FERRARI: An Efficient Framework for Visual Exploratory Subgraph Search in Graph Databases

[Technical Report]

Chaohui Wang[1,4] Miao Xie[1,5] Sourav S Bhowmick[1] Byron Choi[2] Xiaokui Xiao[3] Shuigeng Zhou[4]
[1]School of Computer Science & Engineering, Nanyang Technological University, Singapore
[2]Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR
[3]School of Computing, National University of Singapore, Singapore
[4]Shanghai Key Lab of Intelligent Information Processing, School of Computer Science, Fudan University, China
[5]Alibaba Group, Hangzhou, China
assourav@ntu.edu.sg, choi@hkbu.edu.hk, xkxiao@nus.edu.sg, sgzhou@fudan.edu.cn

***Abstract*—Exploratory search paradigm assists users who do not have a clear search intent and are unfamiliar with the underlying data space. Query formulation evolves iteratively in this paradigm as a user becomes more familiar with the content. Although exploratory search has received significant attention recently in the context of structured data, scant attention has been paid for graph-structured data. An early effort for building *exploratory subgraph search* framework on graph databases suffers from efficiency and scalability problems. In this paper, we present a visual exploratory subgraph search framework called FERRARI, which embodies two novel index structures called VACCINE and ADVISE, to address these limitations. VACCINE is an offline, *feature-based* index that stores rich information related to *frequent* and *infrequent subgraphs* in the underlying graph database and how they can be *transformed* from one subgraph to another during visual query formulation. ADVISE, on the other hand, is an *adaptive*, compact, on-the-fly index instantiated during iterative visual formulation/reformulation of a subgraph query for exploratory search and records relevant information to efficiently support its repeated evaluation. Extensive experiments and user study on real-world datasets demonstrate superiority of FERRARI to a state-of-the-art visual exploratory subgraph search technique.**

I. Introduction

Large collections of small- or medium-sized data graphs are prevalent nowadays in a variety of domains such as cheminformatics (*e.g.,* chemical compounds), bioinformatics (*e.g.,* protein structures), and computer vision (*e.g.,* object recognition). For example, more than a million chemical compounds and drugs are publicly available from sources such as *DrugBank*[1], *eMolecules*[2], and *PubChem*[3]. Subgraph search, which retrieves data graphs containing *exact* or *approximate* match of a user-specified query graph (*i.e.,* subgraph query), is a common and important query primitive for querying these data graphs. Given a *subgraph containment query* (resp. *substructure/subgraph similarity query*) q on a set of data graphs $\mathcal{D}$, the aim is to find all data graphs in $\mathcal{D}$ that contain *exact* (resp. *approximate*) match of q [39], [40].

[1]https://www.drugbank.ca/
[2]https://www.emolecules.com/
[3]https://pubchem.ncbi.nlm.nih.gov/

Majority of research and commercial efforts for subgraph search have focused on "lookup" (*i.e.,* one-shot) query with the assumption that users have clear intent and sufficient knowledge of $\mathcal{D}$ to accurately specify their search goal in form of a connected query graph. *Exploratory search* [25], [36] represents a class of search activities that involves more complicated search process than lookup retrieval. The search process is evolving in nature, and users typically issue multiple queries due to evolving information need. In particular, the search state is ambiguous in the beginning and the query formulation evolves iteratively as a user becomes more familiar with the content. Hence, exploratory search activities are considered as open-ended [36].

Recently, exploratory search has received increasing attention in the IR and database communities [1], [22], [25], [30]. In the database community, a growing number of efforts have focused on building search and exploration frameworks for *structured* data (*e.g.,* relational) [11], [15], [17], [33]. However, there is a dearth of work for realizing such search paradigm on graph-structured data. Consider the following scenario.

Example 1: The recent spate of chemical attacks in Syria, UK, and Malaysia has piqued the interest of Alice, a chemist, to seek information about chemical structures of various nerve agents. She has access to a large database of chemical compounds and is only aware of the fact that non-carbamate nerve agents contain the PO2 structure. Hence, her initial query graph Q_1 is shown in Figure 1(a). While browsing the result matches (denoted as R_1) to Q_1, she observed that many nerve agents have a F or S atom attached to the P atom. Hence, she first modifies Q_1 to Q_2 (Figure 1(b)) by adding the F atom and executes it. She now observes that the results R_2 contain *Novichok-5* (a Novichok agent was used in the recent UK attack) and *RpRc-Soman* (a G-series category agent was used in Syria). The substructures contained in these two agents are shown in Figures 1(c) and 1(d), respectively. Hence, she reformulates Q_2 to Q_3 (Figure 1(c)) by adding the shaded nodes and associated edges and executes it. Alice can now view the chemical structures of various Novichok agents (*e.g., Novichok-5*, *Novichok-7*) in the results as many share this common substructure. Next, she modifies Q_3 by deleting

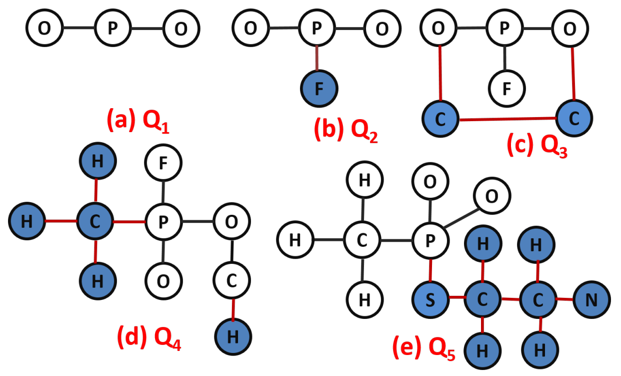


Fig. 1. **Motivating example.**

relevant edges and then reformulates it to Q_4 (Figure 1(d)) by adding the shaded nodes. Execution of this query retrieves the chemical structures of a set of G-series agents (*e.g., RpRc-Soman*, *SpRc-Soman*).

Satisfied, Alice now wishes to explore the nerve agents containing the S atom. She has noticed in R_1 that *Sp-VX*, a V-series agent (this type of nerve agent was used to kill Kim Jong Nam), contains the substructure depicted in Figure 1(e). Hence, she alters Q_4 to Q_5 (Figure 1(e)) by adding the highlighted nodes and associated edges. Execution of this query retrieves a set of V-series agents (*e.g., Sp-VX*, *Rp-VX*) that share this common substructure.

As Alice is a non-programmer, she demands a user-friendly visual interface that supports *direct manipulation-style interaction* [31][4] to formulate these search queries. In fact, direct-manipulation interfaces are already deployed by several commercial entities (*e.g., DrugBank*, *eMolecules*, *PubChem*) in this domain for lookup queries. Hence, there is a compelling need for a *visual exploratory subgraph search* (VESS) framework to easily formulate, reformulate, and execute subgraph queries iteratively and efficiently. ∎

The aforementioned scenario highlights several interesting characteristics of an exploratory subgraph search framework. First, it entails ways to formulate, reformulate, and process a query graph where multiple and iterative query formulation and execution are necessary. This guides users to learn about the underlying graph data and identify possible search directions beyond the initial query graph. Consequently, the initial query graph may often grow in size during exploration as users become familiar with the data space. For instance, in the above example, Q_1 grew from size 2 to size 14 (Q_5) during exploration. Second, it is paramount to provide a direct manipulation-based visual query interface (a.k.a GUI) for exploratory search in order to make it accessible to non-programmers. Since the query graph size may become large during exploration, it is desirable for such a GUI to expose *template patterns* (*i.e.,* small subgraphs) that can potentially decrease the time taken to visually construct a query. A template pattern (pattern for brevity) enables a user to construct multiple nodes and edges in a query graph 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). Furthermore, HCI research shows that users may become frustrated if a large number of small atomic actions (*e.g.,* repeated edge construction) are necessary to accomplish a higher-level task (*e.g.,* subgraph query) [31]. Naturally, template patterns may ease such frustration especially for constructing larger query graphs.

At first glance, it may seem that such exploratory search queries can be processed using any state-of-the-art traditional subgraph search techniques designed for a large collection of small- or medium-sized data graphs [27], [40]. However, given that the topologies of query graphs during iterative exploratory search often overlap significantly, this strategy is inefficient as each query is processed from scratch without leveraging on the computation results of previous queries (shown in Section VIII). For instance, in Example 1 the five query graphs will be evaluated from scratch by any traditional subgraph search technique.

PICASSO [13] is the first effort to crystallize visual exploratory subgraph search (VESS) on graph databases. It leverages the visual subgraph query processing framework of [18] to evaluate each formulation/reformulation incrementally by *blending* (*i.e.,* interleaving) visual query construction and processing. Our initial investigation, however, revealed that the visual subgraph query processor [18] of PICASSO is not efficient and scalable to support iterative query evaluation in a VESS environment as it was originally designed for lookup queries. This is especially so for the following two scenarios that are prevalent in a VESS environment (Example 1): (a) The initial query graph evolves to a large query during exploration, and (b) template patterns are added to a query graph during formulation/reformulation. These scenarios make the construction and maintenance costs of the online index deployed in PICASSO prohibitively expensive. Note that PICASSO constructs an online index called SPIG [18] for *each* newly constructed edge at *each* formulation step within the available GUI *latency* (*i.e.,* time to construct a query edge or template pattern visually). That is, a query graph with n edges creates n SPIGs in PICASSO. Although this is effective in a lookup querying environment where query graphs are typically small (*i.e.,* n is small) in practice [3], the VESS environment demands efficient blending of a larger set of edges (*e.g.,* template patterns) within *similar* GUI latency to support larger query graphs. Unfortunately, the time cost of creating and maintaining n SPIGs corresponding to a template pattern can be significantly higher than the available GUI latency, rendering the blending impractical in PICASSO (detailed in Section IV). This limitation also adversely impacts the scalability of the framework to deal with larger graph databases containing more than a million data graphs (*e.g., eMolecules*). Hence, it is important to design novel indexes that can efficiently support these two scenarios in the VESS paradigm.

In this paper, we present a VESS framework called FERRARI (**F**ramework for **E**xplo**R**atory subg**R**aph Se**A**rch in g**R**aph repos**I**tory) that embodies two novel indexing schemes, VACCINE and ADVISE, to address

[4]The central ideas in direct-manipulation interfaces are visibility of the objects and actions of interest; rapid, reversible, incremental actions; and replacement of typed commands by a pointing action on the object of interest [31].

the aforementioned challenges. VACCINE is an offline, *feature-based* index structure that leverages the widespread availability of modern machines with large memory to store richer information related to frequent and infrequent subgraphs (*i.e.,* fragments) in the underlying graph database and how they can be *transformed* from one fragment to another during visual query formulation. Note that such transformation information is not captured by existing indexes for traditional subgraph query processing [32], [39], [40] as query formulation activity (visual or textual) is orthogonal to the design goals in such environment. ADVISE, on the other hand, is an *adaptive* on-the-fly index designed to support incremental evaluation of subgraph queries in a VESS environment. It is instantiated during visual formulation/reformulation of a subgraph query by utilizing the VACCINE index. In contrast to a set of SPIGs in PICASSO, it is more compact and only one instance of ADVISE is necessary to support VESS. This enables us to blend template patterns within the available GUI latency efficiently, paving the way for handling larger queries in the VESS environment. Our experimental study with real-world datasets demonstrates that FERRARI is up to 4 orders of magnitude faster and more scalable than PICASSO in evaluating exploratory subgraph search queries. In summary, the main contributions of this paper are as follows.

- *A unified model.* We present a unified framework called FERRARI for visual exploratory subgraph search that can be instantiated into different existing VESS approaches.
- *A new offline index.* We present a novel offline index called VACCINE that unifies all the *frequent* and *discriminative infrequent fragments* in the underlying graph database in a single main memory-based data structure capturing rich information w.r.t the way a fragment can be visually formulated from another fragment in a VESS environment.
- *A new online index.* We present a novel adaptive, on-the-fly, space-efficient index called ADVISE to facilitate efficient pruning and retrieval of partial results during query formulation/reformulation in a VESS environment where template patterns may be used to formulate search intentions. Using these indexes, we show how iterative subgraph query evaluation is realized in FERRARI.
- *Experimental study.* Using real-world datasets containing up to 1.3 million data graphs, we show the superiority of our indexing schemes in supporting VESS compared to state-of-the-art techniques.

For clarity, we distinguish between a node in a query/ data graph and a node in indexes by using the terms “node” and “vertex”, respectively. We use v and its variants, v_1, v_2, $\ldots$, to denote nodes in the former and n and m along with their variants to denote vertices in VACCINE and ADVISE indexes, respectively.

The rest of the paper is organized as follows. We introduce concepts related to this work in Section II. We formally present the VESS problem and the FERRARI framework in Section III. We motivate the need for designing novel indexes in FERRARI to efficiently support template patterns in Section IV. Sections V and VI describe the VACCINE and ADVISE indexes, respectively. Section VII narrates how these indexes are exploited in FERRARI to evaluate subgraph queries iteratively in a VESS environment. We present experimental results in Section VIII. Section IX presents related work. Section X concludes the paper. Proofs of theorems and lemmas are given in Appendix A.

II. Background

We begin by introducing some key graph terminologies that we shall use in this paper. Next, we present the GUI actions for formulating visual subgraph queries. Lastly, we briefly give an overview of PICASSO [13], a state-of-the-art framework for VESS.

A. Terminology

We denote a graph as $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of (directed or undirected) edges. A node/vertex in G has an identifier j and is referred to as $v_j \in V$. Nodes and edges can have labels as attributes. We assume that G is a connected graph with at least one edge. The *size* of G is defined as $|G| = |E|$. For ease of presentation, we present data graphs and visual subgraph queries using undirected simple graphs with labeled nodes. The label of node $v \in V$ is denoted as $\ell(v)$ (ℓ when the context is clear).

A graph G is a *subgraph* of another graph G' if there exists a subgraph isomorphism from G to G', denoted by $G \subseteq G'$. In other words, G' is a *supergraph* of G ($G' \supseteq G$). We may also simply call that G' *contains* G. The graph G is called a *proper subgraph* of G', denoted as $G \subset G'$, if and only if $G \subseteq G'$ and $G \not\supseteq G'$.

B. Frequent and Infrequent Features

In this paper, we focus on a graph database or repository containing a large collection of small- or medium-sized data graphs (*e.g.,* chemical compounds, protein structure). Given such a graph database $\mathcal{D}$, we assign a unique identifier (*i.e.,* id) to each data graph in $\mathcal{D}$. A data graph G with id i is denoted as G_i. Let g be a subgraph of $G_i \in \mathcal{D}$ ($0 < i \leq |\mathcal{D}|$) that has at least one edge. Then, g is a *fragment* in $\mathcal{D}$. Informally, we use the term *fragment* to refer to a small subgraph in a data graph or a query graph. Given a fragment $g \subseteq G$ and $G \in \mathcal{D}$, G is referred to as the *fragment support graph* (FSG) of g [18]. We denote the set of FSGs of g as $\mathcal{D}_g$. We refer to $|\mathcal{D}_g|$ as (*absolute*) *support*, denoted by $sup(g)$. We denote the set of identifiers of the data graphs in $\mathcal{D}_g$ as $fsgIds(g)$. Note that in this paper we shall refer to a fragment in a query graph as *query fragment* in order to distinguish it from a fragment in a data graph.

