),(1,3)\}$, respectively. Then Catapult preferentially selects pattern p'_1 as $\mathcal{P}'$ covers 6 unique labels as opposed to 5 if p'_2 is selected.

Diversity. Given a pattern set $\mathcal{P}'$ containing three patterns p_1, p_2, p_3 and two candidate patterns p'_1 and p'_2, let $div(\mathcal{P}', p'_1) = min\{GED(p'_1, p_1), GED(p'_1, p_2), GED(p'_1, p_3)\} = 5$ and $div(\mathcal{P}', p'_2) = min\{GED(p'_2, p_1), GED(p'_2, p_2), GED(p'_2, p_3)\} = 7$. Then, p'_2 has a greater pattern set diversity compared to p'_1 and is preferentially selected.

Cognitive Load. Given two candidate patterns p_1 and p_2, p_1 is preferred over p_2 if p_1 has lower cognitive load than p_2.

Lastly, to tackle very large graph databases, we extend the framework by sampling data graphs judiciously and then generate canned patterns from it (Section 4.3).

Remark. Recall from Section 1, patterns with high coverage do not necessarily support efficient visual query formulation. Hence, we reemphasize that the goal of Catapult is to automatically select canned patterns having low cognitive load and are sufficiently diverse enough to expedite formulation of a variety of visual subgraph queries by a variety of users. Since users may formulate both frequent and infrequent queries in practice, Catapult does not make any restrictive assumption on the type of queries it supports or their results size. Hence, it selects patterns (frequent and infrequent) that can frequently assist in formulating queries.

Furthermore, Catapult is query log-oblivious as such log data may be unavailable especially in "cold start" cases. For instance, query log may be unavailable for some remote public data source (*e.g.*, AIDS) and cannot be exploited when a user downloads it to formulate queries over it. Additionally, there has to be a sufficient volume of such log data to be effective in canned pattern selection. Nevertheless, our canned pattern selection step (Line 6) can be extended to incorporate frequency of patterns in past subgraph queries.

[6]A labelled edge is an edge with labelled vertices.

[7]Since all clusters contain the same number of graphs, we can count number of clusters instead of number of graphs.

Algorithm 2 *CoarseClustering.*

```
Require: Graph database $D$;
Ensure: A set of graph clusters $C$;
1:  $\mathcal{T}_{all} \leftarrow GenerateFrequentSubtrees(D)$;
2:  $\mathcal{T}_{sel} \leftarrow SelectFrequentSubtrees(\mathcal{T}_{all})$;
3:  for $G_i \in D$ do
4:      Initialize $R_i$ as a $|\mathcal{T}_{sel}|$-dimensional zero vector;
5:      for Subtree $T_j \in \mathcal{T}_{sel}$ do
6:          if $G_i$ contains $T_j$ then
7:              Update $j^{th}$ position of vector $R_i$ to 1;
8:          end if
9:      end for
10: end for
11: $C \leftarrow Clustering(R, D)$ /* $R = \{R_1, \cdots, R_{|D|}\}$*/;
```

4 CLUSTER SUMMARY GRAPH (CSG) GENERATION

A large collection of small- or medium-sized data graphs is likely to contain groups of graphs having similar topology. These groups can be obtained via clustering and each group can be represented by a *cluster summary graph* (CSG). Subsequently, we aim to design techniques that enable us to select relevant canned patterns from these CSGs instead of directly computing them from D, which is computationally untenable. Here we describe the CSG generation process. In the next section, we shall elaborate on how canned patterns can be generated from these CSGs.

4.1 Small Graph Clustering

We aim to partition D into a set of *graph clusters* $C = \{C_1, C_2, \ldots, C_d\}$, where $C_i \subseteq D$, $C_i \cap C_j = \emptyset\ \forall i \neq j$, and it maximizes a *clustering property* objective function $f : C \longrightarrow \mathbb{R}$, *i.e.,* find any C in $argmax_C f(C) = \{C|\forall C' : f(C') \leq f(C)\}$. Unfortunately, majority of graph clustering approaches focus on identifying "related" vertices in a *single* large graph [34]. There is scant research on clustering a set of small- or medium-sized graphs (*i.e.,* small graph clustering) [18, 35] and they can be categorized into two classes: *feature vector-based* and *graph structure-based.* The former uses graph properties or subgraph occurrences as a feature vector in a standard clustering algorithm. In contrast, the latter uses graph structures such as MCS or MCCS directly resulting in clusters that are more intuitive and interpretable. However, these techniques are expensive. Hence, in CATAPULT we explore a *hybrid* technique that integrates these two approaches to achieve high quality clusters in reasonable time.

Given D, CATAPULT first assigns each data graph $G \in D$ to an appropriate graph cluster C based on the distance between the *frequent subtree*[8]-based feature vector of G and that of the feature vector representative of C (*i.e.,* clustering property) where $|C| = k$ (*coarse clustering*, Line 1 of Algorithm 1). Then, if $|C|$ is larger than a threshold N, C is further decomposed into smaller clusters where each sub-cluster has size less than N and intra-cluster graphs have smaller topological distances measured using MCCS (*i.e.,* clustering property) compared to inter-cluster data graphs (*fine clustering*, Line 2). Observe that coarse clustering is a feature vector-based approach whereas fine clustering is based on graph topology. Also note that the reason we further decompose "larger" clusters using fine clustering is because it reduces the size of CSGs and their generation cost in the subsequent step by operating on a smaller collection of similar data graphs. We now elaborate on the coarse and fine clustering.

[8]Compared to frequent graphs, frequent subtrees describe crucial topology of graphs but demand lower computational cost.

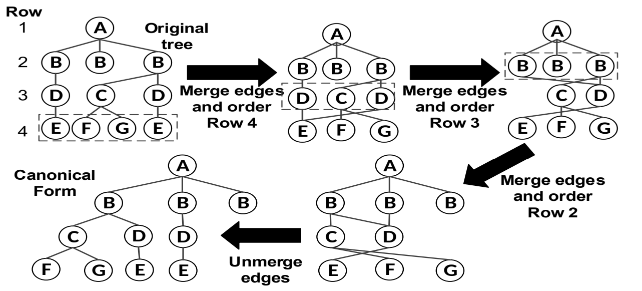
Row
1
2
3
4
Original tree
Merge edges and order Row 4
Merge edges and order Row 3
Merge edges and order Row 2
Unmerge edges
Canonical Form
A
B
C
D
E
F
G

Figure 5: Canonical form.

Coarse clustering. Coarse clustering (Algorithm 2) leverages *frequent subtrees* as feature vectors, which are connected acyclic subgraphs with support greater than or equal to a threshold value. In CATAPULT, frequent subtrees are generated (Line 1) using the approach in [10] and are represented as canonical strings in two steps: (1) canonical tree generation via normalization and (2) conversion of the tree to the canonical string. Normalization of a labeled rooted tree is a bottom-up procedure based on the tree isomorphism algorithm in [1]. Given the original tree, it is performed level-by-level bottom up, using orders among subtrees at each level until the canonical form is obtained. An example of normalization is given in Figure 5. Note that subtrees that are "equal" (*e.g.*, branch $\{B, D, E\}$) are combined in the intermediate steps. The canonical string is obtained by scanning the canonical tree top-down level-by-level in a breadth-first manner. Symbols \$ and # are used to partition families of siblings and the end of the canonical string, respectively. Hence, the canonical string of the tree in Figure 5 is $A\$1B1B1B\$1C1D\$1D\$1F1G\$1E\$1E\#$, assuming that all edges have label of 1.

Observe that a set of frequent subtrees may contain subtrees that are highly similar to others. Hence, the selection of frequent subtree set (Line 2) can be further optimized using maximization of the *uncapacitated facility location function* [21] (Appendix B). Here, frequent subtrees are facilities and the facility cost is the similarity of a subtree to other subtrees in the set. This function is a monotone submodular function and can achieve a near-optimal solution by applying greedy search where the selected subtrees are able to achieve at least $1 - \frac{1}{e} \approx 63\%$ discriminative power for clustering [17]. We model the problem of selecting a greedy

Algorithm 3 *FineClustering.*

```
Require: A set of graph clusters $C$;
Ensure: A set of graph clusters $C$ with size $\leq N$;
 1: Initialize $N$ with the default maximum cluster size;
 2: $C_{large} \leftarrow GetLargeClusters(C, N)$; /*Contains $C_i$ where $|C_i| > N$*/
 3: $C \leftarrow C \backslash C_{large}$;
 4: $C_{new} \leftarrow \phi$;
 5: while $|C_{large}| > 0$ do
 6:     $C_{first} \leftarrow RemoveFirstCluster(C_{large})$;
 7:     $Seed1 \leftarrow SelectRandomGraph(C_{first})$;
 8:     $C' \leftarrow InsertIntoSet(C', Seed1)$;
 9:     for Graph $G \in C_{first} \backslash \{Seed1\}$ do
10:         $\omega_G \leftarrow GetSimilarity(G, Seed1)$;
11:     end for
12:     $Seed2 \leftarrow SelectDissimilarGraph(C_{first} \backslash \{Seed1\}, \omega)$;
13:     $C^{\dagger} \leftarrow InsertIntoSet(C^{\dagger}, Seed2)$;
14:     for $G \in C_{first} \backslash \{Seed1, Seed2\}$ do
15:         $\omega'_G \leftarrow GetSimilarity(G, Seed2)$;
16:         if $\omega_G > \omega'_G$ then
17:             $C' \leftarrow InsertIntoSet(C', G)$;
18:         else
19:             $C^{\dagger} \leftarrow InsertIntoSet(C^{\dagger}, G)$;
20:         end if
21:     end for
22:     $C_{large}, C_{new} \leftarrow ClusterUpdate(C_{large}, C_{new}, C', N)$;
23:     $C_{large}, C_{new} \leftarrow ClusterUpdate(C_{large}, C_{new}, C^{\dagger}, N)$;
24: end while
25: $C \leftarrow C \bigcup C_{new}$;
```

set of frequent subtree as features using the minimization of the dissimilarity between subtrees. This problem can be recast as a maximization of *similarity* between subtrees as follows. Given two subtrees i and j represented as canonical strings, the *subtree similarity* of i and j is defined as $\sigma_{subtree}(i,j) = \frac{|lcs(i,j)|}{max(|i|,|j|)}$ where $|i|$ is the size of i, $lcs(i,j)$ is the longest common subtree between i and j and $max(.)$ is the maximum operator. Formally, the submodular function is defined as: $q(\mathcal{T}_{sel}) = \sum_{i \in \mathcal{T}_{all}} \max_{j \in \mathcal{T}_{sel}}(\sigma_{subtree}(i,j))$ where $\mathcal{T}_{all}$ is the set of all frequent subtrees and $\mathcal{T}_{sel}$ is the set of near-optimal frequent subtrees. Next, CATAPULT iterates through each graph $G_i \in D$ to determine its feature vector. For clarity, the feature vector is a $|\mathcal{T}_{sel}|$-dimensional vector. The j^{th} position of the vector is one if G_i contains the subtree $\mathcal{T}_j \in \mathcal{T}_{sel}$, and zero otherwise (Lines 3-10). Finally, clustering (Line 11) is performed using $\mathcal{T}_{sel}$ as feature vectors. In particular, CATAPULT uses the k-means clustering with k seeds selected using the k-means++ algorithm [4]. We set k as $\frac{|D|}{N}$ where N is the maximum cluster size.

Fine clustering. In fine clustering (Algorithm 3), CATAPULT further breaks down large clusters into smaller ones. It organizes a given set of graphs C_{first} into two new clusters (C' and $C^{\dagger}$) according to the MCCS similarity of the graph $G \in C_{first}$ and seed graphs *Seed*1 and *Seed*2 corresponding to C' and $C^{\dagger}$, respectively. The first seed graph *Seed*1 is selected randomly from C_{large} (Line 7) whereas the second one $Seed2 \in C_{large}$ is selected such that it is most dissimilar to *Seed*1 (Lines 9 to 12). After the clustering process, CATAPULT performs an update of C_{large} and C_{new} by checking the size of the newly-generated clusters. A new cluster will be inserted into C_{large} if its size is larger than N. Otherwise, it will be inserted into C_{new} (Lines 22 to 23). We adopt the McGregor algorithm [27] to compute MCCS.

Example 4.1. Given a graph dataset $D = \{G_1, \cdots, G_{17}\}$, let $N = 6$ and the number of k-means clusters be $K = 3$. Suppose the clusters are $C_1 = \{G_1, G_2, G_3, G_6, G_9, G_{12}\}$, $C_2 = \{G_4, G_7, G_8, G_{13}, G_{14}, G_{15}, G_{17}\}$ and $C_3 = \{G_5, G_{10}, G_{11}, G_{16}\}$ after the coarse clustering phase. Then, the size of C_2 exceeds N and is subjected to fine clustering. A seed (*e.g.*, G_8) is randomly selected from C_2 and is inserted into the first new cluster C'. Then, MCCS similarities are computed between G_8 and remaining graphs in C_2. The graph having the greatest MCCS dissimilarity is selected as the second seed (*e.g.*, G_{15}) and inserted into the second new cluster $C^{\dagger}$. For the remaining graphs (*i.e.*, $C_{2R} = C_2 \setminus \{G_8, G_{15}\}$), MCCS similarities are computed between each graph $G_i \in C_{2R}$ and G_8 (denoted as ω_{G_i}), and between G_i and G_{15} (denoted as ω'_{G_i}). Comparison is made between ω_{G_i} and ω'_{G_i}. A larger ω_{G_i} implies that G_i is more similar to G_8 compared to G_{15}, and G_i is inserted into the first new cluster. Fine clustering splits C_2 into two new clusters $C'_2 = \{G_7, G_8, G_{13}\}$ and $C^{\dagger}_2 = \{G_4, G_{14}, G_{15}, G_{17}\}$. Hence, at the end of the small graph clustering phase, four clusters, namely, C_1, C'_2, $C^{\dagger}_2$ and C_3, are generated. ∎

Remark. The worst case time complexity of small graph clustering is exponential in cost due to the k-means algorithm [3] (See Appendix A). Note that the CATAPULT framework is orthogonal to the choice of a feature vector-based clustering approach as k-means can be replaced with an alternative clustering algorithm. Furthermore, the small graph clustering step is a one-time cost and is only invoked when D is a new dataset. Hence, such a tradeoff is appropriate.

LEMMA 4.2. *Small graph clustering achieves* $\frac{1}{2}+(\alpha-\frac{1}{2})min\_fr$*-approximation of optimum clustering where* min_fr *is the support of frequent subtree and* α *is the probability that correct clustering occurs given that the* MCCS *of a pair of graphs (in the same cluster under optimum clustering) contains frequent subtrees of* D.

4.2 Generation of CSGs

Once the set of graph clusters C are generated, CATAPULT summarizes *each* cluster $C_i \in C$ into a closure graph (recall from Section 2). We refer to it as *cluster summary graph* (CSG). In particular, it iterates through each cluster $C_i \in C$ and performs graph closure [19] by considering a pair of data graphs at a time. Briefly, graph extension of a pair of data graphs is mapped and the closure graph is found by performing edge closure on the extended graph[9]. The CSG S for a cluster C_i is obtained when all data graphs in the cluster have been integrated into the closure graph.

[9]CATAPULT skips the vertex closure step since edge labels are needed subsequently.

Lemma 4.3. *The time and space complexities of the* csgs *generation process are* $O(|D||V_{max}|d^2 log(|V_{max}|))$ *and* $O(|D|(|E_{max}|+|V_{max}|))$*, respectively, where d is the maximum degree of vertices and* $G_{max} = (V_{max}, E_{max})$ *is the largest graph in D.*

4.3 Handling Larger Graph Databases

Small graph clustering can be computationally expensive for large *D*. To alleviate this challenge, Catapult follows a *two-level sampling* approach (*eager sampling* and *lazy sampling*) as depicted in Figure 3. Intuitively, the *eager sampling* is performed *prior* to the small graph clustering phase and the *lazy sampling* is performed *after* the coarse clustering phase. As we shall see in Section 6, this sampling approach achieves a good balance between the quality of canned patterns and the runtime performance of Catapult.

Eager sampling. *Eager sampling* refers to random sampling from *D*. First, it generates a random sample of data graphs from *D*. Given an error bound ϵ and a maximum probability ρ for the error that exceeds ϵ, the size of the random sample ($|S_{eager}|$) is determined by $|S_{eager}| \geq \frac{1}{2\epsilon^2} ln \frac{2}{\rho}$ [38]. For example, given *D* and sampling parameters $\rho = 0.01$ and $\epsilon = 0.02$, $|S_{eager}| = \frac{1}{2(0.02)^2} ln \frac{2}{0.01} = 6623$. Observe that $|S_{eager}|$ is independent of $|D|$. Then, the sample S_{eager} is used to find a frequent subtree set (recall from the *coarse clustering* phase). For a subtree *t*, the probability that the error $e(t, S_{eager}) > \epsilon$ is at most ρ. Note that $e(t, S_{eager}) = |fr(t) - fr(t, S_{eager})|$ where $fr(t)$ and $fr(t, S_{eager})$ are the frequencies of *t* in *D* and S_{eager}, respectively. Hence, by setting $low\_fr < min\_fr$, the potential frequent subtrees found with lower support low_fr for the sample are less likely to miss any frequent subtrees found with support of min_fr for the original dataset. Catapult performs counting on this potential frequent subtree set using the original support threshold (*i.e.,* min_fr) to retrieve the final set of frequent subtrees. The frequent subtree set is then used as feature vectors in the coarse clustering phase.

Lemma 4.4. *Given a frequent subtree set X, a random sample* S_{eager}*, a probability parameter* φ*, the probability that* $x \in X$ *is missed is at most* φ *when* $low\_fr < min\_fr - \sqrt{\frac{1}{2|S_{eager}|} ln \frac{1}{\varphi}}$ *where* low_fr *and* min_fr *are the lower support threshold and the original support threshold, respectively [38]. Note that* min_fr *is a user-specified support value of frequent subtree set and* low_fr *is a lower support threshold for the frequent subtree set due to sampling.*

Lazy sampling. After coarse clustering, some clusters may still be too large for efficient processing. Catapult performs stratified random sampling of large clusters to further reduce their sizes (referred to as *lazy sampling*). For example, suppose after coarse clustering of a dataset of 50K data graphs, cluster C_1 contains 1000 data graphs. Let the

Algorithm 4 *FindCannedPatternSet*.

```
Require: Edge label weight ELW, cluster weight CW, a set of CSGS $\mathbb{S}$, pattern budget
        $b = (\eta_{min}, \eta_{max}, \gamma)$;
Ensure: A set of canned patterns $\mathcal{P}$;
 1: $\mathcal{P} \leftarrow \phi$;
 2: $\mathbb{S} \leftarrow GetWeightedGraph(\mathbb{S}$, ELW);
 3: while $|\mathcal{P}| < \gamma$ do
 4:     $\mathcal{P}_c \leftarrow \phi$;
 5:     $\Pi \leftarrow GetPatternSizeRange(b, \mathcal{P})$;
 6:     for $S \in \mathbb{S}$ do
 7:         for Pattern size $\eta \in \Pi$ do
 8:             $\mathcal{L} \leftarrow \phi$;
 9:             for iteration $i$=0 to $x$ /*$x$ =max no. of random walks*/ do
10:                 PCP $\leftarrow GeneratePCP(G, \eta)$;
11:                 $\mathcal{L} \leftarrow \mathcal{L} \bigcup \{$PCP$\}$;
12:             end for
13:             FCP $\leftarrow GenerateFCP(\mathcal{L})$;
14:             $\mathcal{P}_c \leftarrow \mathcal{P}_c \bigcup \{$FCP$\}$;
15:         end for
16:     end for
17:     $s \leftarrow GetPatternScore(\mathcal{P}_c, \mathcal{P}$, CW);
18:     $p_{best} \leftarrow GetBestPattern(s, \mathcal{P}_c)$
19:     $\mathcal{P} \leftarrow \mathcal{P} \bigcup \{p_{best}\}$;
20:     CW $\leftarrow UpdateClusterWeight($CW$, p_{best}, \mathbb{S})$;
21:     ELW $\leftarrow UpdateEdgeLabelWeight($ELW$, p_{best})$;
22: end while
```

sampling parameters be $p = 0.5$, $Z_{\frac{\alpha}{2}} = Z_{\frac{0.95}{2}}$ and $e = 0.03$. Then $|S_{lazy}| = (\frac{1.65^2 \times 0.5^2}{0.03^2}/50000) \times 1000 = 15.13$ (Lemma 4.5) and C_1 can be further reduced by taking a sample of 15 graphs. Note that fine clustering still needs to be performed if $|S_{lazy(C)}| > N$.