A fragment $g \in \mathcal{D}$ is *frequent* if $sup(g) \geq \alpha|\mathcal{D}|$ where α is the minimum support threshold. We denote the set of frequent fragments in $\mathcal{D}$ as $\mathcal{F}$. We refer to a frequent fragment g as *frequent edge* if $|g| = 1$. On the other hand, if $sup(g) < \alpha|\mathcal{D}|$, then g is an *infrequent* fragment. We denote the set of infrequent fragments in $\mathcal{D}$ as $\mathcal{I}$. Specifically, we classify infrequent fragments into two types, *discriminative* and *non-discriminative* [18], [19]. Given $g \in \mathcal{I}$, let $sub(g)$ be the set of all proper subgraphs of g. If $sub(g) \subset \mathcal{F}$ or $|g| = 1$ (*i.e.,* if

g is an infrequent edge), then g is a *discriminative infrequent fragment* (DIF) in $\mathcal{D}$. We denote a set of DIFs in $\mathcal{D}$ as $\mathcal{I}_d$. Likewise, we refer to an infrequent fragment that is not a DIF as *non-discriminative infrequent fragment* (NIF). Note that if one of the subgraphs of g is a DIF, then g is an infrequent fragment [18].

C. GUI Actions for Visual Subgraph Query Formulation

We introduce a set of GUI *actions* (*actions* for brevity) that a user takes to formulate a subgraph query in any direct manipulation-based visual subgraph query interface.

- add(q,g): The add action denotes a user adding a query fragment (an edge or a template pattern) or a node g to an existing query graph q, and returns the augmented query.
- modify(q,g): This action denotes that a user revokes (deletes) a query fragment or a node g, and returns the modified query graph.
- run(q): The run action models the execution of the visually formulated query fragment by clicking on the Run icon (or equivalent of Run) in a GUI. Note that a user may invoke the run action multiple times in an exploratory search.

We refer to a sequence of such GUI actions taken by a user as *exploration action sequence* (EAS). Reconsider the exploratory search in Example 1 from Q_1 to Q_4. Here, an EAS, denoted by $\mathbb{A}$, can be: [add(Q_1, P-O), add(Q_1, P-O), run(Q_1), add(Q_2, P-F), run(Q_2), add(Q_3, C-O), add(Q_3, C-C), add(Q_3, C-O), run(Q_3), modify(Q_3, C-C), modify(Q_3, C-O), add(Q_4, CH3), add(Q_4, C-P), add(Q_4, C-H), run(Q_4)]. Observe that the add and modify actions precede a run action and are used to construct a query graph. We refer to this sequence of add and modify actions preceding a run action or between a pair of run actions as *query formulation sequence* (QFS). Hence, $\mathbb{A}$ can be expressed as follows: [QFS(Q_1), run(Q_1), QFS(Q_2), run(Q_2), QFS(Q_3), run(Q_3), QFS(Q_4), run(Q_4)].

Remark. We remark that we do not model low-level operations (*e.g.,* mouse click, mouse hover, drag-and-drop) as different GUIs may follow different sequence of low-level operations to realize the aforementioned three GUI actions. Consequently, this enables us to design a visual exploratory subgraph search framework that *underpins any* GUI. For example, in one GUI addition of a node may be simply realized by right clicking on an empty space of the *Query Panel* followed by addition of a label, whereas in another interface this action may require selection of a node label from the *Attribute Panel* and dragging-and-dropping it on the *Query Panel* (*e.g.,* Appendix A). Hence, we model GUI actions at a higher level of abstraction.

D. PICASSO

PICASSO [13] leverages the visual subgraph query processing framework of [18] to realize VESS. It first generates *action-aware frequent (*A^2F*) index* and *action-aware infrequent (*A^2I*) index* from the underlying graph database $\mathcal{D}$ offline to index frequent subgraphs and DIFs, respectively. The A^2F index is a graph-structured index that enables efficient retrieval of FSG identifiers of a given frequent fragment. The A^2I-index is an array of DIFs and associated information. It facilitates pruning of the candidate space for infrequent queries.

When a user adds a new edge or a template pattern p to the current query fragment q (initially empty), the edges of p are first ordered such that q will remain a connected subgraph when it is augmented with each edge in the specified order iteratively. Next, these edges are added to q and a dynamic on-the-fly index called SPIG *set* [18] is constructed. For each new edge, PICASSO retrieves identifiers of data graphs containing q (denoted by R_q) by exploiting the indexes. If q is a frequent fragment or a DIF, then it retrieves $fsgIds(q)$ by probing A^2F-index or A^2I-index, respectively. If q is a NIF, it leverages the SPIG set and the action-aware indexes to generate R_q. This is possible as a NIF must contain at least one DIF.

If R_q is non-empty at a specific step, then q has exact matches in $\mathcal{D}$. On the other hand, if R_q becomes empty, then q has evolved into a similarity search query and candidates that *approximately* match q (based on *maximum connected common subgraph* (MCCS) [32]) are retrieved using the SPIG set by identifying relevant subgraphs of q that need to be matched for retrieving candidates. In the case of deletion of an edge or a query fragment during query reformulation, the SPIG set is updated by removing information related to it. Then, depending on the status of the modified query fragment (*i.e.,* frequent, DIF, or NIF), R_q is updated.

Whenever the Run button is clicked, the current query fragment q is processed to retrieve result matches R by leveraging R_q. If q is a frequent fragment or a DIF, then R is directly computed from R_q without performing subgraph isomorphism test. If it is a NIF, then the exact results are computed by filtering the false candidates using VF2 [5]. Otherwise, if q has evolved to a subgraph similarity query then it generates R from the candidates by extending VF2 to handle MCCS-based similarity verification [18]. Since in an exploratory search a query may be executed several times, for each run the (q, R) pair is stored to generate a *search stream.* PICASSO utilizes these search streams to generate a *multi-stream results exploration wall* where results of initial and reformulated query graphs are juxtaposed in form of parallel (q, R) pairs to facilitate its exploration. This iterative process continues until the exploratory search is terminated.

III. A Unified Framework For VESS

We now formally introduce the *visual exploratory subgraph search (*VESS*) problem* and then present a unified framework called FERRARI to address it.

A. The VESS Problem

Intuitively, in a VESS environment, a user typically undertakes EAS involving multiple runs. After each run(q) a user may browse and explore the results of q before modifying it again. A core challenge in realizing such VESS framework is to devise efficient and scalable techniques for evaluating run(q) to facilitate real-time exploration of results of q. **In this paper, we focus on devising indexing schemes to efficiently support this iterative run(q) action.**

At each run(q), it is imperative for our framework to support *fast* subgraph containment or subgraph similarity search of q. In particular, similar to [13], [18], [32], we adopt maximum connected common subgraphs (MCCS)-based *subgraph similarity distance* to measure similarity between a data graph and a query graph. Given two graphs G_1 and G_2, the MCCS of G_1 and G_2 is the largest connected subgraph of G_1 that is subgraph isomorphic to G_2, denoted as $mccs(G_1, G_2)$. We define the *subgraph similarity distance* between a query graph Q and a data graph G as $dist(G, Q) = |Q| - |mccs(Q, G)|$ [13], [32]. Observe that smaller the $dist(G, Q)$, more similar are Q and G.

Definition 1: Let $\mathbb{A}$ be an EAS *undertaken by a user on a visual query interface for exploring a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_n\}$. Then the goal of* ***visual exploratory subgraph search (*VESS*) problem*** *is to retrieve all the graphs $g_i \in \mathcal{D}$ with $dist(g_i, q) \leq \delta$ for each run(q) $\in \mathbb{A}$ where δ is the subgraph similarity distance threshold.*

B. The FERRARI Framework

Algorithm 1 outlines the FERRARI framework for *each* action in an EAS. It utilizes the direct manipulation-based visual interface of PICASSO for query formulation and results exploration (Appendix A). Similar to PICASSO, FERRARI interleaves (*i.e.,* blends) the formulation and processing of a query fragment so that it does not need to evaluate each run(q) action from scratch. To this end, we assume that an offline index I_O has been constructed from $\mathcal{D}$ prior to formulation of any query. Let q be a visual query fragment formulated by a user during exploratory search. Let $simFlag$ be a Boolean variable to indicate if q is a subgraph similarity query or a containment query (*true* or *false*, respectively). The framework monitors four actions, namely `NewEdgeSet` for adding a set of edges to q (*i.e.,* add(q,g)), `SimQuery` for invoking subgraph similarity search, `Modify` for removing an existing edge in q (*i.e.,* modify(q,g)), and `Run` for executing the current query fragment (*i.e.,* run(q)).

When a user adds a new edge set $eStr$ to q, the algorithm constructs and maintains an adaptive online index I_L for the edge set (Line 5). If q has exact subgraph matches (*i.e.,* $simFlag$ is $false$), then it retrieves the candidates of q (stored in R_q) by leveraging the offline and online indexes (Line 7). If R_q is empty, then it means that there is no exact match for q after the addition of e. Consequently, the user can either modify q or retrieve similar matches to q (Lines 8–9). If the latter is chosen, then q is regarded as a subgraph similarity query and corresponding candidate matches are retrieved (Lines 12–14) by leveraging the indexes. On the other hand, if the former is chosen, then a user-selected edge is removed and I_L is updated (Lines 15–16)[5]. If the user clicks the `Run` button, then the constructed query q is processed to retrieve result matches (Lines 17–18). If q is a subgraph containment query, the exact results $Results$ will be returned after conducting candidates verification (*i.e.,* subgraph isomorphism test), if necessary, on R_q. Otherwise, if

[5]We can easily extend it to handle deletion of a set of edges (*e.g.,* template pattern) by iteratively updating I_L.

Algorithm 1: The FERRARI Framework

```
Input: GUI action $\in \mathbb{A}$, a boolean variable $simFlag$, a
       query fragment $q$, a candidate set $R_q$, a subgraph
       distance threshold $\delta$, an offline index $I_O$, a graph
       database $\mathcal{D}$.
Output: Query results $Results$, $q$, $R_q$, $simFlag$, an online
        index $I_L$.
1  Initialize $eStr \leftarrow \emptyset$
2  if action is NewEdgeSet $eStr$ then
3      for edge $e \in eStr$ do
4          $q \leftarrow q + e$
5          $I_L \leftarrow$ ConstructOnlineIndex($I_O$, $I_L$, $q$, $e$)
6          if $simFlag = false$ then
7              $R_q \leftarrow$ ExactCandidates($I_O$, $I_L$, $q$)
8              if $R_q = \emptyset$ then
9                  action $\leftarrow$ Modify $q$ or invoke similarity
                   search
10         else
11             $R_q \leftarrow$ SimilarCandidates($I_O$, $I_L$, $q$, $\delta$)
12 else if action is SimQuery then
13     $simFlag \leftarrow true$
14     $R_q \leftarrow$ SimilarCandidates($I_O$, $I_L$, $q$, $\delta$)
15 else if action is Modify with edge $e$ then
16     Update $I_L$
17 else if action is Run then
18     $Results \leftarrow$ RetrieveResults($I_O$, $I_L$, $q$, $\delta$)
19 return $q$, $Results$, $R_q$, $simFlag$, $I_L$
```

it is already a subgraph similarity query (*i.e.,* $simFlag$ is true), then data graphs that match q approximately are returned. Note that although FERRARI utilizes the multi-stream results exploration wall of PICASSO to visualize and explore query-results pairs, the indexing framework is not tightly coupled to any specific results exploration framework. Hence, this work is orthogonal to the results exploration problem and any superior exploration interface can be used on top of FERRARI.

Observe that the expensive candidate verification step is performed only after the `Run` button is clicked and not during visual construction of query fragments. Furthermore, FERRARI allows a user to execute a query fragment any time during query formulation facilitating exploratory search. In particular, every run(q) action exploits I_L from the preceding step to evaluate q instead of evaluating it from scratch (Line 18).

Generality of the framework. The VESS framework is general and can be instantiated into different algorithms as follows. In PICASSO, the offline index I_O comprises of A^{2}F and A^{2}I indexes, whereas in this work we use an index called VACCINE (Section V). Line 5 can adopt different online indexing schemes. Particularly, in PICASSO, I_L is instantiated to a SPIG index for *each* new edge. Instead, we instantiate it with an efficient and concise index called ADVISE (Section VI) that can handle a set of edges efficiently. The generation of exact and similar candidates in Lines 7, 11, and 14 can also leverage different strategies. In PICASSO, the offline and online indexes are exploited to compute them. In this work, we also use similar strategies but utilize VACCINE and ADVISE instead. Line 16 can adopt different modification policies. In PICASSO, a set of online indexes (*i.e.,* SPIGs) is modified whereas in this work the ADVISE index is updated. Line 18 also can be implemented using different subgraph verification techniques. Lastly, we assume MCCS-based similarity measure for computing similarity between data graphs and query fragments (*e.g.,* Lines 11, 18). These procedures as well as I_L can also be redesigned for different similarity measures.

IV. The Impact of Template Patterns

In this section, we justify the need to design novel index structures in FERRARI in order to handle template patterns efficiently in a VESS environment. We first quantify the GUI latency available for processing template patterns. Then, we present the limitations of PICASSO in efficiently processing them within the GUI latency.

A *pattern-based visual interface* (PI) exposes a set of template patterns to enable query formulation. Let a pattern p is added to a query fragment (denoted as q_i) at ith step during visual query formulation. Then q_i is processed (*i.e.,* construction and maintenance of online indexes) by utilizing the GUI latency available due to the *successor* action (Lines 2-14 and 15-16 in Algorithm 1). How much is this latency? The successor action (*i.e.,* add action in the $(i+1)$th step) can be a construction of a single edge (either between an existing pair of nodes in q_i or with a new node) or a pattern. In particular, addition of a pattern (resp. edge with a new node) in FERRARI and PICASSO involves the following steps that contribute to the GUI latency (Appendix A).

1) Move the mouse cursor to the *Pattern Panel* (resp. *Attribute Panel*).
2) Scan and select a pattern (resp. label).
3) Drag the selected pattern (resp. label) to the *Query Panel* and drop it.
4) Construct edges (if necessary) between relevant nodes by clicking on them.

Let us refer to the times taken to complete Steps 1, 2, 3, and 4 as *movement time* (denoted by T_m), *selection time* (T_s), *drag time* (T_d), and *edge construction time* (T_e), respectively.

A template pattern p is added to a query fragment in two possible ways. First, p is dropped on an edge of the existing query fragment q, which transforms the modified query to a *connected* graph. In this case, the latency available is as follows:

$$T_{\ell_{p1}} \quad = \quad T_m + T_s + T_d \qquad (1)$$

Second, p is dropped on the *Query Panel* as a separate component from q, and subsequently, an edge is added to connect it with q (a query is processed only when it is a connected graph). Notably, this same principle is used for creating an edge with a new node. In this case,

$$T_{\ell_{p2}} = T_m + T_s + T_d + T_e \qquad (2)$$

Note that any edits to p only increase the latency. Hence, the *minimum* latency available to process q_i with pattern p is $T_{\ell_{min}} = min(T_{\ell_{p1}}, T_{\ell_{p2}}, T_e)$ where T_e is the time to create an edge between existing nodes in q_i.

Limitations of indexing schemes in PICASSO. Observe that we need to process n edges of a pattern within $T_{\ell_{min}}$ time.

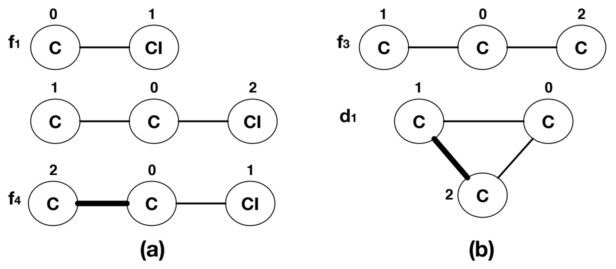


Fig. 2. **Examples of node and edge transformers.**

Hence, PICASSO needs to construct n SPIGs in $T_{\ell_{min}}$ time. That is, it has on average $T_{\ell_{min}}/n$ time to construct a SPIG, which is significantly lower than $T_{\ell_{min}}$ especially when n increases. Unfortunately, as reported in Section VIII, it becomes impractical to construct and maintain the SPIG set within the available GUI latency during exploratory search. This justifies the need for more efficient indexing schemes in order to effectively handle the impact of formulation of larger queries using template patterns in the VESS environment. Ideally, we should construct and maintain *only one* online index (instead of $|q|$ SPIGs for query q) within $T_{\ell_{min}}$ regardless of the number of edges added by an add action. In the subsequent sections, we shall introduce our indexing schemes to realize this goal.