Lemma 4.5. *Given a set of data graphs D containing* $|C|$ *clusters, the size of a random sample set* $S_{lazy(C)}$ *required to estimate a cluster* $C \in C$ *is defined as*

$$|S_{lazy(C)}| = \frac{|S_{sample}|}{\sum_{C_i \in C} |C_i|} \times |C| \quad (1)$$

where S_{sample} *is the sample for D. Here* $|S_{sample}| = \frac{Z^2 pq}{e^2}$ *where* Z^2 *is the abscissa of the normal curve that cuts off an area* α *at the tails (1-*α *is the desired confidence level, e.g., 95%), e is the desired level of precision, p is the estimated proportion of a graph being sampled in D, and* $q = 1 - p$.

5 SELECTION OF CANNED PATTERNS

Given a set of csgs $\mathbb{S}$, Catapult follows a greedy iterative approach for selecting canned patterns for a gui. In each iteration, *candidate patterns* are generated from each csg $S \in \mathbb{S}$ and the "best" pattern for that iteration is added to the partial canned pattern set $\mathcal{P}'$. *Weights* related to coverage are assigned to the csgs to ensure that in each subsequent iterations, candidate patterns are derived from csgs that are not yet covered by $\mathcal{P}'$. Catapult performs random walks on these weighted csgs and leverages on the statistics obtained from the walks to propose a candidate canned pattern (*final candidate pattern*) for each size in the range $[\eta_{min} - \eta_{max}]$ (*i.e.,* pattern budget *b*). A *pattern score* based on coverage, diversity, and cognitive load is computed for each candidate pattern and utilized to select the next best pattern to be added into $\mathcal{P}'$. Weights of the csgs are then updated based on the selected pattern. These steps are repeated until either the required number of canned patterns are discovered or when no new pattern can be found. We now describe the algorithm (Algorithm 4) in detail.

Weighted csg construction (Line 2). The csg S of a cluster C is a summarized representation of data graphs contained in C. Each edge e in S is assigned a *weight* w_e based on its label coverage in the dataset (*i.e.,* global occurrence) and in the cluster (*i.e.,* local occurrence) as follows: $w_e = lcov(e, D) \times lcov(e, C)$ where $lcov(e, X) = \frac{|L(e,X)|}{|X|}$.

Weighted random walk for candidate pattern generation (Lines 6 to 16). We adopt a random walk-based approach for candidate generation as each random walk starts afresh in each iteration and has the potential to cover different regions of the csgs, thus producing diverse candidate patterns. Given a weighted csg S, Catapult performs random walk to generate a variety of *potential candidate patterns* (pcp) from which a *final candidate pattern* (fcp) is derived. These pcps collectively form a *candidate library* $\mathcal{L}$. We elaborate on the generation of pcp and fcp.

Each random walk to generate a pcp starts with a *seed edge* (*i.e.*, edge with largest weight). In every iteration, the pcp is "grown" by adding an adjacent edge until the required number of edges is achieved or when no more edges can be added. An adjacent edge is selected as follows [16]: (a) Find all adjacent edges (referred to as *candidate adjacent edges* (cae)) of the partial pcp. (b) Multiply all cae with the least common multiplier (lcm) of denominators of weights of the cae. Hence, these cae now have integer weights. (c) Replace each cae (u, v) with integer weight k by k cae (u, v) of weight 1. (d) Randomly select a cae. At the end of each random walk, a pcp is added to the candidate library $\mathcal{L}$.

Generation of the fcp starts with the first edge (*i.e.*, most frequent edge in $\mathcal{L}$). Similar to the pcp, the fcp is "grown" an edge at a time. In order to ensure that the fcp is a connected subgraph, the next added edge selected is the most frequent edge in $\mathcal{L}$ that is also connected to the previous added edge.

Pattern score computation (Line 17). The *pattern score*, which selects the best candidate (*i.e.*, candidate with the highest score), is computed by combining $scov()$, $lcov(\cdot)$, $div(\cdot)$, and $cog(\cdot)$. Observe that the *scov* computation is extremely expensive when $|D|$ is large. Hence, we estimate *scov* in terms of the cluster coverage *ccov*. That is, $scov(\mathcal{P}, D) \simeq ccov(\mathcal{P}, cw, C)$ where C is a set of clusters of D and cw is the cluster weight vector such that $cw_i = \frac{|C_i|}{|D|}$ and $ccov(\mathcal{P}, cw, C) = \sum_{i \in C} cw_i \times I_i$ where $I_i = 1$ if the csg of C_i contains a subgraph isomorphic to $p \in \mathcal{P}$ and $I_i = 0$ otherwise. Hence, given D with clusters C, a pattern $p = (V_p, E_p)$, and a canned pattern set $\mathcal{P}$, the *pattern score* of p is:

$$s_p = ccov(p, cw, C) \times lcov(p, D) \times \frac{div(p, \mathcal{P} \setminus p)}{cog(p)} \quad (2)$$

Notice that as recommended by [37], we combine *scov*, *lcov*, *div*, and *cog* using multiplicative utility function as we do not have prior knowledge of the *trade-off rate* (*i.e., x* units of criterion A is equivalent to y units of criterion B) between these criteria (more details in [23]). Also, s_p increases when *ccov* or *div* increases or when *cog* decreases. Hence, given two candidate patterns p_1 and p_2, p_1 is considered superior to p_2 if $s_{p_1} > s_{p_2}$.

Recall that ged is used to compute the pattern set diversity (Section 3.2), which is known to be computationally expensive [32]. Hence, Catapult uses a *pruning step* based on the *lower bound* of ged to reduce the number of exact ged computation.

Definition 5.1. Given two graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, the ***lower bound ged*** *is given as $GED_l(G_A, G_B) = |V| + |E|$ where $L(V_A)$ is the set of labels of vertices in V_A, $|V| = ||V_A| - |V_B|| + Min(|V_A|, |V_B|) - |L(V_A) \bigcap L(V_B)|$ and $|E| = ||E_A| - |E_B||$.*

Observe that the lower bound computes the exact number of vertex modification ($|V|$ in Definition 5.1) and the minimum number of edge modifications ($|E|$) that are necessary.

Lemma 5.2. *Given two graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, the worst case time complexity of computing the lower bound of* ged *is $O(|V_A| log |V_A|)$ where $|V_A| \geq |V_B|$.*

The lower bound can be exploited to compute geds as follows: (a) Compute lower bound of ged (ged$_l$) of candidate patterns p_c with each canned pattern p in $\mathcal{P}'$. (b) Order canned patterns in $\mathcal{P}'$ in increasing ged$_l$ and store the list as Y. (c) Iterate through Y. In each iteration, (1) compute ged(p, p_c) where $p \in Y$, (2) update ged$_{min}$ if ged$(p, p_c) <$ged$_{min}$ and (3) remove all p from Y with ged$_l >$ged$_{min}$.

Updating weights (Lines 20 to 21). In this step, the cluster weight and edge label weight (recall from Section 3.3) are updated after each selection of a new canned pattern p by utilizing the *multiplicative weights update* method [2] as follows: (a) *Cluster weight update:* if the csg of a cluster C contains subgraph isomorphic to p, then the new cluster weight of C is $w'_C = (1 - n) \times w_C$ where w_C is the original cluster weight. We set $n = 0.5$ according to [2]. (b) *Edge label weight update:* if an edge e has label corresponding to that of an edge in p, then the new edge label weight of e is $w'_e = (1 - n) \times w_e$ where w_e is the original edge label weight.

Remark. Observe that the above algorithm results in higher chance of covering frequently occurring edge labels since there are more edges containing these labels and a walk is likely to pass through one or more such edges. These edges are likely to occur in frequent queries. In contrast, edge labels with low frequency are more likely to occur in infrequent queries. Hence, Catapult balances the number of frequent and infrequent patterns by using the aforementioned weighted random walk approach.

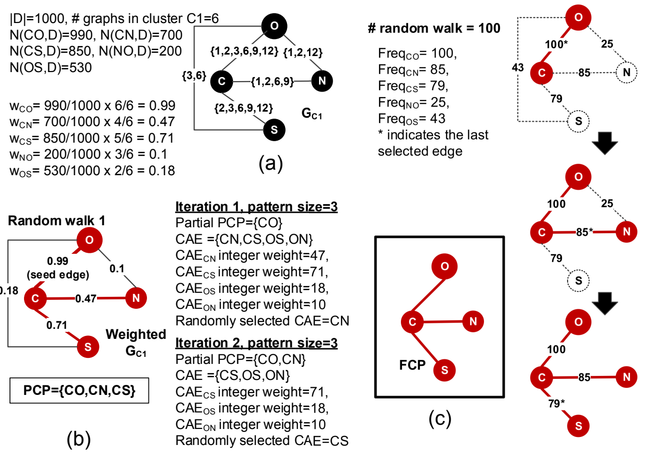

Figure 6: Canned pattern selection consisting of generating (a) weighted CSG; (b) PCP; and (c) FCP.

Furthermore, CATAPULT follows a uniform distribution in generating $\mathcal{P}$ to ensure that the sizes of canned patterns are evenly distributed (Definition 3.1). However, it can be easily modified as follows to accommodate a different size distribution by allowing the canned patterns per size to vary in the range [1-k] where $k < \gamma$: (1) change pattern budget to $b = (\eta_{min}, \eta_{max}, \Psi_{dist}, \gamma)$ where Ψ_{dist} is the desired pattern size distribution, and (2) modify the *GetPatternSizeRange* procedure (Line 5) to generate the range of required size based on Ψ_{dist} for each while-loop iteration.

Example 5.3. Figure 6 illustrates the canned pattern selection process. Let $\gamma = 9$, $\eta_{min} = 3$ and $\eta_{max} = 5$. CATAPULT first generates the weight w_e for each CSG (Figure 6(a)). For example, $w_{CO} = lcov(CO, D) \times lcov(CO, S_{C1}) = 0.99$. Then, random walks are performed to derive a library of PCPs for each pattern size. Figure 6(b) illustrates an instance of random walk to generate a PCP of size 3 for weighted CSG S_{C1}, starting at seed edge (C, O) (largest weight). A set of CAEs is obtained by converting edge weights to integers. For instance, the CAE of (C, N) consists of 47 copies of (C, N). A CAE (*e.g.*, (C, N)) is then randomly selected and added to the partial PCP. This process is repeated until the PCP is fully constructed. The constructed PCP (*e.g.*, {(C, O), (C, N), (C, S)}) is then added to the library. Next, CATAPULT proceeds to identify the FCP from the PCP library based on the frequency of labelled edges occurring in the library. Figure 6(c) illustrates the steps of finding a FCP of size 3 from S_{C1}. Based on 100 random walks, the most frequent edge is identified as (C, O), which forms the first edge in the FCP. The second edge (*i.e.*, (C, N)) in the FCP is the most frequent edge in the library that is connected to (C, O). CATAPULT continues to identify the next edge until the FCP is constructed. Note that in every iteration, each CSG "proposes" a FCP for each pattern size. The pattern score for each FCP is computed. The FCP (*e.g.*, $P_1 = \{(C, O), (C, N), (C, S)\}$) with the largest pattern score is then selected as the best candidate pattern and added to the set. The cluster weights are updated by first identifying all CSGs containing subgraphs that are isomorphic to P_1 and multiplying their cluster weights by 0.5. The weights of (C, O), (C, N) and (C, S) in edge label occurrence are updated as well by multiplying their initial weights by 0.5. CATAPULT repeats these steps for selecting subsequent canned patterns. ∎

THEOREM 5.4. *The worst case time and space complexities of canned pattern selection (Algorithm 4) are* $O(|V_{S_{max}}|!|V_{S_{max}}||\mathbb{S}|+|\mathcal{P}|(|V_{P(max)}|^3+x\eta_{max}^2|\mathbb{S}||E_{S_{max}}|))$ *and* $O(|\mathbb{S}|(|E_{S_{max}}|+\eta_{max}^2)+|D||E_{max}|)$*, respectively, where* S_{max} *is the largest CSG in the set of CSGs* $\mathbb{S}$ *and* x *is the number of random walk iterations.*

Remark. Our data-driven approach for selecting canned patterns enables an end user to customize her interface by specifying the pattern budget. Also, notice that the worst case time complexity of canned pattern selection is mainly due to subgraph isomorphism test necessary for checking cluster coverage. In this work, we use the VF2 algorithm [14].

6 PERFORMANCE STUDY

CATAPULT is implemented in Java with JDK1.8. Small graph clustering is realized in C++ using the *Boost* library. We now investigate the performance of CATAPULT and report the key results. Additional results and GUI details are discussed in Appendix C and [23]. All experiments are performed on a 64-bit Windows desktop with Intel Xeon CPU E5-1630 (3.70GHz) and 32GB of main memory.

6.1 Experimental Setup

Datasets. We use the following datasets. (a) The AIDS antiviral dataset[10] has 40,000 (40K) data graphs. We use it and its subset containing 10K graphs, referred to as AIDS40K and AIDS10K, respectively. (b) The *PubChem* dataset[11] consisting of 23,238 (23K); 250,000 (250K); 500,000 (500K); and 1 million (1M) chemical compound graphs. Unless otherwise stated, *PubChem* refers to the 23K dataset. (c) *eMolecule* dataset[12] consisting of 10K chemical compounds (referred to as *eMol*).

Competitors. We compare our data-driven approach with two commercial visual subgraph query interfaces (*PubChem* and *eMol*) where the canned patterns are manually selected. We also compare CATAPULT with a frequent subgraph-based canned pattern selection strategy as a baseline (Appendix C).

Query set. We generate subgraph queries by randomly selecting connected subgraphs from the dataset. For each dataset, **1000 subgraph queries** with sizes in the range of [4–40] are randomly generated.

Parameter settings. Unless specified otherwise, we set $\eta_{min} = 3$, $\eta_{max} = 12$, $N = 20$, $k = \frac{|D|}{N}$, and $|\mathcal{P}| = \gamma = 30$.

[10]https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data
[11]ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/
[12]https://www.emolecules.com/info/plus/download-database

Performance measures. We use the following measures for performance: (a) *Clustering time:* Time taken to perform clustering in the small graph clustering phase. (b) *Pattern generation time (PGT):* Time taken to select canned pattern set $\mathcal{P}$ (Algorithm 4). (c) *CSG compactness (denoted as ξ_t):* Given a graph cluster C and a threshold t, the *CSG compactness* of a CSG $S_C = (V_{S_C}, E_{S_C})$ of C is $\xi_t = \frac{|E_t|}{|E_{S_C}|}$ where every edge $e \in E_t \subseteq E_{S_C}$ is contained in at least $t \times |C|$ graphs in C. Intuitively, it measures the compactness of a CSG of a cluster. (d) *Missed percentage (MP):* Percentage of query set containing no canned patterns. $MP = \frac{|Q_M|}{|Q|} \times 100\%$ where Q is the query set and $Q_M \subseteq Q$ does not contain subgraphs that are isomorphic to any $p \in \mathcal{P}$. (e) *Reduction ratio (denoted as μ):* Given a subgraph query Q, μ is the ratio of the number of steps reduced when $\mathcal{P}$ is used for constructing Q to the total number of steps needed for constructing it using the edge-at-a-time mode ($step_{total}$). That is, $\mu = \frac{step_{total} - step_{\mathcal{P}}}{step_{total}}$ where $step_{\mathcal{P}}$ is the minimum number of steps required to construct Q when $\mathcal{P}$ is used. Note that a step refers to addition of a vertex/edge/pattern or relabelling a vertex label.

We assume a canned pattern $p \in \mathcal{P}$ can be used in Q iff $p \subseteq Q$. Further, when multiple patterns are used to construct Q, for simplicity we assume that their corresponding isomorphic subgraphs in Q do not overlap. Then, given Q and $\mathcal{P}$, the problem of finding a collection of canned patterns $\mathcal{P}_Q \subseteq \mathcal{P}$ that maximally covers Q can be modelled as a maximum weighted independent set problem [33] where each pattern $p \in \mathcal{P}_Q$ is contained in Q and the *weight* of p is the number of vertices in it. The maximum weighted independent set is $\mathcal{P}_Q$ and each pattern $p \in \mathcal{P}_Q$ is treated as a single step. Note that $\mathcal{P}_Q$ is strictly a bag as it may contain multiple instances of p if there are multiple non-overlapping subgraphs in Q that are isomorphic to p. Hence, $step_{\mathcal{P}} = |\mathcal{P}_Q| + |V_Q \backslash V_{\mathcal{P}_Q}| + |E_Q \backslash E_{\mathcal{P}_Q}|$.

6.2 Experimental Results

Exp 1: Small graph clustering. First, we evaluate the effect of our small graph clustering strategy in terms of clustering time and CSG compactness. Figure 7 reports the performance on AIDS10K and AIDS40K for the following scenarios: (a) coarse clustering only (CC), (b) MCCS-based fine clustering only (mccsFC), (c) MCS-based fine clustering only (mcsFC), and (d) coarse and fine clustering (*i.e.,* hybrid) with MCCS (mccsH) and MCS (mcsH). As expected, CC is generally faster but produces CSGs with low compactness. Occasional clusters with a large number of data graphs can result in CSGs with poor compactness due to the large variability in topology of the data graphs in a cluster. In our experiments, the largest cluster produced in AIDS40K contains 14000 graphs. In contrast, MCCS-based fine clustering (mccsFC) produces more compact CSGs but is much slower. Interestingly, our proposed hybrid approach mccsH produces CSGs that are

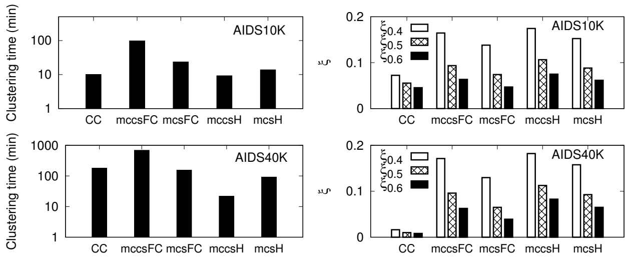

Figure 7: Small graph clustering phase.

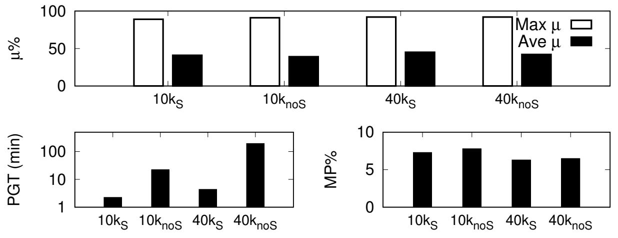

Figure 8: Effect of sampling. $10k_S$ (resp. $40k_S$) and $10k_{noS}$ (resp. $40k_{noS}$) denotes sampling and no sampling of *AIDS*10*k* (resp. *AIDS*40*k*), respectively.

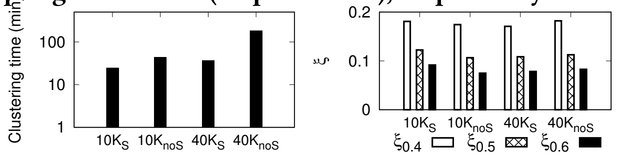

Figure 9: Effect of sampling on the clustering phase.

most compact for both datasets at a reasonable clustering time. *This justifies the need for hybrid strategy for small graph clustering.* In subsequent experiments, we shall use mccsH.

Exp 2: Sampling vs No sampling. Next, we evaluate the effect of sampling using AIDS dataset in terms of PGT, MP, and μ. For eager sampling, we set $\rho = 0.01$ and $\epsilon = 0.02$ whereas for lazy sampling, $p = 0.5$, $Z_{\frac{\alpha}{2}} = Z_{\frac{0.95}{2}}$ and $e = 0.03$. From Figure 8, we observe that there are no significant differences for both AIDS10K and AIDS40K in terms of μ and MP. But the PGT differs by up to 2 orders of magnitude. We also examine the effect of sampling on the quality of graph clusters. Figure 9 depicts that CSG compactness did not change significantly whereas the clustering time increases by up to one fold. Hence, *the sampling approaches in* CATAPULT *reduce running time significantly without affecting the quality of selected canned patterns significantly.* In the subsequent experiments, we shall use these sampling parameters.