V. VACCINE Index

In this section, we describe our offline index called VACCINE (**V**isual **AC**tion-**C**onscious **IN**tegrated f**E**ature index). We begin by introducing two *primitive transformers* that we shall be using for constructing the VACCINE index. Next, we describe the structure of the index. Finally, we elaborate on the algorithm to construct VACCINE from the underlying data graphs.

A. Primitive Transformers

A *primitive transformer* (transformer for brevity) transforms a frequent fragment or a DIF g to another fragment g' by adding a new node or an edge such that $|g'| - |g| = 1$. Specifically, we use two types of transformer, namely *node* and *edge* transformers. Given a fragment $g = (V, E)$, the *node transformer*, denoted as $\Psi_g(i, \ell)$, transforms g to a new graph $g' = (V', E')$ by adding a frequent edge from $v_i \in V$ to a new node $v_{new} \notin V$ with label $\ell(v_{new})$. That is, $(v, v_{new}) \in E'$. On the other hand, the *edge transformer*, denoted as $\Phi_g(i, j)$, transforms g to g' by adding an edge between two non-adjacent nodes $v_i \in V$ and $v_j \in V$. That is, $(v_i, v_j) \notin E$ but $(v_i, v_j) \in E'$.

Example 2: Consider the graphs f_1 and f_4 in Figure 2(a). Assume that these fragments are frequent or DIF. The number above a node in these two graphs is the identifier of the node. Then f_4 is generated from f_1 by utilizing $\Psi_{f_1}(0, \text{C})$. Specifically, we add a new frequent edge (v_0, v_2) (shown in bold) in f_1 to transform it to f_4. Now consider the fragments f_3 and d_1 in Figure 2(b). In this case, d_1 is generated from f_3 by utilizing the edge transformer $\Phi_{f_3}(1, 2)$, which adds an edge between two non-adjacent nodes v_1 and v_2 in f_3 (shown in bold). ∎

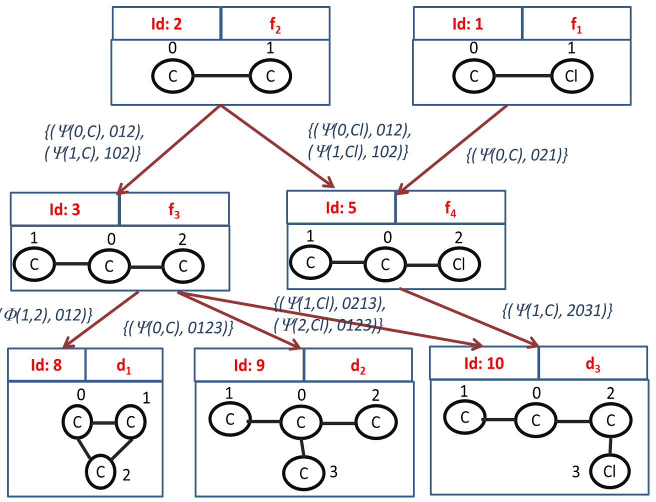


Fig. 3. **VACCINE index. For clarity, we do not show the CAM code and $\mathcal{L}_g$ associated with each vertex.**

Remark. Observe that these transformations can be used to simulate the way a new edge in a query graph can be constructed during visual query formulation. Specifically, the node transformer simulates construction of a new node and associated edge, whereas the edge transformer represents construction of an edge between existing nodes in a query graph. For example, suppose a user has first constructed f_1 in Figure 2 during visual exploratory search. Next, she augments her query by adding a node C to the existing C atom (f_4). Observe that this add action is simulated by the node transformer $\Psi_{f_1}(0, \text{C})$. In the next step, suppose she adds an edge between the non-adjacent Cl and C nodes (nodes 1 and 2 in f_4). This action is represented by the edge transformer $\Phi_{f_4}(1, 2)$. In summary, these transformers capture the add(q,g) action taken by a user in a QFS where g is an edge.

B. Structure of VACCINE

Given a graph database $\mathcal{D}$ and a minimum support threshold α, the VACCINE index is a directed acyclic graph $G_I = (V_I, E_I)$. Each vertex $n \in V_I$ represents a frequent fragment or a DIF g in $\mathcal{D}$ (*i.e.,* $g \in \mathcal{F}$ or $g \in \mathcal{I}_d$). We refer to the fragment represented by n as $n.g$ (n when the context is clear). Each $n \in V_I$ is a 4-tuple $v = (id, c, cam(g), \mathcal{L}_g)$, where id is its unique identifier, c is its category (*i.e.,* frequent or DIF), $cam(g)$ is the CAM code[6] [12] of g, and $\mathcal{L}_g$ is a set of identifiers of the data graphs which are subgraph isomorphic to g (*i.e.,* $\forall g'_i \in \mathcal{L}_g, g \subseteq g'_i$). Note that as the number of infrequent fragments (both DIFs and NIFs) can be prohibitively large and a NIF must contain at least one DIF (recall from Section II-B), VACCINE only stores the frequent and DIF fragments.

The edges between vertices, E_I, model the relationships between fragments represented by these vertices using the primitive transformers. Specifically, an edge $(n_1, n_2) \in E_I$ is labeled with a set of 2-tuple $(\tau, \mathcal{C})$ elements, denoted by $\mathbb{T}_{1,2} = \{(\tau_1, \mathcal{C}_1), (\tau_2, \mathcal{C}_2) \ldots (\tau_k, \mathcal{C}_k)\}$ ($\mathbb{T}$ when the context is clear), where $\tau \in \{\Psi, \Phi\}$ represents the node or edge transformer to transform the fragment $n_1.g$ to $n_2.g$ and $\mathcal{C}$ is the canonical labeling[7] of $n_2.g$, which is computed using *Nauty* [26] by utilizing $n_1.g$ and primitive transformers. Note that all graphs that are isomorphic to each other share same CAM code and canonical labeling. Consider the two 3-nodes graphs in Figure 2(a). The canonical labeling from the middle graph to f_4 is `(0,2,1)` (*i.e.,* 0, 1, and 2 in the middle graph are mapped to nodes 0, 2, and 1 in f_4, respectively). Observe that there can be more than one way to transform $n_1.g$ to $n_2.g$ by utilizing the transformers. Hence, for each edge (n_1, n_2), $\mathbb{T}$ captures all transformations that can generate $n_2.g$ from $n_1.g$. Furthermore, $|n_2.g| - |n_1.g| = 1$ and $n_1.g \subset n_2.g$. We refer to n_1 (resp. n_2) as the *parent* (resp. *child*) of n_2 (resp. n_1). Also, we refer to a vertex $n \in V_I$ as a *root* (resp. *leaf*) if it has no incoming (resp. outgoing) edges.

Example 3: Reconsider the fragments f_1 and f_4 in Figure 2(a). We index them in the VACCINE index with two vertices, n_1 (representing f_1) and n_2 (representing f_4), and an edge (n_1, n_2). In this case, $\mathbb{T}_{1,2} = \{(\Psi_{f_1}(0, \text{C}),$ `(0,2,1)`$)\}$, where $\Psi_{f_1}(\cdot)$ is the node transformer for transforming f_1 to f_4 and `(0,2,1)` is the canonical labeling. Similarly, for the frequent fragment f_3 and DIF d_1 in Figure 2(b), two vertices, n_3 and n_4, are created for them, respectively. In this case, $\mathbb{T}_{3,4} = \{(\Phi_{f_3}(1, 2),$ `(0,1,2)`$)\}$, where $\Phi(\cdot)$ is the edge transformer and `(0,1,2)` is the canonical labeling. An instance of the VACCINE index is shown in Figure 3. Observe that an edge can be labeled with more than one $(\tau, \mathcal{C})$ pairs. ∎

How can VACCINE facilitate visual exploratory search? Recall that (Section II-B) an edge in $\mathcal{D}$ is either a frequent edge or a DIF. Hence, these edges are indexed in VACCINE. Since edges are iteratively added to a visual query (either manually by a user or automatically when a template pattern is added to a query fragment), we always begin with a frequent edge or an infrequent edge (*i.e.,* DIF). Progressively, the query fragment may remain frequent or DIF or morph to a NIF due to sequence of add actions during the query formulation process. The edges of VACCINE efficiently capture this sequence of add actions, thereby capturing possible users' actions during exploratory search. For example, consider the first three query fragments in Figure 4 (top row) during an exploratory search. Here, the EAS $\mathbb{A}$ is: [add(Q_1, C-C), add(Q_2, C-Cl), add(Q_3, C-C)]. Observe that the first fragment Q_1 is a root (id 2) in Figure 3. The second fragment Q_2 is a child of this vertex (id 5). The next fragment generated by the third add action is a child of vertex 5 (*i.e.,* $n.id = 10$). Note that in the case a user took the action add(Q_2, C-C) instead of add(Q_2, C-Cl), then it would match the vertex 3. Hence, the edges of VACCINE capture all possible add actions taken by a user during query

[6]Let a graph g is represented by an adjacency matrix M. Every diagonal entry of M is filled with the label of the corresponding node and every off diagonal entry is filled with 1 or 0 if there is no edge. The CAM code is formed by concatenating lower triangular entries of M, including the entries on the diagonal. The order is from top to bottom and from the leftmost entry to the rightmost entry. We choose the maximal code among all possible codes of a graph by lexicographic order as this graph's canonical code.

[7]Intuitively, canonical labeling is a process in which a graph is relabeled in such a way that isomorphic graphs are identical after relabeling. Hence, isomorphism testing on two graphs can be performed by simply comparing their canonical labeling.

formulation that result in frequent or DIF fragments. Observe that the modify action (*i.e.,* deletion of a fragment) is also captured by VACCINE as the resultant query fragment due to this action is essentially a fragment due to a previous add action. Since each vertex in the index is associated with the FSG identifiers of data graphs containing the corresponding fragment, we can find data graphs matching a current query fragment easily during VESS for any visual action taken by a user. This facilitates evaluation of the run action at any point during exploration.

C. Index Construction

Given $\mathcal{D}$ and α, our idea is to first fetch all frequent edges from $\mathcal{D}$ into the VACCINE index G_I and then index other frequent fragments and DIFs by transforming the frequent edges using the primitive transformers. Lastly, edges between these fragments are created in VACCINE based on their formulation relationships using the transformers.

Algorithm 2 outlines the procedure for constructing the VACCINE index. First, it fetches all frequent fragments $\mathcal{F}$ in the graph database $\mathcal{D}$ (Line 1). Each fragment $f \in \mathcal{F}$ is associated with its $fsgIds(f)$. Next, we obtain all frequent edges (*i.e.,* frequent fragments with only one edge) from $\mathcal{F}$ and store them in a matrix fe (Lines 4-8). These edges are considered as seeds for building the VACCINE index. Each element of the matrix is a Boolean value to represent whether the edge is frequent or not. For example, if $fe[\mathtt{C}][\mathtt{C}]$ is true, then $(\mathtt{C}, \mathtt{C})$ is a frequent edge. All infrequent edges are recorded as DIFs in the VACCINE index (Line 10).

Next, the algorithm iterates through all $f \in \mathcal{F}$ (*i.e.,* all frequent fragments) and creates a vertex n_f in the index G_I corresponding to f where $\mathcal{L}_f = fsgIds(f)$. It builds the index further by applying the primitive transformers on each frequent fragment f (Lines 13-14). These transformation operations add a frequent edge to f at all possible locations and check whether the transformed graph f' is a frequent fragment or a DIF. If it is, then it creates an edge from n_f to $n_{f'}$ to capture the relationship between them based on the corresponding primitive transformers. Observe that $|f'| - |f| = 1$. We now elaborate on the procedures to realize these transformers.

NODETRANSFORMER(): Similar to the subgraph extension process in several frequent subgraph mining techniques [8], [38], given a frequent fragment $f = (V, E)$ in G_I, it goes through each vertex $v_f \in V$ and adds all possible frequent edges to it. The fe matrix is utilized to find relevant frequent edges for adding to v. Addition of a frequent edge to f results in a new fragment f', and this transformation operation is recorded.

EDGETRANSFORMER(): This procedure creates all fragments generated by the edge transformer and records the information in VACCINE. Specifically, it traverses through every pair of nodes (v_i, v_j) in the frequent fragment f and checks if an edge exists between them. If an edge does not exist, then it adds a new frequent edge between these two nodes using the edge transformer $\Phi_g(i, j)$ to create a new fragment f'. This edge transformation operation is recorded.

After this, the newly created fragment f' is processed by the PROCESSNEWGRAPH procedure as follows. Note that

Algorithm 2: VACCINECONSTRUCT()

```
Input: A dataset of graphs $\mathcal{D}$, a minimum support threshold $\alpha$.
Output: $G_I$, the VACCINE index of $\mathcal{D}$.
1  $\mathcal{F} \leftarrow$ frequent fragments of $\mathcal{D}$ with $\alpha$
2  $E \leftarrow$ All edges in $\mathcal{D}$
3  $G_I \leftarrow \emptyset$
4  $fe[label][label] \leftarrow \emptyset$
5  for $edge\ e \in E$ do
6      if $e \in \mathcal{F}$ then
7          $fe[e.srcLabel][e.trgLabel] \leftarrow true$
8          $fe[e.trgLabel][e.srcLabel] \leftarrow true$
9      else
10         record $e$ as a DIF edge into $G_I$
11 for $f \in \mathcal{F}$ do
12     Assign $f$ into $G_I$ as vertex $n_f$
13     NODETRANSFORMER($f, G_I, fe$)
14     EDGETRANSFORMER($f, G_I, fe$)
15 for $n \in V_I, n \in \mathcal{I}_d, n.\mathcal{L}_g = \emptyset$ do
16     $C \leftarrow \bigcap_{p \in n.parent} p.\mathcal{L}_g$
17     $n.\mathcal{L}_g \leftarrow C$
```

this procedure is invoked by the NODETRANSFORMER and EDGETRANSFORMER procedures.

PROCESSNEWGRAPH(): Algorithm 3 describes this procedure. It adds a new graph f' generated from f into the current VACCINE index if f' is a frequent fragment (Line 1) or a DIF (Line 6). Also, it adds a directed edge $(n_f, n_{f'})$ and computes the canonical labeling of (f, f'). Lastly, it annotates the edge with $\mathbb{T}$ (Lines 4-5 and 9-10) to capture the transformation relationships between f and f'. To check whether a transformed fragment is frequent, we only need to compare it with all frequent fragments in $\mathcal{F}$. That is, to find all s (Algorithm 3), we iterate through all subgraphs containing the new edge added by the primitive transformers to check whether they are frequent fragments.

Algorithm 3: PROCESSNEWGRAPH()

```
Input: Frequent fragment $f$, the VACCINE index $G_I$, a transformer $\tau \in \{\Phi, \Psi\}$.
1  if $f' \in \mathcal{F}$ then
2      Add $f'$ into $G_I$ as a vertex $n_{f'}$
3      Add a new edge $(n_f, n_{f'})$ in $G_I$
4      $C \leftarrow$ canonical labeling of $(f, f')$
5      $\mathbb{T}.add((\tau, C))$
6  else if $\forall s \subset f', s \in \mathcal{F}$ then
7      Add $f'$ into $G_I$ as a DIF vertex $n_{f'}$
8      Add a new edge $(n_f, n_{f'})$ in $G_I$
9      $C \leftarrow$ canonical labeling of $(f, f')$
10     $\mathbb{T}.add((\tau, C))$
```

Observe that after traversing through all $f \in \mathcal{F}$ using the aforementioned steps, G_I consists of frequent fragment fragments and DIFs. Particularly, all leaf vertices of the VACCINE index represent DIFs or frequent fragments. At this point, data graphs associated with a DIF vertex have not been recorded yet because such information is not contained in $\mathcal{F}$. Hence, in the final step the algorithm adds this information as follows

(Lines 15–17, Algorithm 2). For a given vertex and its parent in the VACCCINE index, the union of the data graph identifiers associated with the vertex and its parent is the same as those of the parent. Thus, to complete the index, the algorithm computes the intersection of the data graph identifier sets of all parent vertices of a given vertex n of a DIF ($\mathcal{I}_d$) to generate its data graph identifier set $\mathcal{L}_g$. Recall that subgraphs of a DIF are frequent fragments.