Exp 3: Comparison with commercial GUI. We compare CATAPULT with *PubChem* (Figure 1) and *eMol* (https://reaxys.emolecules.com/). Canned patterns on the GUI that are of size 3 or larger are extracted for our study. Specifically, the *PubChem* GUI has 12 patterns with size in the range [3-8], of which 11 contain no vertex labels (referred to as *unlabelled patterns*). On the other hand, *eMol* GUI has 6 unlabelled patterns with size in the range [3-8]. Hence, we generate

Table 1: Queries used for user study. CID is the unique identifier of the *PubChem* repository.

Query	PubChem CID (*PubChem*)	PubChem CID (*eMol*)
Q_1	7809 ($\|E\|$=18)	57491213 ($\|E\|$=12)
Q_2	769013 ($\|E\|$=29)	98037 ($\|E\|$=17)
Q_3	169132 ($\|E\|$=34)	52426 ($\|E\|$=23)
Q_4	22749902 ($\|E\|$=39)	17081 ($\|E\|$=33)
Q_5	63559561 ($\|E\|$=40)	10097586 ($\|E\|$=35)

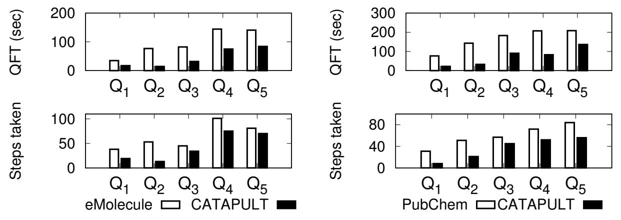

Figure 10: User study.

12 and 6 patterns in the size range [3-8] in CATAPULT for comparison with *PubChem* and *eMol*, respectively. Query sets for *PubChem* and *eMol* are generated according to Section 6.1. Furthermore, we redefine the reduction ratio as follows: $\mu_G = \frac{step_{\mathcal{P}(\text{GUI})} - step_{\mathcal{P}(\text{CATAPULT})}}{step_{\mathcal{P}(\text{GUI})}}$ where $step_{\mathcal{P}(X)}$ is the number of steps required to construct a subgraph query when $\mathcal{P}$ obtained from X is used.

Since majority of canned patterns in *PubChem* and *eMol* are unlabelled graphs, in order to compute MP and μ, we perform a *vertex relabelling step* before computing these measures. Specifically, we map unlabelled canned patterns to labeled subgraph queries. The queries are first relabelled such that all vertices have same label (*e.g.*, C) and all vertices in unlabelled canned patterns are assigned this label as well. Note that the vertex relabelling step is favorable to performances of these two GUIs as it underestimates the number of steps for $step_{\mathcal{P}(\text{GUI})}$. In *PubChem* and *eMol* GUIs, when unlabelled patterns are used, a user undertakes any one of the following steps to label its vertices: (1) *2-step labelling*: Select a vertex label (step 1), then click an unlabelled vertex to assign the label (step 2). (2) *1-step labelling*: Click on an unlabelled vertex to assign the label.

Note that *2-step labelling* is used if currently no vertex label is selected or if the previously selected label does not match the label of the current vertex whereas *1-step labelling* is used otherwise. For example, a user first specifies a vertex label (*e.g.*, C) by choosing from the GUI (Figure 1) and then click on a vertex v_1 she wishes to assign the label. Suppose the remaining vertices also have the same label C. Then she simply clicks on the remaining vertices in turn to assign the label. Hence, a total of two additional steps are needed to label vertex v_1 and one additional step to label remaining ones. Consequently, $step_{\mathcal{P}(\text{GUI})} = step_{\mathcal{P}(\text{GUI})} + |V_{P_l}|$ where $|V_{P_l}|$ is the total number of vertices in unlabelled canned patterns used to construct the subgraph query assuming the optimistic case of 1-step labelling.

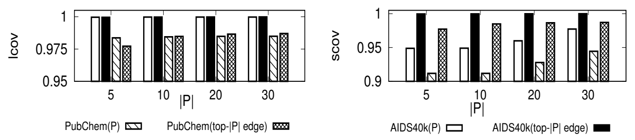

Figure 11: Effect of varying $|\mathcal{P}|$ on coverage.

First, we observe that the average *cog* of canned patterns is lowest in CATAPULT (*cog*(*eMol*) = 2.05 vs *cog*(CATAPULT) = 1.83; *cog*(*PubChem*) = 2.53 vs and *cog*(CATAPULT) = 2.01). The average diversity (*div*) of CATAPULT-derived patterns are high with values 9 (*eMol*) and 7.4 (*PubChem*). Second, patterns in CATAPULT have superior μ_G compared to *eMol* GUI, having maximum and average μ_G of 0.86 and 0.18, respectively. There are also fewer subgraph queries in CATAPULT that cannot be formulated using canned patterns ($\text{MP}_{\text{CATAPULT}}$=14.4 vs $\text{MP}_{eMol\text{GUI}}$=29.4). Third, *PubChem* has extremely low MP ($\text{MP}_{PubChem\text{GUI}}$=0.2 vs $\text{MP}_{\text{CATAPULT}}$=18.6) due to the lack of vertex labels (which relax vertex mapping) and the topological variety of patterns. However, CATAPULT-generated patterns still perform superior to *PubChem* having max. and avg. μ_G 0.79 and 0.03, respectively. *In summary, CATAPULT has the best performance.*

Exp 4: User study. We conducted a user study to investigate the impact of data-driven canned pattern selection on query formulation time (QFT). We compare CATAPULT with *PubChem* and *eMolecule* (referred to as $\mathcal{P}$(GUI)) for a set of user-formulated queries. For each dataset, we select 5 queries (Table 1) of size in the range [12-40] from respective repositories. These queries span a variety of structures (cycles, carbon chains, etc.) and contain different vertex labels (*i.e.*, $L(V_Q)$={C, Cl, H, N, O, S}). 25 unpaid volunteers (ages from 20 to 30) took part in the study in accordance to HCI research that recommends at least 10 participants [15, 26]. 56% and 20% of the volunteers have taken undergraduate chemistry/chemical engineering and biology courses, respectively. None of them are authors of this paper. They are trained to use the three GUIs. For every query, they were given some time to determine the steps needed to formulate it visually and are informed to use canned patterns as much as possible. Every query was formulated 5 times by 5 different participants. Recall that the vertices of $\mathcal{P}$(GUI) are unlabelled. Hence, as in *Exp. 3*, these vertices are assigned a common label that is not in $L(V_Q)$ and participants have to relabel them to the correct vertex label during query formulation. The QFT and the number of steps taken are recorded.

Figure 10 reports the avg. readings for each query. Note that QFTs include the search time for relevant patterns. Clearly, the canned patterns generated by CATAPULT facilitate more efficient (shorter QFT and lesser number of steps) query formulation than those obtained from the other GUIs. In particular, CATAPULT's patterns achieve up to 78% (resp. 81%)

and 74% (resp. 75%) reduction in terms of QFT and number of steps, respectively, for *PubChem* (resp. *eMol*) queries.

Exp 5: Coverage. We examine the coverage of $\mathcal{P}$ using *scov* and *lcov* (See Section 3.2) and compare them with the coverage of top-$|\mathcal{P}|$ frequent edges. Figure 11 plots the results for AIDS40K and *PubChem*. Results are qualitatively similar for other datasets. We observe that *scov* increases as $|\mathcal{P}|$ increases. This highlights that additional canned patterns added to the set are topologically distinct from existing ones, resulting in increase of coverage. Naturally, the top-$|\mathcal{P}|$ frequent edges have higher *scov* than CATAPULT's canned patterns due to their small size and greater chance to occur in a data graph. For smaller $|\mathcal{P}|$, CATAPULT's canned patterns tend to have slightly higher *lcov* compared to the top-$|\mathcal{P}|$ frequent edges since canned patterns are larger and there is higher likelihood of having more unique edges compared to the frequent edges. However, as $|\mathcal{P}|$ increases, this effect is reversed. This is due to CATAPULT's canned patterns having a relatively stable set of unique edges generated by our random walk-based algorithm which tends to favour paths with greater support. In contrast, distinct edge labels of top-$|\mathcal{P}|$ frequent edges grow as $|\mathcal{P}|$ increases. In particular, *scov* (resp. *lcov*) of top-$|\mathcal{P}|$ frequent edges and CATAPULT's canned patterns vary in the range [0.98-1] (resp. [0.98-1]) and [0.91-0.98] (resp. [0.98-1]), respectively. Recall that the edge-at-a-time mode is inefficient as it may require more steps than the pattern-at-a-time mode. Hence, *CATAPULT's patterns not only have good coverage (scov ~ 94% on average for all datasets) but also support efficient query formulation.*

Exp 6: Scalability. We examine the scalability of CATAPULT using *PubChem* with dataset sizes in the range $\{23K, 250K, 500K, 1M\}$. Similar to *Exp 3*, 12 canned patterns of size in the range [3-8] are extracted using CATAPULT. Figure 12 reports the results. As expected, clustering time and PGT increase as $|D|$ increases. In particular, the increase is about an order of magnitude when $|D|$ increases from 23K to 1 million data graphs. The larger datasets also resulted in lower MP and negative average relative reduction ratio μ_{DS} where $\mu_{DS} = \frac{step_{\mathcal{P}(PubChem_{DS})} - step_{\mathcal{P}(PubChem_{23K})}}{step_{\mathcal{P}(PubChem_{DS})}}$, $DS \in \{23K, 50K, 500K, 1M\}$ and $step_{\mathcal{P}(PubChem_x)}$ is the number of steps required to construct a subgraph query based on canned patterns generated by CATAPULT for *PubChem* dataset of x size. The negative μ_{DS} implies that on average, $step_{\mathcal{P}(PubChem_{DS})} < step_{\mathcal{P}(PubChem_{23K})}$ when $DS > 23K$. That is, the quality of canned patterns improves with the dataset size. Interestingly, improvements in terms of MP and μ_{DS} show an anti-monotonic trend where the best results are obtained when $|D| = 250K$. μ_{DS} and MP improve by 21.2% and 43%, respectively, when compared to $|D| = 23K$. Comparatively, PGT and clustering time are 8.43 and 2.93 times slower. This is likely due to two competing effects: (a)

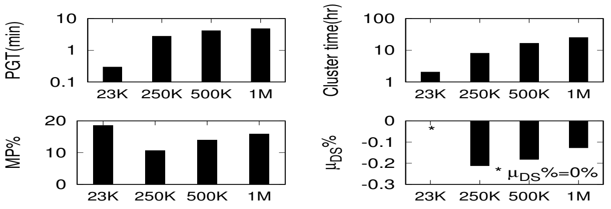

Figure 12: Scalability study (*PubChem*).

larger dataset improves quality of canned patterns but (b) sampling degrades the pattern quality. Specifically, when the dataset size increases from 23K to 250K the effect of former is dominant but subsequently the latter has greater impact. Hence, *we can generate high quality canned patterns without processing the entire dataset* (*e.g.,* 250K instead of 1M).