Example 4: Suppose there are two frequent edges, (C,Cl) (denoted as f_1) and (C,C) (denoted as f_2), in $\mathcal{D}$. Figure 3 depicts a portion of the VACCINE index rooted at f_1 and f_2. We illustrate how it is constructed by Algorithm 2. First, for each node in f_2, it adds relevant frequent edges to it. The new fragment f_3 is generated when the frequent edge (C,C) is added to node 0 in f_2. Since f_3 is a frequent fragment, we add it to the index, connect it to its parent vertex f_2, and record the transformer and canonical labeling information. Similarly, another frequent edge f_1 is added to generate the fragment f_4. Note that after the edge (C,C) is connected to node 1 in f_2, the new fragment is isomorphic to f_3. Hence, the algorithm only adds the transformation information in $\mathbb{T}$. Subsequently, d_1 is inserted to the index after it is generated using edge transformer on f_3. Since d_1 is a DIF, it is not necessary to generate successive infrequent fragments from d_1 by adding frequent edges. Lastly, the $fsgIds$ of d_1 is computed from its parent f_3. ∎

Lemma 1: Suppose N is the number of frequent fragments and DIF*s, N_{fe} is the number of frequent edges in $\mathcal{D}$, and N_{fmax} is the maximum number of nodes in a frequent fragment. Then a* VACCINE *index has $O(N(C^2_{N_{fmax}} + N_{fmax}N_{fe}))$ edges, where $C^2_{N_{fmax}}$ denotes the total number of different combinations of selecting two different nodes from a frequent fragment.*

Theorem 1: Given a dataset $\mathcal{D}$, suppose N_f is the number of frequent fragments. Let N_{dif} is the number of DIF*s, $\mathcal{E}$ is the set of all edges in $\mathcal{D}$, and $\mathcal{L}$ is the set of all labels in $\mathcal{D}$. Let the maximum number of nodes in a frequent fragment is denoted as N_{fmax} and the maximum number of edges in a* DIF *is denoted as N_{dmax}. Assume the time complexity of mining the frequent fragments from $\mathcal{D}$ is C_{ff}. Similarly, let the time complexity of canonical labeling process is C_{cl}. Then the time complexity for building* VACCINE *index is $O(C_{ff} + |\mathcal{E}| + N_f C_{cl}(N_{fmax}|\mathcal{L}| + N^2_{fmax}) + N_{dif}N_{dmax})$.*

Remark. The candidate subgraph generation step in frequent subgraph mining [8], [38] also extends subgraphs with frequent edges. Specifically, in frequent subgraph mining, the goal of subgraph extension is to check whether a subgraph is frequent by computing its frequency. In contrast, during our index construction process, we generate *both* frequent and DIF fragments and determine the transformation relationships between these fragments. Note that we deliberately resist to tightly integrate frequent subgraph mining technique and VACCINE construction steps in FERRARI. That is, building of transformation relationships between different graph patterns is not injected into the mining process. This design choice ensures the generality of our framework as VACCINE can be seamlessly integrated on top of *any* state-of-the-art frequent subgraph mining technique (*e.g.,* [8], [38]). Similarly, we can use any subgraph isomorphism testing algorithm in our framework.

Lastly, observe that we resist to build any index on the template patterns and as a result resort to process edges of a pattern one by one (Algorithm 1). Given a dataset $\mathcal{D}$, different GUIs may have different patterns. In some GUIs, the patterns are unlabeled graphs (*e.g., PubChem*), and in others, they may be labeled ones. Importantly, the number of template patterns and their sizes are typically small in practice (*i.e.,* users typically will not browse through a long list of patterns to formulate queries). Many of these patterns may also be frequent fragments or DIFs, which are already indexed by VACCINE. Furthermore, a user may modify these patterns on the *Query Panel* to formulate queries. By processing the edges one by one we can seamlessly handle these issues without building additional data structures for these patterns. This enables FERRARI to handle any GUI containing template patterns without building GUI-specific data structures that need to be updated every time the type and content of patterns in a GUI evolve.

D. Comparison with PICASSO

As reported in [18], [19], existing indexing schemes for traditional subgraph querying techniques [27], [39], [40] are not suitable for efficient query evaluation in this visual querying paradigm. Specifically, they do not store visual formulation relationships between different frequent and infrequent fragments to capture users' actions during VESS. This information is critical toward efficient support of the run action in VESS.

Most germane to the VACCINE index is the indexing scheme in PICASSO [13]. The former differs from the latter in the following ways. PICASSO utilizes the A^{2}F and A^{2}I indexes [18], [19] for exploratory subgraph search (recall from Section II-D). The A^{2}F index is a graph-structured index having a *memory-resident* and a *disk-resident* components called the MF-index and the DF-index, respectively. In contrast, VACCINE consolidates both frequent fragments and DIFs into a single memory-resident index. More importantly, action-aware indexes do not record the primitive transformers to connect these fragments. Specifically, the A^{2}F index records edges between two frequent subgraphs f_1 and f_2 if f_1 is a subgraph of f_2 and $|f_2| - |f_1| = 1$. The A^{2}I index is simply an array of DIFs arranged in ascending order of their sizes. Hence, VACCINE captures richer information related to how visual actions (add(q,g)) modify a query fragment (by storing transformer information). Besides, VACCINE does not have any A^{2}I index or its variant. DIFs are integrated with frequent fragments and are not simply stored in a sorted array. Consequently, the algorithm to construct VACCINE also differs from [13].

VI. ADVISE Index

In this section, we introduce a novel data structure called ADVISE (**AD**aptive **VI**sual **S**ubgraph **E**xploration) index, which is progressively constructed and maintained on-the-fly during add and modify actions in an EAS by utilizing the VACCINE index.

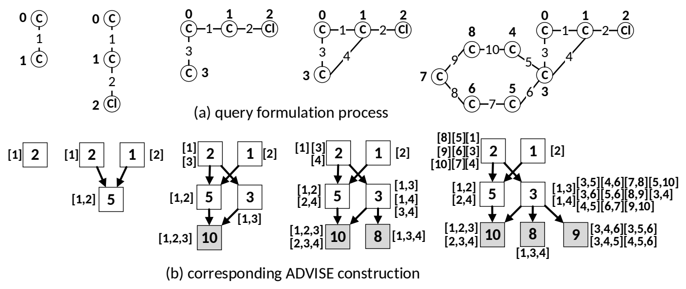

(a) query formulation process
(b) corresponding ADVISE construction

Fig. 4. **Example of ADVISE index construction for add action. The bottom row shows the ADVISE index for a set of add actions (top row). The node id is shown in rectangular box and its $\mathbb{E}_q$ is shown in square brackets. The vertices representing DIFs are colored in gray.**

A. *Definition*

The goal of ADVISE index is to progressively keep track of the candidate data graphs of an evolving visual query fragment q efficiently by concisely storing information related to all frequent fragments and DIFs contained in q at any point of time during exploration. As we shall see in Sections VII and VIII, this facilitates efficient execution of the run action anytime during exploratory search.

Intuitively, an ADVISE index is a directed graph where each vertex m represents a frequent or a DIF fragment in the current query q and each edge (m', m) represents the *containment* relationship between the fragments m and m' (*i.e.,* $f(m') \subset f(m)$ and $|f(m)| = |f(m')| + 1$ where $f(m)$: $m \rightarrow f_g$ s.t. $f_g \subseteq q$.). Each vertex m is associated with the identifier of the vertex n (*i.e., matching vertex*) in VACCINE containing the fragment that matches $f(m)$ and a set of edge id sets in q that matches the edges in $f(m)$. Formally, it is defined as follows.

Definition 2: Let $q = (V_q, E_q)$ be a visual graph query fragment during exploratory search on a graph database $\mathcal{D}$ and contains ℓ edges with ids $1, 2, \ldots \ell$. Let $G_I = (V_I, E_I)$ be the VACCINE *index on $\mathcal{D}$. Then, the* ADVISE *index of q is a directed graph $G_A = (V_A, E_A)$ that satisfies the following conditions.*

- *For each $m \in V_A$, $\exists$ an injective function, $f(m)$: $m \rightarrow f_g$ s.t. $f_g \subseteq q$ and $\exists n \in V_I, n.g = f_g$. We refer to n as the matching vertex and denote its identifier as $id(n_m) = n.id$.*
- *By slightly abusing the notations of trees, each $(m', m) \in E_A$ represents the parent-child relationship between two vertices m' and m where m is the child of m' iff $f(m') \subset f(m)$ and $|f(m)| = |f(m')| + 1$.*
- *Each $m \in V_A$ is a 2-tuple $m = (id(n_m), \mathbb{E}_q)$ where $\mathbb{E}_q$ is a set of edge id sets in q that matches the edges in $f(m)$.*

Observe that unlike VACCINE, each vertex m in ADVISE does not store identifiers of data graphs in $\mathcal{D}$ that contain the fragment represented by m (*i.e.,* $f(m)$). Instead, it simply stores the identifier of the matching vertex of m in VACCINE. We can easily retrieve the identifiers of these data graphs on demand from this matching vertex. *Furthermore, during visual exploratory search, we create only one instance of* ADVISE. It is worth noting that such a query-dependent index is not constructed by any traditional subgraph querying techniques during query formulation. In this conventional paradigm, the *complete* query is first specified *before* it is processed. These approaches exploit knowledge of the *complete* query graph to compute and utilize auxiliary data structures. In contrast, the construction of ADVISE is intertwined with the formulation of a visual subgraph query that is revealed progressively.

B. *ADVISE Construction For Add Action*

We now describe the procedure to construct the ADVISE index on the fly for each add action. In the next subsection, we shall elaborate on how it is maintained for each modify action. We first introduce the ADVISE construction process using an example. Then, we describe the algorithm formally. Observe that the construction process entails creation of vertices and edges (Definition 2) and their associated annotations for a visual query fragment.

Recall that a user may either create an edge or drag-and-drop a template pattern during an add action. We consider a template pattern as an *edge stream* (a collection of edges). Consequently, a single new edge (referred to as *seed*) is processed iteratively to construct an ADVISE index.

Example 5: Consider the query formulation process in Figure 4(a). The ADVISE index construction process for this query is depicted in Figure 4(b). Suppose we use the VACCINE index in Figure 3 for the construction. The user first adds the edge `C-C` (id 1), which is the seed. Then the ADVISE construction process first searches the VACCINE index based on the CAM code of the edge. Since it matches the f_2 fragment in vertex 2 (*i.e.,* n_2 is the matching vertex) of the VACCINE index, a new vertex is added in the ADVISE index corresponding to n_2 as shown in the bottom row. Hence, the id of this vertex is set to 2 and $\mathbb{E}_q(m_2) = \{\{1\}\}$.

Next, the user adds the edge `C-Cl` (id 2), which is the new seed. Now the algorithm retrieves the identifier (*i.e.,* 1) of the vertex in VACCINE that matches the new edge (*i.e.,* f_1). Consequently, another vertex is created in ADVISE whose id is 1 and $\mathbb{E}_q(m_1) = \{\{2\}\}$. Next, it utilizes the primitive transformer information $\mathbb{T}$ encoded in the outbound edges of the vertex n_1 in VACCINE to find other vertices representing fragments that can be constructed using these two edges in the query (*e.g.,* `C-C-Cl`). Hence, the frequent fragment `C-C-Cl` (id 5) is retrieved (using $\Psi_{f_2}(0, \texttt{Cl})$) and is added as a vertex in ADVISE with $id = 5$ and $\mathbb{E}_q(m_5) = \{\{1, 2\}\}$. Since f_2 and f_1 are parents of f_5 in VACCINE, the vertices 2 and 1 should link to 5 in ADVISE. Hence, edges $(2, 5)$ and $(1, 5)$ are constructed in ADVISE to indicate that they are subgraphs of the fragment in vertex 5.

The user once again adds an edge `C-C` (id 3) as a seed to the query graph. Similar to the above step, it is first matched to the vertex with $id = 2$ in VACCINE. Observe that the vertex to represent this edge has already been created in ADVISE earlier. Hence the set of edge sets of this vertex is updated to $\{\{1\}, \{3\}\}$ (*i.e.,* $\mathbb{E}_q(m_2) = \{\{1\}, \{3\}\}$). Next, the remaining labels in the query graph (*i.e.,* `C`, `Cl`) are added to this seed and searched in the VACCINE index by utilizing the primitive transformers associated with the edges of n_2 in this index. For instance, the label `C` is added to `C-C` resulting in the fragment `C-C-C`, which is processed by leveraging the

Algorithm 4: The ADVISE construction algorithm.

```
Input: The VACCINE index $G_I$, an instance of the ADVISE index $G_A$, query fragment $q = (V_q, E_q)$, a single new edge $seed$.
Output: an updated $G_A$.
1  $S \leftarrow \emptyset$, $n \leftarrow$ GETVERTEX($G_I, seed.camcode$)
2  if $n$ is Not Null then
3      if $\exists n_s \in G_A$ for representing $n$ then
4          UPDATEVERTEX($G_A, seed, n$)
5      else
6          ADDVERTEX($G_A, (n.id, \{seed\})$)
7      $S$.ENQUEUE($(seed, n)$)
8      while $S \neq \emptyset$ do
9          $(s, n) \leftarrow S$.DEQUEUE()
10         for $f_q \in q, s \subset f_q$ and $|f_q| - |s| = 1$ do
11             $m \leftarrow$ MATCHINGINVACCINE($f_q, G_I, n, \tau$)
12             if $\exists n_m \in G_A$ for representing $m$ then
13                 UPDATEVERTEX($G_A, f_q, n_m$).
14             else
15                 ADDVERTEX($G_A, (m.id, E_{f_q})$)
16             ADDEDGE($G_A, n_m.parents, n_m$)
17             $S$.ENQUEUE($(f_q, n_m)$)
```

transformer $\Psi_{f_2}(0, \texttt{C})$ in VACCINE. Finally, the whole query fragment (*i.e.,* C−C−C−Cl) is processed. Since it is a DIF, it is stored in VACCINE (vertex n_{10}). Hence, a vertex with id 10 is added to ADVISE. Following this, edges $(5, 10)$ and $(3, 10)$ are created in ADVISE to indicate that the fragments represented by vertices 3 and 5 are subgraphs of the fragment in 10.

Next, an edge (id 4) is added to connect nodes 1 and 3. Consequently, we update ADVISE by following the above strategy. Note that fragments represented by vertices 8 and 10 are DIFs. Hence, we do not add any children to these vertices as it will create a NIF, which is not indexed in VACCINE.

Lastly, the benzene ring pattern is added. Each edge in this pattern is added sequentially into the query automatically and is processed by following the aforementioned strategy. Observe that the query is now a NIF. ∎

Remark. In the above example, the first edge C−C (id 1) is a frequent edge. How do we construct ADVISE when a user first constructs an infrequent edge? Recall that an infrequent edge is a DIF. Hence, it is already indexed in VACCINE. Consequently, the construction process of ADVISE is similar to the above example. ‘

Algorithm: Algorithm 4 outlines the formal procedure for constructing an ADVISE index. We first list the following functions to facilitate the exposition.