7 RELATED WORK

The vision of data-driven construction of visual graph query interfaces was laid out by Bhowmick *et al.* in [6] and a demo presented in [40]. Our work builds upon these high-level ideas. In particular, [40] neither incorporates frequent tree-based clustering nor sampling strategies to handle larger graph databases. Furthermore, canned patterns are generated greedily using breadth-first-search by maximizing an objective function that is based on the size of candidate patterns and their coverage within a CSG. That is, it ignores label coverage, diversity, and cognitive load of canned patterns. Also, candidate pattern generation and selection are treated as two separate phases. In contrast, candidate pattern generation and selection are intertwined in CATAPULT by utilizing a random walk-based strategy. More recently, Zhang *et al.* [41] demonstrated the idea of *precision interfaces* with the vision of generating interactive analysis interfaces from query logs. This work is primarily focused on structured data (*e.g.,* SQL) and does not address the canned pattern selection problem.

8 CONCLUSIONS

In this paper, we take a concrete step towards realizing the vision of data-driven visual graph query interface construction. We focus on the problem of automatic selection of canned patterns, which is at the core of expediting visual query formulation. We present a novel framework called CATAPULT to this end. Specifically, we propose a small graph clustering strategy to summarize topologically similar data graphs into CSGs and a random walk-based strategy to select canned patterns with high coverage, high diversity, and low cognitive load from them. Our experimental study demonstrates superiority of the CATAPULT framework to manually-selected canned patterns in traditional visual graph query interfaces.

Acknowledgments. K. Huang, S. S. Bhowmick and H. E. Chua are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. B. Choi is supported by HK-RGC GRF 12201315 and 12232716. S. Zhou is supported by National Natural Science Foundation of China (NSFC) under grant No. U1636205.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, J.E. Ullman. The design and analysis of computer algorithms. *Addison-Wesley*, 1974.

[2] S. Arora, E. Hazan, S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory Comput.* 8(1), 2012.

[3] D. Arthur, S. Vassilvitskii. How slow is the k-means method?. *In SCG*, 2006.

[4] D. Arthur, S. Vassilvitskii. k-means++: The advantages of careful seeding. *In SIAM*, 2007.

[5] B. Bahmani,et al. Scalable k-means++. *In VLDB*, 2012.

[6] S. S. Bhowmick, B. Choi, C. E. Dyreson. Data-driven visual graph query interface construction and maintenance: challenges and opportunities. *PVLDB*, 9(12): 984-992, 2016.

[7] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(8):689-694, 1997.

[8] T.Y.S. But, P.H. Toy. The Mitsunobu reaction: origin, mechanism, improvements, and applications. *Chem. Asian J.*, 2(11):1340-1355, 2007.

[9] Y. Chi, et al. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowl. Inf. Syst.*, 8(2), 2005.

[10] Y. Chi, et al. Indexing and mining free trees. *In ICDM*, 2003.

[11] W.G. Cochran. *Sampling techniques.* Third edition. Wiley, New York, New York, USA.

[12] C.A.C Coello, G.B. Lamont, D.A. Van Veldhuizen. Evolutionary algorithms for solving multi-objective problems. *2nd Edition*, Springer, 2007.

[13] D. Conte, P. Foggia, M. Vento. Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 11(1):99-143, 2007.

[14] L.P. Cordella, P. Foggia, C. Sansone. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367-1372, 2004.

[15] L. Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3), 2003.

[16] F. Geerts, et al. Relational link-based ranking. *In VLDB*, 2004.

[17] C. Guestrin, A. Krause, A.P. Singh. Near-optimal sensor placements in gaussian processes. *In ICML*, 2005.

[18] S. Günter, H. Bunke. Self-organizing map for clustering in the graph domain. *Pattern Recogn. Lett.*, 23(4):405-417, 2002.

[19] H. He, A.K. Singh. Closure-tree: An index structure for graph queries. *In ICDE*, 2006.

[20] W. Huang, P. Eades, S.H. Hong. Measuring effectiveness of graph visualizations: A cognitive load perspective. *Inf. Vis.*, 8(3): 139-152, 2009.

[21] K. Jain, V.V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and Lagrangian relaxation. *J. ACM*, 48(2):274-296, 2001.

[22] N. Jiang, Y. Bu, Y. Wang, M. Nie, D. Zhang, X. Zhai. Design, synthesis and structure-activity relationships of novel diaryl urea derivatives as potential EGFR inhibitors. *Molecules*, 21(11): 1572, 2016.

[23] H. Kai et al. CATAPULT: data-driven selection of canned patterns for efficient visual graph query formulation. *Technical Report.* Available at: http://www.ntu.edu.sg/home/assourav/TechReports/catapult-TR.pdf, June 2018.

[24] S. Khuller, A. Moss, J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1): 39-45, 1999.

[25] S.G. Kobourov, S. Pupyrev, B. Saket. Are crossings important for drawing large graphs? *In GD*, 2014.

[26] J. Lazar, J.H. Feng, H. Hochheiser. Research methods in human-computer interaction. John Wiley & Sons, 2010.

[27] J.J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):29-34, 1982.

[28] M.D. McKay, R.J Beckman, W.J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239-245, 1979.

[29] M. Meilă. The uniqueness of a good optimum for k-means. *In ICML*, 2006.

[30] S. Nijssen, J.N. Kok. The gaston tool for frequent subgraph mining. *Electron. Notes Theor. Comput. Sci.*, 127(1): 77-87, 2005.

[31] T. Ramraj, R. Prabhakar. Frequent subgraph mining algorithms - a survey. *Procedia Computer Science*, 47:197-204, 2015.

[32] K. Riesen, M. Neuhaus, H. Bunke. Bipartite graph matching for computing the edit distance of graphs. *In GbRPR*, 2007.

[33] S. Sakai, M. Togasaki, K. Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Appl. Math.* 126(2-3): 313-322, 2003.

[34] S.E. Schaeffer. Graph clustering. *Comput. Sci. Rev.*, 1(1):27-64, 2007.

[35] T. Schäfer, P. Mutzel. StruClus: structural clustering of large-scale graph databases. *CoRR abs/1609.09000*, 2016.

[36] H. Shang, X. Lin, Y. Zhang, J.X. Yu, W. Wang. Connected substructure similarity search. *In SIGMOD*, 2010.

[37] C. Tofallis. Add or multiply? A tutorial on ranking and choosing with multiple criteria. *INFORMS Trans. on Education*, 14(3): 109-119, 2014.

[38] H. Toivonen. Sampling large databases for association rules. *In VLDB*, 1996.

[39] Y. Chi, R.R. Muntz, S. Nijssen. Frequent subtree mining. In T. Washio, et al., editors, *Advances in Mining Graphs, Trees and Sequences*, 2005.

[40] J. Zhang, et al. DaVinci: Data-driven visual interface construction for subgraph search in graph databases. *In ICDE*, 2015.

[41] H. Zhang, V. Raj, T. Sellam, E. Wu. Precision interfaces for different modalities. *In SIGMOD*, 2018.

A PROOFS

Proof of Theorem 3.2 (Sketch). Multi-objective optimization problem has complexity at least as hard as any of its single objectives since additional objectives introduce additional constraints on the solutions and optimizing a single objective may result in solutions that are sub-optimal with regards to other objectives. In the canned pattern selection problem, which optimizes four objectives, the maximum subgraph coverage objective is akin to the maximum set cover problem where given a number k and a collection of sets $S = \{S_1, S_2 \cdots, S_m\}$, the goal is to select at most k of these sets such that the maximum number of elements are covered. Here, k is the number of canned patterns allowed in the GUI and S is the set of candidate patterns where each pattern covers a certain number of graphs in the graph repository D. The maximum set cover problem is NP-hard and a greedy algorithm achieves an approximation of $(1 - \frac{1}{e})$ [24]. Hence, the canned pattern selection problem is also NP-hard.

Time complexity of small graph clustering (Sketch). In small graph clustering, frequent subtrees in coarse clustering (Algorithm 2, Line 1) are obtained using the approach in [10] which has worst-case time complexity of $O(b|D|l^2 + \frac{c|D|l^{1.5}|V_{max}|}{log l})$ [39] where b is the maximum number of trees a graph can contain; $|D|$ is the number of graphs considered; l is the number of leaves removed; and c is the number of candidate frequent trees. Refinement of the set of frequent subtrees (Line 2) is achieved by maximizing a submodular function. In the worst-case, the entire set of frequent subtrees is selected (*i.e.*, $|\mathcal{T}_{sel}| = |\mathcal{T}_{all}|$) and the worst-case time complexity is $O(|\mathcal{T}_{all}|^2(3|V_{max}|)^2)$ since the largest breadth-first canonical string is $3|V_{max}|$ [9]. The generation of the feature vectors (Lines 3 to 10) iterates through each pair of $G_i \in D$ and $T_j \in \mathcal{T}_{sel}$. In the worst case, $|\mathcal{T}_{sel}| = |\mathcal{T}_{all}|$ and the time complexity is $O(|D||\mathcal{T}_{all}|)$. Seed selection using k-means++ and graph clustering using k-means (Line 11) have worst-case time complexities of $O(k|D||\mathcal{T}_{all}|)$ [5] and $O(2^{\Omega(\sqrt{|D|})})$ [3], respectively.