- GETVERTEX($G_I, camcode$): Find a vertex in the VACCINE index G_I that matches the CAM code $camcode$.
- ADDVERTEX($G_A, (n.id, \{e\})$): Add a new vertex in the ADVISE index G_A that represents n in VACCINE whose id is $n.id$ and $(n.id, \{e\})$ represents the 2-tuple of the new vertex.
- UPDATEVERTEX(G_A, e, n): Update the vertex n in G_A by adding e in $n.\mathbb{E}_q$.
- MATCHINGINVACCINE(f_q, G_I, n, τ): Match the fragment f_q according to the outbound edges and τ of vertex n in G_I and return the corresponding vertex, representing the fragment f_q, in G_I.
- ADDEDGE($G_A, n.parents, n$): Add edges from parent vertices of n to the vertex n in G_A.

Algorithm 5: REMOVEEDGE()

```
Input: ADVISE index $G_A$, an edge $e$ of the query $q$
Output: Updated ADVISE index.
1  for $m \in G_A$ do
2      for $edgeSet \in \mathbb{E}_m$ do
3          if $e \in edgeSet$ then
4              $\mathbb{E}_m \leftarrow \mathbb{E}_m - edgeSet$
5      if $\mathbb{E}_m = \emptyset$ then
6          $G_A$.DELETE($m$)
7  return $G_A$
```

Given a query graph q and a seed $seed$, Algorithm 4 first finds the matching vertex n in the VACCINE index G_I having the same CAM code as that of the $seed$ (Line 1). If n can be found in G_I, it indexes it in G_A as n_s (Lines 2-6). If n_s has already been created in G_A earlier, it is updated by invoking UPDATEVERTEX($G_A, seed, n$) (for adding $seed$ in $n.\mathbb{E}_q$). Otherwise, it creates a new vertex in G_A to represent $seed$. Next, it iteratively finds all frequent fragments and DIFs in q that contain $seed$ and index them in G_A. Specifically, it first inserts $seed$ and its corresponding matching vertex n to a queue S (Line 7). Note that each element $(s, n) \in S$ is a 2-tuple where the former is a fragment of q and the latter is its matching vertex n in V_I. For each $(s, n) \in S$ and fragment f_q satisfying the condition in Line 10, the algorithm indexes these frequent fragments and DIFs of q in G_A (Lines 11-17). Specifically, for each f_q it searches VACCINE by utilizing the primitive transformers associated with the edges of n in this index. Then it indexes the matching vertex n_m in G_A by either invoking ADDVERTEX for a new vertex (Line 15) or invoking UPDATEVERTEX for an existing vertex in G_A (Line 13). Besides, the relationships between n_m and its parent nodes $n_m.parents$ are added as edges (Line 16). After storing n_m in G_A, f_q and n_m are enqueued to S (Line 17) for finding larger frequent fragments and DIFs of q and indexing them in G_A in subsequent iteration. The algorithm terminates when S is empty. Observe that it does not index NIFs.

Theorem 2: Given a VACCINE *index $G_I = (V_I, E_I)$ and a query $q = (V_q, E_q)$, the time and space complexities of building* ADVISE *index for q is $O(|E_q| * (|V_I|C_{CAM} + min(|V_I|, x_{fq})(|V_q| + |E_q|)))$ and $O(m * min(|V_I|, 2^{|E_q|} - 1))$, respectively, where x_{fq} is the maximum number of frequent fragments and* DIFs *of q in G_I for all query edges, m is the space complexity of a vertex in the* ADVISE *index, and C_{CAM} is the time complexity of comparing the* CAM *codes of a pair of graphs.*

Remark. Observe that in principle the content of ADVISE index can also be constructed directly from $\mathcal{D}$ instead of utilizing the VACCINE index. However, this will be prohibitively expensive, rendering the construction of ADVISE *within* the GUI latency impractical.

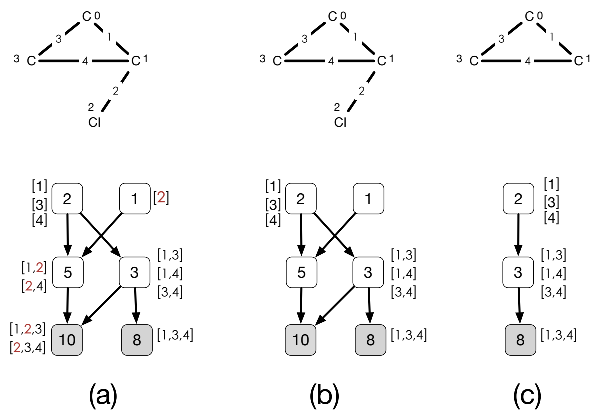


Fig. 5. **Example of query modification.**

C. Handling Modify Action

In exploratory search, a user may modify the current query fragment at any point of time (*e.g.,* after browsing result matches returned by a run action) by deleting[8] a set of edges (*i.e.,* modify action). Upon deletion of an edge, we firstly iterate through $\mathbb{E}_m$ of each vertex in the ADVISE index G_A. If the deleted $edge$ belongs to an edge set in $\mathbb{E}_m$ of a vertex m, then it modifies $\mathbb{E}_m$ of m by removing the edge set that contains the deleted $edge$. Subsequently, after removal of the edge sets from $\mathbb{E}_m$, we remove all vertices from G_A whose $\mathbb{E}_m$ are empty after the update. Observe that the complexity of removing an edge from a query is $O(N_A)$ where N_A is the number of vertices in G_A. These steps are repeated for every deleted edge. The formal algorithm is given in Algorithm 5.

Example 6: Consider Figure 5. The top row shows the modification to the query fragment, and the bottom row illustrates the ADVISE index update. Suppose the user removes the edge C-Cl (id 2) from the query fragment in Figure 5(a). First, it searches ADVISE to identify vertices whose $\mathbb{E}_m$ contains the edge C-Cl (vertices 1, 5, and 10). Next, all edge sets containing this edge are deleted (shown in Figure 5(b)). Finally, all vertices whose $\mathbb{E}_m = \emptyset$ (*i.e.,* 1, 5, and 10) are removed from the index. The updated ADVISE index is shown in Figure 5(c). ∎

D. Comparison with PICASSO

In PICASSO [13], for each new edge e_ℓ, a SPIG is created. A SPIG [18] is a directed graph $S_\ell = (V_\ell, E_\ell)$ where each vertex $v \in V_\ell$ represents a subgraph g of the query fragment containing e_ℓ. There is a directed edge from v' to v if $g' \subset g$ and $|g| = |g'| + 1$. Each vertex v is associated with the CAM code of the corresponding g, a list of identifier of edges of g, and a *Fragment List* of g. The *Fragment List* contains the following four attributes. If g is a frequent fragment or a DIF, then the corresponding identifier of the matching vertex in the offline index for storing frequent fragments or DIFs is stored in *frequent id* or DIF *id* attribute, respectively. However, if g is a NIF, then the *frequent subgraph id set* stores the frequent ids of all largest proper subgraphs of g that are frequent. Lastly, the DIF *subgraph id set* of g contains the DIF ids of all subgraphs of g that are DIFs.

The ADVISE index differs from SPIGs in the following ways. First, each vertex in ADVISE is lightweight as it only stores an identifier and an edge list. In contrast, each vertex in a SPIG stores a larger collection of attributes. Particularly, the computation of frequent subgraph id set and DIF subgraph id set for each vertex in a SPIG can be expensive. Second, PICASSO generates a set of SPIGs, one for each edge in the query graph. In contrast, we only create a *single* ADVISE index regardless of the number of edges added to the query. Consequently, the construction cost of ADVISE is not only cheaper than SPIGs, but the former is also more space efficient. Specifically, ADVISE takes only $O(min(|V_I|, x_{fq})(|V_q| + |E_q|))$ (Theorem 2) to index all frequent fragments and DIFs that contain e_ℓ. In contrast, PICASSO takes $O(min(|V_I|, x_{fq})(|V_I| C_{CAM}))$ because its indexes do not preserve the formulation relationships between different fragment fragments and DIFs. Consequently, it needs to find corresponding vertices for frequent and DIF subgraph id sets by CAM code comparison.

VII. Processing of Run Action

We have now all the machinery in place to present the algorithms for blending visual query formulation with query processing. An EAS of exploratory subgraph search may contain multiple run actions. In this section, we show how the VACCINE and ADVISE indexes are utilized to efficiently support processing of run(q) during exploratory search. Since q for each run action can be either subgraph containment or subgraph similarity search, we discuss how these two categories of search can be realized. Specifically, we describe how the procedures in Lines 7, 11, 14, and 18 of Algorithm 1 are implemented when the action-aware indexes and SPIGs of PICASSO are replaced with VACCINE and ADVISE, respectively. Note that our goal here is not to design yet another subgraph search technique but rather on how the PICASSO framework for the run action can be adopted to suit our proposed indexing schemes.

Subgraph Containment Search. Recall from Section II-D, in PICASSO if a query graph is a frequent fragment or a DIF, then its matches can be directly retrieved from the action-aware indexes without any verification. On the other hand, if the query graph is a NIF, then additional verification is performed on the candidate data graphs by utilizing the VF2 algorithm [5]. Hence, we follow the same strategy but replace the inefficient indexing schemes of PICASSO with VACCINE and ADVISE. Let us illustrate it with an example.

Reconsider Figure 4. Suppose during exploratory search a user clicks on Run after constructing the third and fourth edges. In the case of the third edge, observe that there is a vertex in ADVISE representing this query fragment (vertex 10). Thus, we retrieve the matching vertex of 10 in VACCINE and return all data graph identifiers associated with it without any further verification. However, when the Run button is clicked again after adding the fourth edge, there does not exist a vertex in ADVISE that represent the query fragment. Hence, the query is a NIF and we retrieve all leaves of ADVISE (vertices 8 and

[8]Update of an edge can be considered as deletion followed by addition of a new edge (*i.e.,* modify and add actions).

10). Observe that these leaves must be subgraphs of the query. Next, we intersect the candidate data graph identifiers retrieved from the corresponding matching vertices in VACCINE for these leaves. Finally, we verify these candidates and generate the exact matches of the query.

Subgraph Similarity Search. Given δ, the key intuition followed by PICASSO for subgraph similarity search is to iteratively modify the formulated query graph by removing edges according to δ and then invoke the subgraph containment search procedure for all modified queries whose distance from the query fragment is less than δ [13], [18]. Consequently, we exploit this framework, but instead of SPIGs we utilize our more efficient ADVISE index. Specifically, we iterate through a queue containing pairs of modified query graph q_i and corresponding ADVISE index G_{A_i} for different δ and compute all valid subgraph similarity queries (w.r.t δ) and their corresponding matches. After traversing through all the edges of q_i for all legal values of δ, a list of similar candidate data graphs is returned. The similarity verification of these candidates is performed by utilizing the VF2-based technique used in [13].

It is worth noting that despite exhaustive enumeration of modified query graphs satisfying δ constraint, this approach is efficient in practice in exploratory search environment (see Section VIII) for the following reasons. First, δ is typically small in practice as its large value will make the result matches topologically very different from a user's search intent. Second, during exploration when a search query grows large, it indicates that the user has more precise knowledge of the topology of her search intent. Consequently, the chance of δ to be large also diminishes significantly. Last but not the least, the efficient representation of the ADVISE index enables us to efficiently compute all candidate matches.

Remark. The formulation sequences of a query q do not affect the candidate set because it only depends on q, and the VACCINE and ADVISE indexes. Hence, different query formulation sequences of a query graph (*i.e.,* different ordering of edges) in our framework do not have significant impact on the query processing cost.

VIII. Performance Study

In this section, we investigate the performance of FERRARI and report the key results. All experiments are performed on a machine with 64 bits Intel dual Core i7 3.7 Ghz CPU and 16 GB RAM, running on Windows 7.

A. Experimental Setup

Datasets. We use the following three real-world datasets (Table I) for our experimental study.

- **AIDS:** This antiviral database contains topological structures of around 40K chemical compounds. We randomly select graphs to create four subsets with sizes 10K (4.4MB), 20K (9.4MB), 30K (14.4MB), and 40K (19.4MB).
- ***eMolecules***: This dataset contains 1300K chemical structure graphs. We randomly select graphs to create four subsets with sizes 320K (91.4MB), 640K (183.5MB), 960K (278.8MB), and 1300K (367.7MB).

TABLE I
Datasets (K = '000).

Database	*# of graphs*	*Avg. Nodes*	*Avg. Edges*	*Max. # of Nodes*	*Max.# of Edges*
AIDS	$40K$	25	27	222	251
eMolecules	$1300K$	17	17	459	472
PubChem	$800K$	41	42	801	838

- ***PubChem***: This dataset contains chemical compound graphs [21]. We randomly select graphs to create four subsets with sizes 100K (80MB), 200K (140MB), 400K (283MB), and 800K (559MB).

Algorithms. We compare FERRARI with PICASSO [13], which is the state-of-the-art VESS framework. Both are developed in Java. We set the minimum support threshold $\alpha = 0.1$ for FERRARI and PICASSO unless specified otherwise. For PICASSO, we set the fragment size threshold $\beta = 8$ as recommended in [18]. We use *gSpan* [38] for computing the frequent fragments.

Query set: Recall that during exploratory search, the size of a query graph may evolve from small to large (Example 1). That is, a user typically formulates and executes query graphs of different sizes. Hence, we select a set of queries of different sizes. Since these queries need to be formulated visually by users, it is not possible to evaluate a large number of them. Our experience suggests that having to formulate many queries with different query formulation sequences (QFSs) strongly deters users from participating in a user study. Hence, we select 11, 7, and 6 queries for the AIDS, *eMolecules* and *PubChem* datasets, respectively. These queries are selected based on several features such as size (6-43), topology (path, tree, graph, and cycle), and label variety. Figures 6, 7 and 8 depict these queries. Specifically, in AIDS (resp. *eMolecules*) Q_1 to Q_9 (resp. Q_1 to Q_5) do not have any exact matches but Q_{10} and Q_{11} (resp. Q_6 and Q_7) do. Similarly, in *PubChem* Q_4 to Q_6 do not have exact matches but Q_1 to Q_3 do. Labels on edges of a query represent the default sequence of steps for visual query formulation in FERRARI and PICASSO by taking an edge-at-a-time approach. Unless mentioned otherwise, we shall be using the default sequence for formulating a particular query. The template patterns used in our experimental study are encompassed by dotted red lines in these queries. We use the GUI of PICASSO for query formulation in FERRARI (see Appendix A).

Participants profile. Twelve unpaid volunteers (ages from 21 to 27) participated in the experiments in accordance with HCI research that recommends at least 10 users [23], [24]. None of them are authors of this paper. They were first trained to use the GUI. To describe the queries to them, we provided printed visual subgraph queries. A participant then draws the given query using a mouse in our GUI. For every query, the participants were given some time to determine the steps that are needed to formulate it so that the effect of "thinking" time is minimized. Note that the faster one formulates a query, the lesser time available for ADVISE and SPIGs construction. Each query was formulated 5-10 times by different participants (in edge-at-a-time or pattern-at-a-time modes) following different QFSs. Hence, more than **160 queries** are executed.

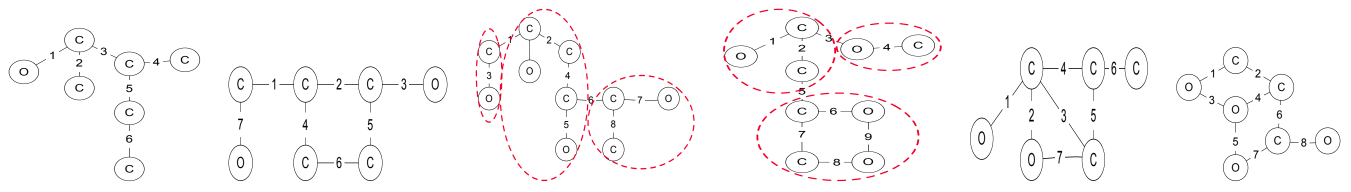
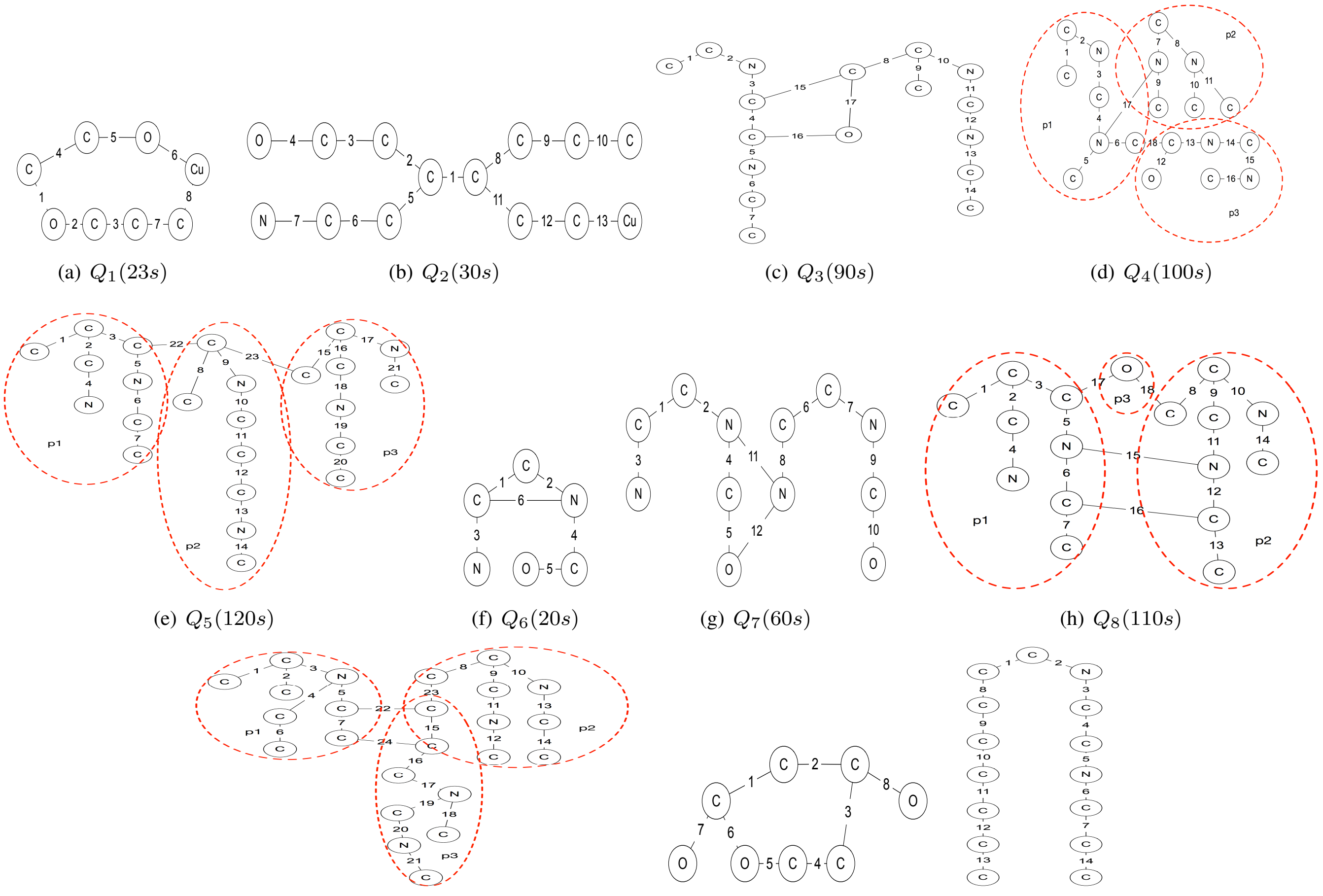

(a) $Q_1(20s)$ (b) $Q_2(18s)$ (c) $Q_3(30s)$ (d) $Q_4(40s)$ (e) $Q_5(15s)$ (f) $Q_6(28s)$

Fig. 6. **Queries for *PubChem* dataset. The query formulation time is shown in parenthesis.**


(a) $Q_1(23s)$ (b) $Q_2(30s)$ (c) $Q_3(90s)$ (d) $Q_4(100s)$

(e) $Q_5(120s)$ (f) $Q_6(20s)$ (g) $Q_7(60s)$ (h) $Q_8(110s)$

(i) $Q_9(150s)$ (j) $Q_{10}(37s)$ (k) $Q_{11}(50s)$

Fig. 7. **Queries for AIDS dataset. The query formulation time is shown in parenthesis.**

Performance measures: Recall that an EAS comprises of a sequence of add, modify, and run actions. Hence, we measure their performances. The add and modify actions result in creation and maintenance of the ADVISE and SPIGS in FERRARI and PICASSO, respectively. Consequently, we measure the construction and update cost of these indexes. The run action executes a subgraph query of a specific size and type (subgraph containment or subgraph similarity). It is not impacted by previous formulation steps or ADVISE construction. Hence, we measure the *system response time* (SRT), which refers to the duration between the time a user presses the Run icon to the time when she gets the query results. Observe that any VESS framework that efficiently supports these actions naturally can support exploratory subgraph search efficiently as such search process comprises of a sequence of these three actions.

B. *Experimental Results*

Exp 1: Performance of the run action. In this set of experiments, each query is formulated by a sequence of add actions (*i.e.,* modify action is excluded) and executed using the run action. We first investigate the performance of subgraph containment queries of different sizes. Specifically, we use $Q_{10} - Q_{11}$ on AIDS dataset, $Q_6 - Q_7$ on *eMolecules*, and $Q_1 - Q_2$ on *PubChem*. Figures 9(a)-(c) plot the SRTs. Note that PICASSO suffers from out-of-memory issue for Q_{11}, for *eMolecules*, and for Q_2 of *PubChem*. Observe that FERRARI outperforms PICASSO by up to 4 orders of magnitude (Q_{10}). When larger query graphs are formulated, the size of SPIGS generated by PICASSO increases significantly. Consequently, the construction time and memory usage of SPIGs increase significantly. This issue is largely alleviated in FERRARI due to the usage of efficient ADVISE index.

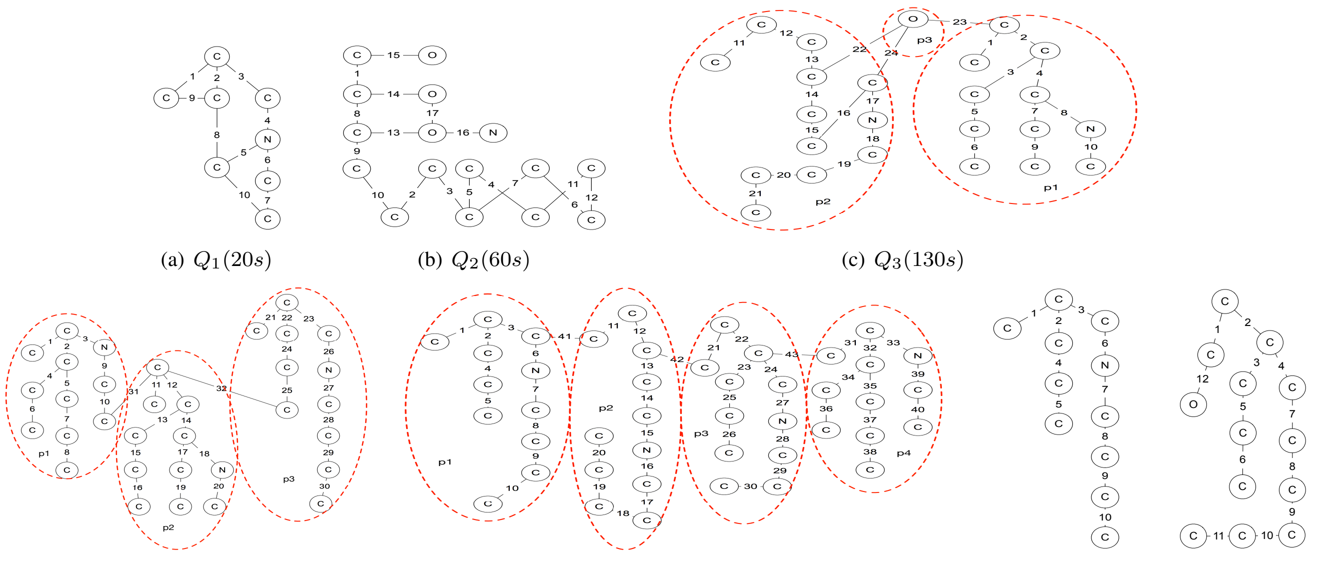

(a) $Q_1(20s)$ (b) $Q_2(60s)$ (c) $Q_3(130s)$ (d) $Q_4(220s)$ (e) $Q_5(500s)$ (f) $Q_6(60s)$ (g) $Q_7(70s)$

Fig. 8. **Queries for *eMolecules* dataset. The query formulation time is shown in parenthesis.**

Next, we study the performance of subgraph similarity queries for different δ. We report the performances of top-3 largest queries on *eMolecules* and *PubChem* datasets, *i.e.,* ($Q_3 - Q_5$) on *eMolecules* (320K) and $Q_4 - Q_6$ on *PubChem* (400K). Note that PICASSO failed to handle these queries for aforementioned reasons. Figures 9(d) and 9(e) plot the SRTs. We can observe that the SRT increases as δ increases. This is expected as the verification workload increases as δ increases. Recall that, in practice δ is set to a small value.

We also test the scalability of FERRARI by utilizing a subset of queries. We set $\delta = 2$. Figure 9(f) reports the results of the study. Unlike PICASSO, the run action of FERRARI can efficiently process larger exploratory query graphs on dataset containing more than one million data graphs.

Next, we study the impact of template pattern-based query formulation. We use $Q_4 - Q_5$ and $Q_8 - Q_9$ of AIDS, $Q_3 - Q_5$ of *eMolecules*, and $Q_3 - Q_4$ of *PubChem*. The number of patterns used to formulate each query is at most 4. After dragging these patterns from the *Pattern Panel* to the *Query Panel*, we connect them to the query fragment by adding relevant edges. In addition to the default QFS (denoted by s_1), we use three different QFS for these queries (denoted by s_2, s_3, and s_4). Table II shows the average SRT. PICASSO failed to process them as it failed to handle the template patterns within the GUI latency. In contrast, the performance of the run action in FERRARI is efficient and not adversely impacted by this mode of query formulation. Note that Q_4 (*PubChem*) has larger SRT as it is a similarity search query on a data source that contains data graphs much larger than those in *eMolecules* or AIDS. Furthermore, the choice of QFS does not significantly impact on the query performance. Consequently, FERRARI does not need to employ any selectivity-based strategy for ordering the edges in a template pattern.

Finally, we evaluate the impact of minimum support threshold α on the run action. We set different values of α from 0.1 to 0.3 in AIDS (40K), from 0.05 to 0.2 in *eMolecules* (960K), and from 0.05 to 0.2 in *PubChem* (400K). Note that α affects the number of frequent fragments and DIFs in the VACCINE index. Figure 10 reports the SRTs for different values of α for four representative queries. Importantly, the SRT fluctuates in a small range with the variation of α. Note that the SRT depends not only on α but also on the query structure. If a query contains several NIFs, the possibility of finding matching fragments in the VACCINE index may be reduced. In this case, we need more time for verification during query execution. On the other hand, if a query contains several frequent fragments, the pruning power of VACCINE is enhanced as α increases. Consequently, the SRT will be reduced significantly.

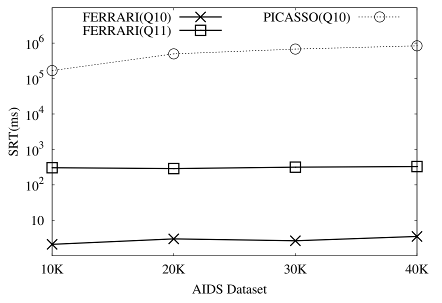
TABLE II
Impact of template patterns on the run action (avg. SRT (msec)).

Query	*Dataset*	s_1	s_2	s_3	s_4
Q4	AIDS (40K)	14	10	11	7
Q5	AIDS (40K)	48	28	35	30
Q8	AIDS (40K)	13	21	18	25
Q9	AIDS (40K)	40	23	30	34
Q3	eMolecules (960K)	50	70	65	55
Q4	eMolecules (960K)	80	90	100	90
Q5	eMolecules (960K)	300	180	200	190
Q3	PubChem (400K)	12	15	14	15
Q4	PubChem (400K)	2100	2000	2150	2210

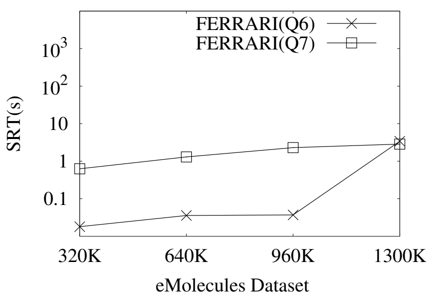
TABLE III
Performance of the add action for different QFS (in msec).

Method	*Query*	*Dataset*	s_1	s_2	s_3	s_4
FERRARI	Q1	AIDS (40K)	12	10	14	6
PICASSO			1800	1900	1800	1600
FERRARI	Q6	AIDS (40K)	19	13	21	16
PICASSO			160	400	300	200
FERRARI	Q3	eMolecules (960K)	810	550	740	550
FERRARI	Q5	eMolecules (960K)	380	350	360	230
FERRARI	Q3	PubChem (400K)	17	15	13	16
FERRARI	Q5	PubChem (400K)	380	390	410	350

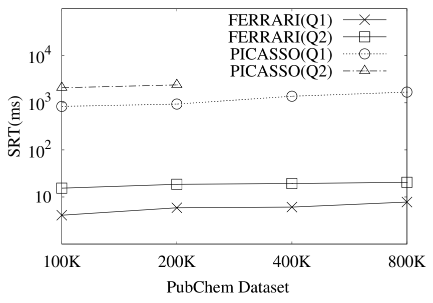
Exp 2: Performance of the add action. An add action in an EAS triggers the construction of ADVISE index. Hence, we



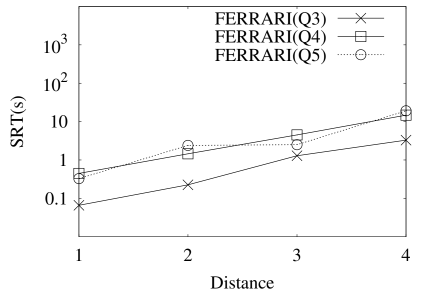
(a) Run action (AIDS)



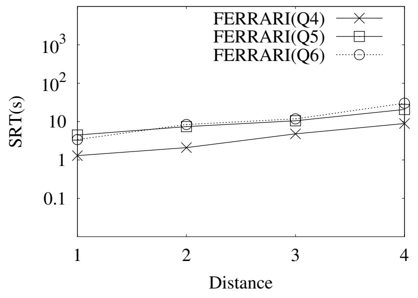
(b) Run action (*eMolecules*)



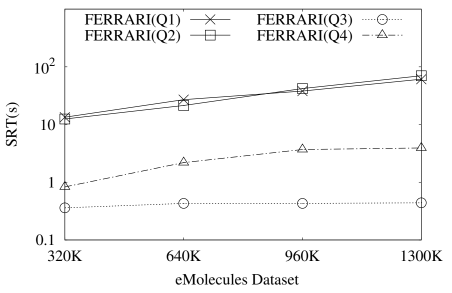
(c) Run action (*PubChem*)