In the fine clustering phase (Algorithm 3), cluster size checking (Line 2) takes $O(k)$ time since there are k clusters. For each cluster-splitting step, the worst-case time complexity of measuring similarity based on MCCS takes $O(\sum_{i\in|D|}\frac{(|V_{Seed1}|+1)!}{(|V_{Seed1}|-|V_i|+1)!)})$ [13] whereas that for selecting a dissimilar graph (Line 12) takes $O(log(|C_{first}|-1))$ time. In general, $O(\sum_{i\in|D|}\frac{(|V_{Seed1}|+1)!}{(|V_{Seed1}|-|V_i|+1)!)}) \gg O(log(|C_{first}|-1))$. Hence, in the worst case, after k-means clustering, k-1 clusters contains 1 graph each and the remaining cluster contains $|D|-k-1$ graphs. Assuming $|V_{seed}| = |V_{max}|$ and $|D|-k-1 > N$ (*i.e.*, fine clustering is required), the fine clustering phase takes $O(\sum_{i=1}^{|D|-N-k-1}(|D|-k-1-i)\frac{(|V_{max}|+1)!}{(|V_{max}|-|V_i|+1)!)})$ where the last cluster produced by the fine clustering phase contains N graph whereas every other cluster (those generated by the fine clustering phase) contains one graph.

Taken together, the small graph clustering phase has worst-case time complexity of $O(b|D|l^2+\frac{c|D|l^{1.5}|V_{max}|}{logl}+(|\mathcal{T}_{all}||V_{max}|)^2+k|D||\mathcal{T}_{all}|+2^{\Omega(\sqrt{|D|})}+(|D|-N-k-1)\times\sum_{i\in|D|}\frac{(|V_{max}|+1)!}{(|V_{max}|-|V_i|+1)!})$. The worst-case time complexity reduces to $O(2^{\Omega(\sqrt{|D|})})$ for large dataset.

Proof of Lemma 4.2 (Sketch). Given a dataset of graphs $D = \{G_1, G_2, \cdots, G_n\}$, we denote the optimum clustering of D into k disjoint, nonempty clusters as $C_{OPT} = \{C_1, C_2, \cdots, C_k\}$. We further denote the clustering obtained by small graph clustering as $C = \{C'_1, C'_2, \cdots, C'_{k'}\}$. The misclassification error distance of C with respect to C_{OPT} is essentially $\frac{|D'|}{|D|}$ where D' is the set of misclassified graphs based on C_{OPT} [29]. We assume fine clustering based on MCCS produces C_{OPT}. In the worst case small graph clustering performs coarse clustering only (frequent subtree-based feature vector clustering) when sizes of all generated clusters are less than or equals to N. Let A denote correct classification of G_i, B denote that the frequent subtree set of D contains MCCS of two graphs G_i and G_x. Observe that the probability of B denoted as $Pr(B)$ is equivalent to min_fr (Theorem 4.4). Then, $Pr(A) = Pr(A\bigcap(B\bigcup\overline{B})) = Pr(A\bigcap B) + Pr(A\bigcap\overline{B}) = Pr(A|B)Pr(B) + Pr(A|\overline{B})Pr(\overline{B}) = Pr(A|B)min\_fr + \frac{1}{2}(1 - min\_fr)$ where $Pr(A|\overline{B}) = 0.5$ since in the worst case, there is random chance of correct classification given $\overline{B}$. $Pr(A|B)$ is the probability of correct classification given B and this is likely to occur when $C_m = \{m|\max_{j\in G_C}\textsc{mccs}(G_i, j), m \in G_C\}$ and $C_m = \{m|\max_{l\in G_C}\textsc{subtree}(G_i, l), m \in G_C\}$ where $\textsc{subtree}(G_i, l)$ is the similarity of the frequent subtree vector of G_i and l. Hence, the small graph clustering achieves $\frac{1}{2} + (\alpha - \frac{1}{2})min\_fr$-approximation of C_{OPT} where $Pr(A|B) = \alpha$.

Proof of Lemma 4.3 (Sketch). The worst-case time complexity to form a closure graph is $O(|V_{max}|d^2log(|V_{max}|))$ [19] where d is the maximum degree of vertices. In the worst-case, there will be $|D|-N-k-1$ clusters and the resulting time complexity is $O((|D|-N-k-1)|V_{max}|d^2log(|V_{max}|))$. For large dataset, $|D| \gg N$ and $|D| \gg k$. Hence, the worst-case time complexity is reduced to $O(|D||V_{max}|d^2log(|V_{max}|))$.

Space complexity: The worst-case space complexity for storing the graph clusters C is $O(|D|(|E_{max}|+|V_{max}|))$ whereas that for storing the set of closure graphs is $O(|C|(|E_{max}| + |V_{max}|))$. In addition, the worst-case space complexity for generating a graph closure is $O(|V_{max}| + |E_{max}|)$ [19]. Taken together, the worse-case space complexity for generation of CSG is $O(|D|(|E_{max}| + |V_{max}|))$ since $|D| \gg |C|$.

Proof of Lemma 4.5 (Sketch). Each graph cluster C can be considered as a strata. Under proportional stratified sampling, the sample size of a strata is given as $|S_C| = \frac{|S_G||N_C|}{|N_G|}$ [28] where $|S_C|$ and $|S_G|$ are the sample sizes for cluster C and the set of graphs G, respectively; and $|N_C|$ and $|N_G|$ are the population size of C and G, respectively. Further, for large population, a representative sample size can be obtained as $|S_{sample}| = \frac{Z^2pq}{e^2}$ where Z^2 is the abscissa of the normal curve that cuts off an area α at the tails (1-α is the desired confidence level, *e.g.*, 95%), e is the desired level of precision, p is the estimated proportion of a graph being sampled in the dataset and $q = 1 - p$ [11].

Proof of Lemma 5.2 (Sketch). The worst case time complexity of computing the lower bound GED is due to the identification of common vertex labels in $L(V_A)$ and $L(V_B)$. This can be done via sorting both label lists and then comparing them which yields complexity of $O(|V_A|log|V_A|)$.

Proof of Theorem 5.4 (Sketch). Finding weights of edges in the closure graphs require $O(|\mathbb{S}||E_{\mathbb{S}_{max}}|)$ time in the worst case where $\mathbb{S}_{max} \in \mathbb{S}$ is the largest closure graph. Generating PCP requires $O(x\eta_{max}^2|\mathbb{S}||\mathcal{P}||E_{\mathbb{S}_{max}}|)$ time where x is the number random walk iterations. CATAPULT utilizes edge occurrence from the random walk to identify the FCP. For every PCP library, computing edge occurrence requires $O(x\eta_{max})$ time while FCP generation takes $O(\eta_{max}|E_{\mathbb{S}_{max}}|)$ time. Computing pattern score requires subgraph isomorphism test for each closure graph to find *ccov* ($O(|V_{\mathbb{S}_{max}}|!|V_{\mathbb{S}_{max}}|)$ [14]) and $|\mathcal{P}'|$ times of graph edit distance computation ($O(|V_{P(max)}|^3)$ [32]

where $P(max)$ is the largest pattern in $\mathcal{P}$) to find div, yielding $O(|V_{\mathbb{S}_{max}}|!|V_{\mathbb{S}_{max}}||\mathbb{S}| + |\mathcal{P}||V_{P(max)}|^3)$ worst case time complexity for each FCP. Updating of cluster weights and edge label occurrence require $O(|V_{\mathbb{S}_{max}}|!|V_{\mathbb{S}_{max}}||\mathbb{S}|)$ and $O(\eta_{max})$ time, respectively. Taken together, the pattern mining and selection phase have worst-time complexity of $O(|V_{\mathbb{S}_{max}}|!|V_{\mathbb{S}_{max}}||\mathbb{S}| + |\mathcal{P}|(|V_{P(max)}|^3 + x\eta^2_{max}|\mathbb{S}||E_{\mathbb{S}_{max}}|))$.

Space complexity: There are $|\mathcal{P}|$ canned patterns. Since we expect canned patterns to be subgraphs of D, their sizes should be less than $O(|V_{max}| + |E_{max}|)$. Hence, storage space needed for candidate patterns is $O(|\mathcal{P}|(|V_{max}| + |E_{max}|))$. In addition, CATAPULT allocates weights to each closure graph and this requires $O(|\mathbb{S}||E_{\mathbb{S}_{max}}|)$ space. In the worst case, maintaining the ELW requires $O(|D||E_{max}|)$ space assuming that every edge in each graph in D has a unique label. For each PCP library, $O(x\eta_{max})$ space is needed where x is the number of random walk instances. During each iteration, there are $\eta_{max} - \eta_{min} + 1$ candidate canned patterns per closure graph. Hence, in the worst case, Algorithm 4 has space complexity $O(|\mathcal{P}|(|V_{max}| + |E_{max}|) + |\mathbb{S}||E_{\mathbb{S}_{max}}| + |D||E_{max}| + \eta^2_{max}|\mathbb{S}|)$ since $x\eta_{max} \ll |D||E_{max}|$ in a large graph repository. This can be further reduced to $O(|D||E_{max}| + |\mathbb{S}|(|E_{\mathbb{S}_{max}}| + \eta^2_{max}))$ since $|D| \gg |\mathcal{P}|$ and in the worst case, G_{max} is a strongly connected graph where $|E_{max}| > |V_{max}|$.

B UNCAPACITATED FACILITY LOCATION PROBLEM

Given the cost for opening facilities and the cost for connecting cities to facilities, the *uncapacitated facility location problem* seeks a solution that minimizes the cost of connecting each city to an open facility [21]. Formally, given a bipartite graph with bipartition (F, C) where F is the set of facilities and C is the set of cities, let f_i be the cost of opening facility i and c_{ij} be the cost of connecting city j to (opened) facility i. The **uncapacitated facility location problem** seeks to

$$\begin{aligned} &min \sum_{i\in F, j\in C} c_{ij}x_{ij} + \sum_{i\in F} f_i y_i \\ &\text{s.t. } \forall j \in C : \sum_{i\in F} x_{ij} \geq 1, \forall i \in F, j \in C : y_i - x_{ij} \geq 0 \qquad (3) \\ &\forall i \in F, j \in C : x_{ij} \in \{0,1\}, \forall i \in F : y_i \in \{0,1\} \end{aligned}$$

where y_i and x_{ij} are indicator variables denoting whether facility i is open and whether city j is connected to facility i, respectively. Note that the first constraint ensures that every city is connected to at least one facility whereas the second constraint ensures that this particular facility must be open.