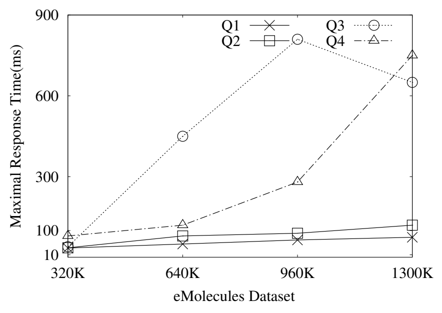
(d) Run action (*eMolecules*)



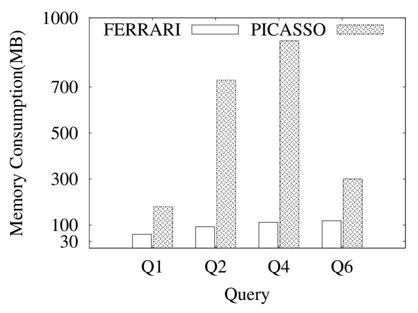
(e) Run action (*PubChem*)



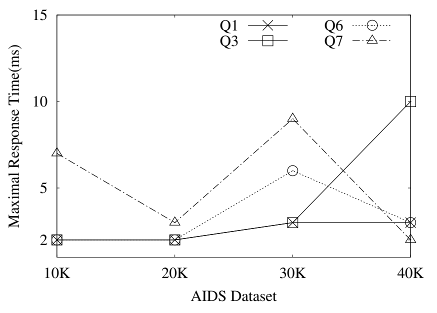
(f) Scalability (in sec)



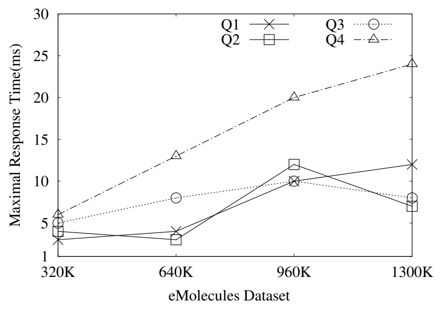
(g) MCC (*eMolecules*)



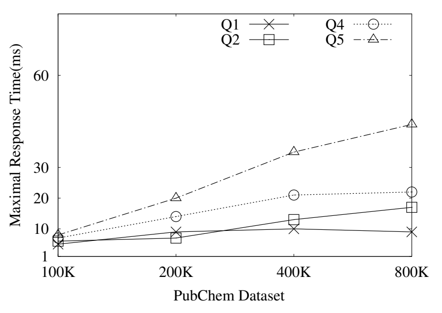
(h) Memory cost (AIDS)



(i) Modify action (AIDS)



(j) Modify action (*eMolecules*)

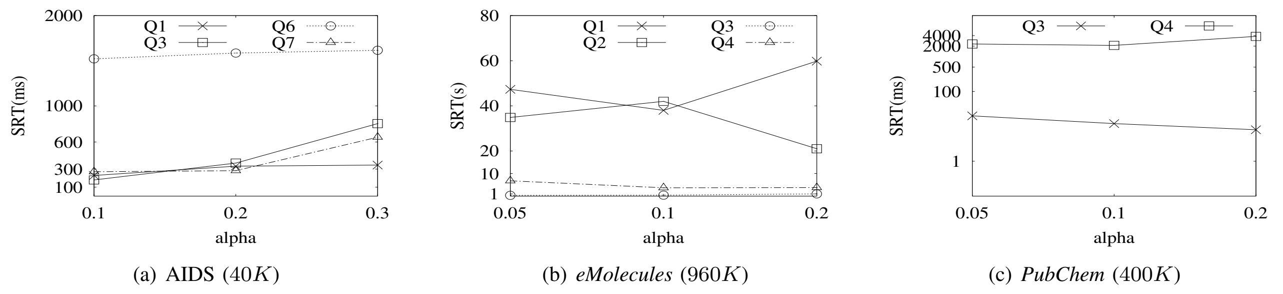


(k) Modify action (*PubChem*)

Fig. 9. **Experimental results of run, add, and modify actions.**

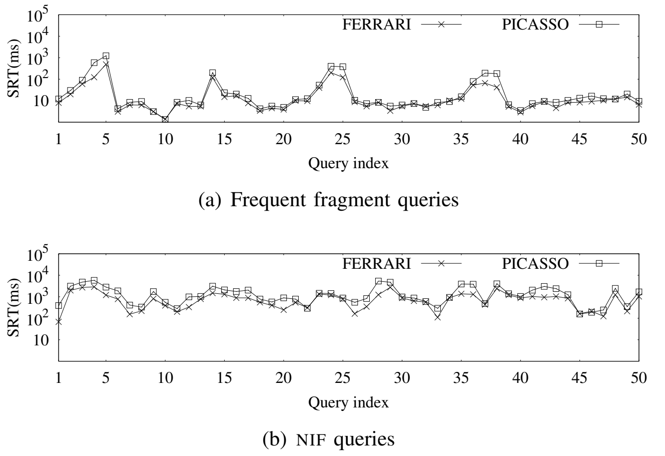
report the construction cost of ADVISE in order to investigate the performance of the add action. Specifically, in this experiment each query graph is formulated using a sequence of add actions (*i.e.,* QFS) without any modify action. We record the *maximum construction cost* (MCC), which is the maximum time taken to construct the ADVISE index among all the add actions for a given QFS. Note that this time reflects the worst-case scenario of ADVISE index construction cost for a given QFS. Also, interleaving multiple run actions in the QFS do not impact MCC as it is only influenced by the query construction steps and not query execution. We set $\alpha = 0.1$, $\delta = 2$ and follow the default sequence of query formulation. Figure 9(g) reports the MCC in FERRARI for representative queries on *eMolecules*. Observe that ADVISE construction can finish within a second (typically less than the available GUI latency).

Next we investigate the impact of different QFS on MCC. Specifically, we compare the MCC consumed by FERRARI and PICASSO for constructing ADVISE and SPIGs, respectively. Table III reports the average MCC (all participants) for four different QFS. We do not report MCC of PICASSO if it fails to handle a query/dataset. Observe that ADVISE index construction in FERRARI is very efficient and is not significantly impacted by different QFS. Particularly, it can be more than



(a) AIDS ($40K$) (b) *eMolecules* ($960K$) (c) *PubChem* ($400K$)

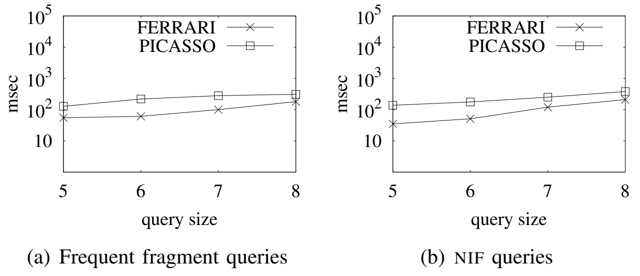
Fig. 10. **Effect of α on the run action.**



(a) Frequent fragment queries

(b) NIF queries

Fig. 11. **Performance of run action (AIDS) using a visual subgraph query simulator [2].**



(a) Frequent fragment queries (b) NIF queries

Fig. 12. **Performance of add action (AIDS) using a visual subgraph query simulator [2].**

200X faster than the SPIG set construction in PICASSO. Note that for Q_1, the MCC for PICASSO is high because the SPIG set construction in the last step is 10X slower than the previous steps.

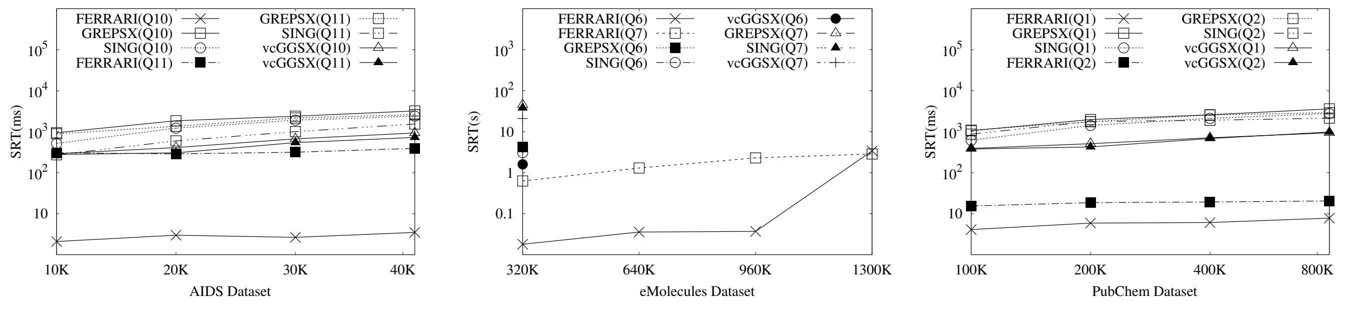
Lastly, we record the difference of memory consumption of ADVISE and SPIGs before and after the query formulation. Figure 9(h) shows the memory cost for a subset of queries in the AIDS dataset. Q_2 and Q_4 lead to out-of-memory problem in PICASSO, so the memory consumption reported here is the last step before it crashed. Clearly, the memory consumption of ADVISE is significantly smaller than SPIGs.

Exp 3: Performance of the modify action. Recall that the modify action leads to update of the ADVISE index. Hence, we measure the update cost of this index. Specifically, in this experiment we always delete the first constructed edge (*i.e.,* modify action) after the last add action of a QFS to simulate the worst-case scenario. That is, each QFS is a sequence of add actions followed by a modify action. Each query is formulated by five different participants and we report the *maximum* modification time (Figures 9(i)-(k)) for four representative queries. Observe that the modification time is cognitively negligible. That is, the cost of updating the ADVISE index is negligible. Furthermore, given that it takes less than 50 msec to update the ADVISE index, deletion of a set of edges at one go (*e.g.,* template pattern) on the GUI by a user can be efficiently realized by processing them sequentially.

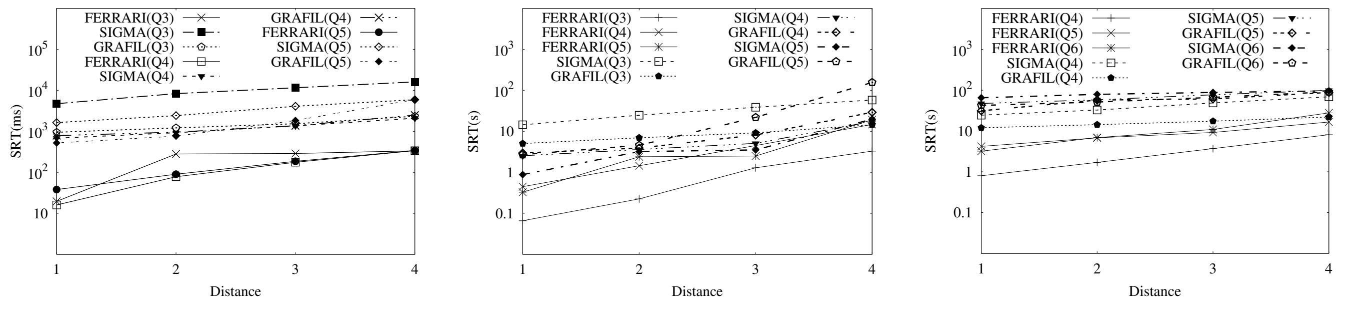
Exp 4: Automated performance comparison. We use a recently proposed *visual subgraph query simulator* called VISUAL [2] to automatically select a larger collection of small-sized subgraph queries and simulate their formulation. Since the current implementation of VISUAL simulates small-sized subgraph query formulation (size up to 9) using edge-at-a-time formulation mode, we cannot use this tool to evaluate larger size visual queries or template pattern-based query formulation. We integrated VISUAL on top of FERRARI and PICASSO and randomly generated 100 queries (50 frequent and 50 NIFs) of $5-8$ edges, with different topologies (path, tree, graph, and cycle), and simulated their formulation for all possible QFS. Since VISUAL does not support the modify action, we are confined to study add and run actions. We use the AIDS $40K$ dataset ($\alpha = 0.1$) since PICASSO suffers from out-of-memory issue for *eMolecules* and *PubChem* datasets.

Figure 11 shows the avg. SRT (performance of run action) for all possible QFS. Observe that even for small-size query graphs FERRARI consistently outperforms PICASSO. Figure 12 plots the average processing time of *each* edge in 5-edge to 8-edge query graph sets. Note that this time is used to construct ADVISE and SPIGs in FERRARI and PICASSO, respectively (performance of add action). Observe that the construction cost of ADVISE is consistently lower than SPIG set construction. That is, FERRARI *is superior to* PICASSO *even when the size of subgraph queries is small during exploration.*

Exp 5: Run action with traditional graph querying schemes. Here we compare the performance of run action in FERRARI with the following traditional techniques: (a) SIGMA [27] and (b) GRAFIL [40], which are traditional subgraph search techniques; (c) SING [7], (d) GRAPHGREPSX (GREPSX for brevity) [4], and (e) VCGGSX [35], which are exact subgraph matching techniques. Particularly, a recent study [20] demonstrated that GREPSX has superior performance compared to several existing exact subgraph

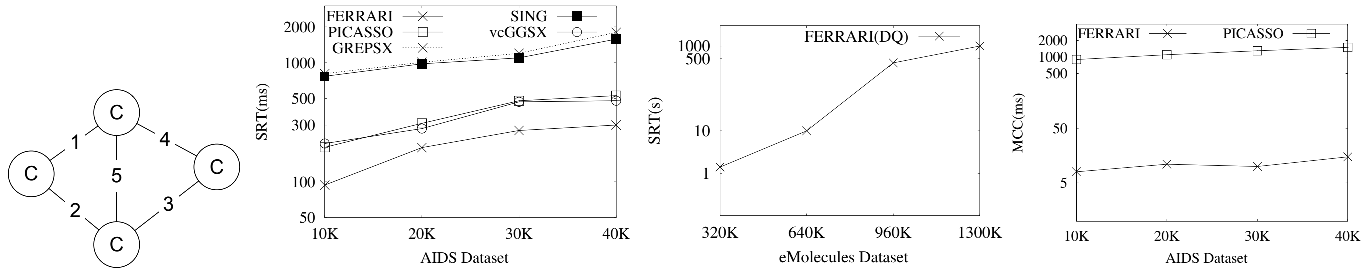


(a) Subgraph containment (AIDS) (b) Subgraph containment (*eMolecules*) (c) Subgraph containment (*PubChem*)



(d) Subgraph similarity (AIDS (40K)) (e) Subgraph similarity (*eMolecules* (320K)) (f) Subgraph similarity (*PubChem* (200K))

Fig. 13. **Comparison of the run action of FERRARI with traditional subgraph querying techniques.**



(a) A dense query (6s) (b) AIDS (c) *eMolecules* (d) MCC

Fig. 14. **Performance comparison with a dense query.**

matching techniques. We obtain the executable software of these algorithms from their authors. We shall use the default settings of these techniques as suggested in [4], [7], [27], [35], [40].

Figures 13(a)-(c) plot the SRTs of FERRARI, SING, GREPSX, and VCGGSX using the subgraph containment queries $Q10 - Q11$ on AIDS, $Q6 - Q7$ on *eMolecules*, and $Q1 - Q2$ on *PubChem*. For SING, GREPSX, and VCGGSX, SRT represents the query execution time. FERRARI is up to 3 orders of magnitude faster than these approaches. Note that SING, GRAFIL, GREPSX, VCGGSX, and SIGMA confronted the out-of-memory issue on *eMolecules* containing more than 320K data graphs and on *PubChem* with more than 400K data graphs. Figures 13(d)-(f) plot the results for subgraph similarity queries. We use the same query set and setup discussed earlier. We can observe that FERRARI can be up to 3 orders of magnitude faster than these approaches or have comparable performance for higher δ.