C ADDITIONAL RESULTS

Exp 7: Size of $|\mathcal{P}|$. We examine the effect of varying $|\mathcal{P}|$. We observe that varying $|\mathcal{P}|$ do not have significant effect on μ (Figure 13). As expected, PGT increases as $|\mathcal{P}|$ increases and this effect is most noticeable in the larger dataset (AIDS40K).

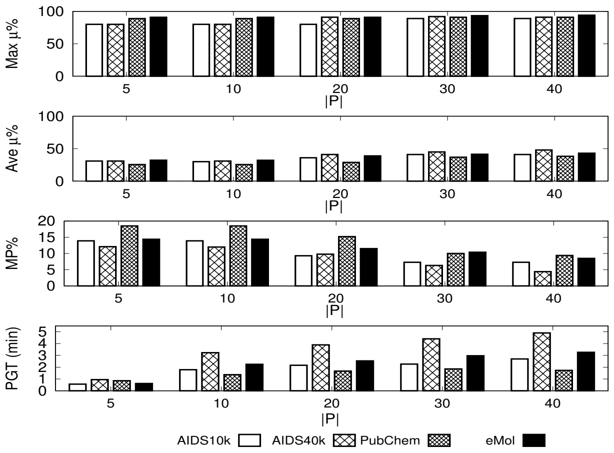

Figure 13: Effect of varying $|\mathcal{P}|$.

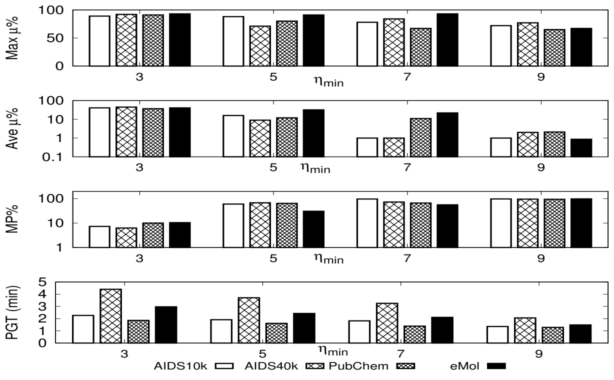

Figure 14: Effect of varying η_{min}.

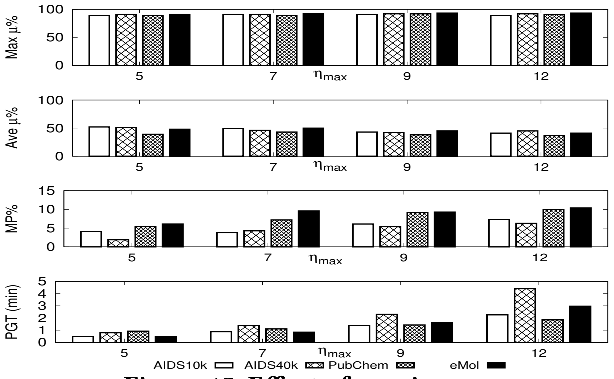

Figure 15: Effect of varying η_{max}.

In addition, improved coverage of the query set is observed as the number of canned patterns is increased. In particular, MP displays a downward trend where there is $\sim$ 50% reduction when $|\mathcal{P}|$ is increased from 10 to 40. The average cog of patterns in $\mathcal{P}$ for all dataset varies in the range [1.65–1.97], highlighting low cognitive load of the patterns.

Exp 8: Varying pattern size. In this set of experiments, we examine the effect of varying the pattern size. We first set $\eta_{max} = 12$ and vary η_{min} in the range $[3-9]$. From Figure 14, we observe that the increase in η_{min} results in increasing MP

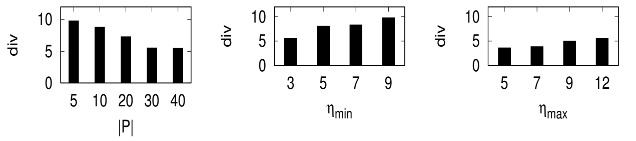

Figure 16: Effects of varying pattern budget on *div*.

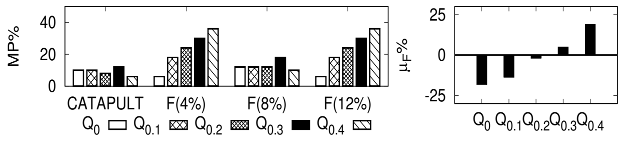

Figure 17: CATAPULT vs frequent subgraph patterns.

and correspondingly, decreasing average μ. This is due to the fact that the probability of a query graph Q containing a large canned pattern is comparatively lower than that of a small canned pattern. As expected, PGT decreases as η_{min} increases since there are fewer PCPs generated in Algorithm 4.

Next, we set $\eta_{min} = 3$ and vary η_{max} in the range [5 – 12]. We observe that varying η_{max} has little impact on MP (Figure 15) as compared to varying η_{min}. In particular, when η_{max} is varied, MP$_{max}$-MP$_{min}$ varies in the range [3.5 – 4.3]. In contrast, when η_{min} is varied, MP$_{max}$-MP$_{min}$ varies in the range [84.2 – 89.9]. Due to the relatively small effect on MP, maximum and average μ remain relatively constant when η_{max} is varied. PGT increases as η_{max} increases due to the generation of larger number of PCPs.

Finally, we examine the effect of varying η_{min}, η_{max} and $|\mathcal{P}|$ on *div* and *cog* for AIDS10K. We observe that increasing $|\mathcal{P}|$ resulted in decreasing *div* (Figure 16). This is expected as it is more likely to find a similar graph in a large dataset than a small one. When η_{min} increases, we observe increasing *div* as there tend to be greater diversity between larger patterns. In contrast, *cog* remains relatively constant ($cog \in [1.59 - 2.36]$). Results are qualitatively similar in other datasets.

Exp 9: Comparison with frequent subgraph-based technique. We use the AIDS10K for this experiment. We generate frequent subgraphs (denoted as $\mathcal{F}$) using GASTON [30]. In particular, we set $|\mathcal{F}| = 30$ where every frequent subgraph has size in the range [3 – 12] and the maximum number of patterns per size is $\frac{|\mathcal{F}|}{(12-3+1)}$. We vary the support threshold in the range $\{4\%, 8\%, 12\%\}$. Similar to *Exp 3*, we redefine the reduction ratio as follows: $\mu_{\mathcal{F}} = \frac{step_{\mathcal{F}} - step_{\mathcal{P}}}{step_{\mathcal{F}}}$. Note that for this experiment we cannot simply choose the randomly generated query set. This is because such query set may be unduely biased towards containing frequent subgraphs as they occur more often. In real-world applications user queries can be frequent or infrequent subgraphs. Hence, we generate query set Q_x where x is the fraction of queries that are infrequent. We set $|Q_x| = 50$.

Figure 17 plots the results. Observe that when $x = 0$, the queries are all frequent. Naturally, CATAPULT performs worse

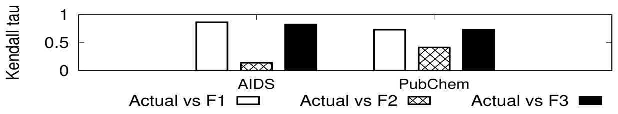

Figure 18: Comparison of *cog* metrics.

as its canned patterns contain a mixture of frequent and infrequent patterns. However, when $x > 0$, CATAPULT's performance improves and outperforms the frequent subgraph-based technique when $x = 0.3$. Specifically, MP for CATAPULT remains relatively constant whereas it increases linearly for $\mathcal{F}$ (4% and 12%). Furthermore, the *div* of CATAPULT is higher than $\mathcal{F}$ (7.4 vs 1.74). *In summary, patterns generated by CATAPULT are superior to frequent subgraph-based patterns.*

Exp 10: Cognitive load. Finally, we evaluate three putative measures ($F1$ to $F3$) for determining cognitive load. Given a pattern $p = (V_p, E_p)$, $F1 = |E_p| \times 2\frac{|E_p|}{|V_p|(|V_p|-1)}$ (Recall from Section 3.2), $F2 = \sum_{v \in V_p} deg(v) = 2|E_p|$ where $deg(v)$ is the degree of vertex v, and $F3 = 2\frac{|E_p|}{|V_p|}$. Observe that $F2$ is a degree-based measure. We conducted a user study with 15 participants on two datasets where each participant is given a pattern p and a query Q pair one at a time on a GUI and asked to determine if $p \subseteq Q$ (*i.e.,* whether p is useful for formulating Q) by clicking a yes/no button. The time taken from viewing p to clicking the button is recorded. For each dataset, 6 queries (size in range [18-39]) and 6 patterns of different topologies and cognitive load are given to each participant in random order. In particular, $|V|$ and $|E|$ of a pattern vary in the range [4-13] and [3-13], respectively. Note that not all $p \subseteq Q$ and an incorrect decision on a (p, Q) pair from a participant is ignored (97.2% decisions are correct). This is to minimize cases where a participant clicks the button without checking. For each dataset, the patterns are ranked in increasing time for each participant, where the smallest rank indicates shortest time taken and implies lowest cognitive effort needed to perform the task. Then, for a pattern p_i, an average rank is obtained by averaging the ranks assigned to p_i. Finally, the overall rank (*Actual rank*) of the pattern set for a given dataset is obtained by ordering the patterns in increasing average rank. Note that we do not use the average time taken to perform the overall rank as rank reversal may occur due to outliers (*e.g.,* extremely long time taken by a participant). Further, the patterns are given another set of ranking in increasing $F1$ (corr. for $F2$ and $F3$). Figure 18 plots the Kendall rank correlation coefficient of the actual ranks w.r.t. the ranks obtained using $F1$, $F2$ and $F3$. Observe that $F1$ *(avg. 0.8) is a more effective measure compared to* $F2$ *(avg. 0.28) and* $F3$ *(avg. 0.78).* Interestingly, users required the longest time on average for a clique pattern (*e.g.,* $|V| = 4$, $|E| = 6$) due to increased edge crossings leading to more decision making time [25].