Lastly, we compare FERRARI with state-of-art techniques for processing a dense query graph. We extract one of the densest subgraphs (Figure 14(a)) from AIDS (40K) and *eMolecules* datasets. Note that due to chemical properties of compounds we cannot find any cliques with more than three nodes in all datasets. Figures 14(b) and (c) report the performances. Observe that FERRARI is consistently superior to other algorithms. Despite the superiority of VCGGSX in the verification and filtering step compared to SING and GREPSX, the interactive nature of FERRARI coupled with efficient indexes makes it more suitable for the VESS framework. We also record the difference of memory consumption before and after the query formulation. The maximum difference of FERRARI is nearly 500MB (AIDS), whereas it is nearly 1.1GB for PICASSO. Hence ADVISE is more space efficient than SPIGs. Figure 14(d) reports the MCC of FERRARI and PICASSO, which once again supports our claim that ADVISE is more efficient than SPIGs. *In summary,* FERRARI *is more suitable than traditional subgraph search techniques for* VESS.

Exp 6: Impact of query formulation sequence (QFS) on SRT. Next, we investigate the impact of QFS on the SRT. In addition to the default QFS (denoted by s_1), we use four different

TABLE IV
Average SRT (sec) for different QFS.

Method	*Query*	*Dataset*	s_1	s_2	s_3	s_4
FERRARI	Q1	AIDS (40K)	0.23	0.336	0.356	0.42
PICASSO			223.4	213.5	239.8	220.9
FERRARI	Q3	AIDS (40K)	0.17	0.15	0.16	0.17
FERRARI	Q6	AIDS (40K)	1.52	1.6	1.63	1.26
PICASSO			5.36	5.79	6.12	6.61
FERRARI	Q1	eMolecules (960K)	38	37	41	40
FERRARI	Q3	eMolecules (960K)	0.43	0.46	0.49	0.46
FERRARI	Q5	eMolecules (960K)	3.56	3.2	2.1	4.3
FERRARI	DQ	AIDS (40K)	0.3	0.35	0.31	0.41
PICASSO			0.53.4	0.76	0.61	0.58
FERRARI	DQ	eMolecules (320K)	1.4	1.6	1.3	1.5
FERRARI	Q3	PubChem (400K)	0.014	0.015	0.016	0.017
FERRARI	Q5	PubChem (400K)	7.3	6.8	7.9	8.1

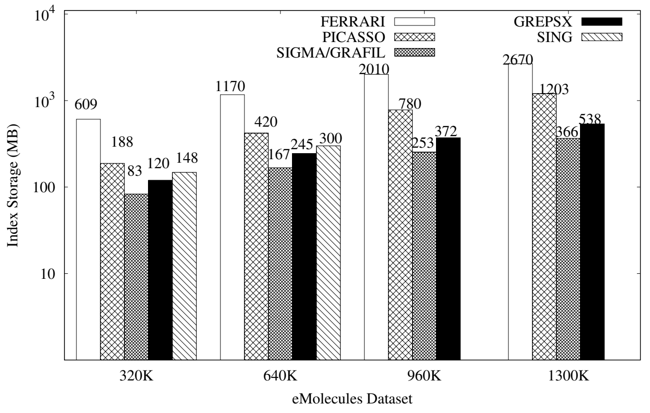


Fig. 15. **Size of VACCINE.**

QFS for representative queries (denoted by s_2, s_3, and s_4). Table IV shows the average SRT for different QFS. Note that DQ refers to the dense query in Figure 14(a). Observe that SRTs vary slightly for different QFS. In summary, QFS has cognitively negligible impact on the SRT.

Exp 7: Performance of VACCINE. Figure 15 shows the index sizes of different techniques for the *eMolecules*. Note that SIGMA and GRAFIL have the same index structure. All indexing strategies are independent of δ. SING suffers from out-of-memory problem for datasets containing more than $640K$ graphs. Observe that GREPSX consumes the smallest storage space as it only indexes paths up to a specific length. On the other hand, FERRARI's index size is larger than other systems because it needs to store all frequent fragments and DIFs along with their transformation information. Importantly, the gap between the index size of FERRARI and other techniques reduces significantly. Specifically, it utilizes 2.2X and 5X more space than PICASSO and GREPSX, respectively, for the largest dataset. Nevertheless, VACCINE index consumes less than 3GB for the largest dataset, which can easily fit in the main memory of modern machines.

Figure 16 plots the index building time and index size for different values of α. The building time constitutes the time to generate the frequent fragments using *gSpan* and indexing their primitive transformer-based relationships in the VACCINE index. Naturally, as α increases, the number of frequent fragments decreases leading to shorter index construction time as well as index size. Importantly, the index can easily fit in the main memory of modern machines for different α values and datasets.

In summary, although VACCINE consumes more space, we emphasize that its construction is a one-time cost. In view of the performance improvement brought to exploratory search queries, such a trade-off is appropriate.

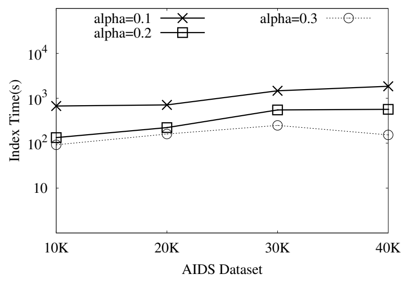
C. Summary

In summary, the key results are as follows.

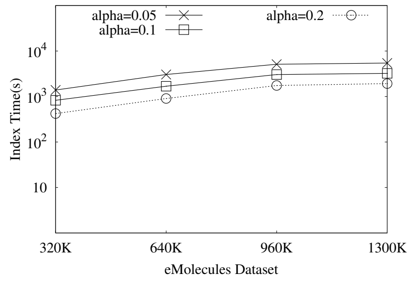
- The run action on our framework can be up to 4 orders of magnitude faster than PICASSO on large datasets. Even for small-sized queries, FERRARI consistently outperforms PICASSO due to superior indexing framework. Furthermore, it is not significantly impacted by a specific *mode* of query formulation (*i.e.,* template pattern-based or edge-based) or the choice of QFS.
- We investigated the construction cost of ADVISE index as it influences the performance of the add action. It is significantly faster than PICASSO and can finish within a second (*i.e.,* typically less than the available GUI latency). Particularly, it can be more than 200X faster than the SPIG set construction. It is also not significantly impacted by different QFS. Furthermore, the memory consumption of ADVISE is significantly smaller than SPIGs (up to 8X).
- The cost of updating the ADVISE index (*i.e.,* modify action) is cognitively negligible.
- FERRARI can be up to 3 orders of magnitude faster than VESS frameworks hinged on traditional subgraph search techniques.
- Although the size of VACCINE is up to 2.2X than the offline index in PICASSO, it consumes less than 3GB for the largest dataset, which can easily fit in the main memory of modern machines. Importantly, the gap between the size of VACCINE and existing indexes reduces with increasing dataset size.

IX. Related Work

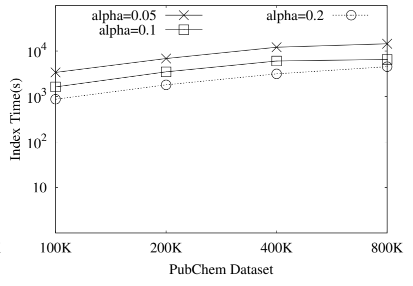
Yahya *et al.* [37] report exploratory querying framework on knowledge graphs. It focuses on automatic query relaxation, query suggestions, and explanation of answers. Recently, [28] focused on large graph exploration using keywords. In contrast, we focus on efficient evaluation visual exploratory subgraph queries on a large collection of data graphs. Hence, FERRARI is complementary to these efforts. As mentioned earlier, the techniques in PICASSO [13] are designed for an exploratory search environment where the subgraph queries remain small and hence cannot efficiently handle scenarios when the query grows large. Furthermore, PICASSO does not support efficient processing of multiple edges (*i.e.,* template patterns) at each step during visual query formulation. Hence, FERRARI is a much more generic and scalable VESS framework. Under the hood, as detailed in Sections V-D and VI-D, the VACCINE and ADVISE indexes are different from their counterparts [18], [19] in PICASSO.



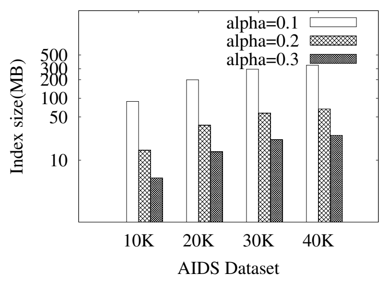
(a) AIDS



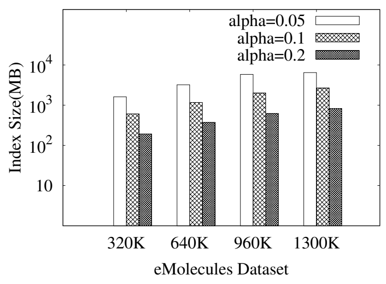
(b) *eMolecules*



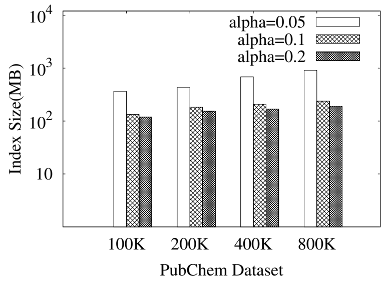
(c) *PubChem*



(d) AIDS



(e) *eMolecules*



(f) *PubChem*

Fig. 16. **Impact of α on VACCINE construction time and size.**

The paradigm of blending visual graph formulation and processing [14], [18], [19], [34] has been explored in the context of lookup queries (executed only once). QUBLE [14] and BOOMER [34] realize this on a large network for subgraph search and p-homomorphism queries, respectively. In contrast, our work focuses on exploratory search on a large collection of small- or medium-sized data graphs. The efforts in [18], [19] are designed for a large set of small- or medium-sized graphs. In fact, as mentioned earlier, PICASSO [13] utilizes the indexing frameworks of [18], [19] for exploratory search. Hence, they suffer from the aforementioned limitations.

The majority of incremental algorithms for graphs [6], [9], [10] focus on incrementally updating the query results in response to the changes in the underlying graph. In contrast, given a data graph collection we focus on updating partial results as the query fragment evolves during exploration. More importantly, these algorithms are not designed to exploit visual interaction behaviors of users and GUI latency.

Recently, there are efforts to develop techniques that aid visual graph query construction [16], [29], [41]. They focus on facilitating visual query formulation by providing suggestions [16], [41] and refinement strategies [29]. They do not focus on processing exploratory subgraph search queries.

X. Conclusions

In this paper, we present two novel index structures to efficiently support a visual exploratory subgraph search framework called FERRARI. Specifically, in contrast to the previous effort in [13], FERRARI is efficient and scalable with respect to the size of the query graph as well as the number of data graphs. Experimental studies on real data graphs validated the merit and superiority of our proposed technique compared to state-of-the-art exploratory subgraph search techniques. As for future work, we intend to explore how the VESS paradigm can be realized on large networks.

Acknowledgments. The first three authors are supported by AcRF MOE2015-T2-1-040 and AcRF Tier-1 Grant RG24/12. Shuigeng Zhou is supported by National NSF of China (Grant No. U1636205).

References

[1] J. Ahn, P. Brusilovsky. Adaptive Visualization for Exploratory Information Retrieval. *Info. Proc. & Man.* 49, 5, 2013.
[2] S.S. Bhowmick, H.-E. Chua, B. Choi, C. Dyreson. VISUAL: Simulation of Visual Subgraph Query Formulation To Enable Automated Performance Benchmarking. *In TKDE*, 29(8), 2017.
[3] A. Bonifati, W. Martens, T. Timm. An Analytical Study of Large SPARQL Query Logs. *In PVLDB*, 11(2), 2017.
[4] V. Bonnici, A. Ferro, et al. Enhancing graph database indexing by suffix tree structure. In *Pattern Recognition in Bioinformatics*, 2010.
[5] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. on PAMI*, 26(10), 2004.
[6] C. Demetrescu, D. Eppstein, Z. Galil, G. F. Italiano. Dynamic Graph Algorithms. *In Algorithms and theory of computation handbook.* Chapman & Hall/CRC, 2010.
[7] R. Di Natale, A. Ferro, et al. Sing: Subgraph search in non-homogeneous graphs. *BMC Bioinformatics*, 11(1), 2010.
[8] M. Elseidy, E. Abdelhamid, et al. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *PVLDB* 7(7), 2014.
[9] W. Fan, C. Hu, C. Tian. Incremental Graph Computations: Doable and Undoable. *In SIGMOD*, 2017.
[10] W. Fan, X. Wang, Y. Wu. Incremental Graph Pattern Matching. *TODS*, 38(3), 2013.
[11] A. Galakatos, A. Crotty, et al. Revisiting Reuse for Approximate Query Processing. *PVLDB*, 10(10), 2017.
[12] J.P. Huan, W. Wang, J. Prins. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.
[13] K. Huang, S. S. Bhowmick, S. Zhou, B. Choi. PICASSO: Exploratory Search of Connected Subgraph Substructures in Graph Databases. *PVLDB*, 10(12), 2017.
[14] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, S. Zhou. QUBLE: Towards Blending Interactive Visual Subgraph Search Queries on Large Networks. *VLDB J.* 23(3), 2014.
[15] S. Idreos, O. Papaemmanouil, S. Chaudhuri. Overview of Data Exploration Techniques. *In SIGMOD*, 2015.
[16] N. Jayaram, S. Goyal, C. Li. VIIQ: Auto-Suggestion Enabled Visual Interface for Interactive Graph Query Formulation. *In PVLDB*, 8(12), 2015.

[17] P. Jayachandran, K. Tunga, N. Kamat, A. Nandi. Combining User Interaction, Speculative Query Execution and Sampling in the DICE System. *In PVLDB*, 7, 2014.

[18] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. PRAGUE: A practical framework for blending visual subgraph query formulation and query processing. In *In ICDE*, 2012.

[19] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. Gblender: towards blending visual query formulation and query processing in graph databases. In *ACM SIGMOD*, 2010.

[20] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *In PVLDB*, 8(12), 2015.

[21] S. Kim, et al. PubChem Substance and Compound Databases. *Nucleic acids research*, 44(D1), Oxford University Press, 2015.

[22] G. Koutrika, et al. Exploratory Search in Databases and the Web. *EDBT Workshop*, 2014.

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

[24] J. Lazar, J. H. Feng, H. Hochheiser. Research Methods in Human-Computer Interaction. John Wiley & Sons, 2010.

[25] G. Marchionini. Exploratory Search: from Finding to Understanding. *Commun. ACM*, 49(4), 2006.

[26] B. D. McKay, A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60, 2014.

[27] M. Mongiova, R. D. Natale, R. Giugno, A. Pulvirenti, and A. Ferro. Sigma: A set-cover-based inexact graph matching algorithm, *In J. of Bioinformatics and Comp. Biology*, 2010.

[28] M. H. Namaki, Y. Wu, X. Zhang. GExp: Cost-aware Graph Exploration with Keywords. *In SIGMOD*, 2018.

[29] R. Pienta, F. Hohman, et al. Visual Graph Query Construction and Refinement. *In SIGMOD*, 2017.

[30] B. Sarrafzadeh, E. Lank. Improving Exploratory Search Experience through Hierarchical Knowledge Graphs. *In SIGIR*, 2017.

[31] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs. Designing the User Interface: Strategies for Effective Human-Computer Interaction. *Pearson*, 5th Edition, 2009.

[32] H. Shang, et al. Connected Substructure Similarity Search. *In SIGMOD*, 2010.

[33] T. Siddiqui, et al. Effortless Data Exploration with Zenvisage: An Expressive and Interactive Visual Analytics System. *In PVLDB*, 10(4), 2016.

[34] Y. Song, H. E. Chua, S. S. Bhowmick, B. Choi, S. Zhou. BOOMER: Blending Visual Formulation and Processing of p-Homomorphic Queries on Large Networks. *In SIGMOD*, 2018.

[35] S. Sun, Q. Luo. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. *In ICDE*, 2019.

[36] R. W. White, R. A. Roth. Exploratory Search: Beyond the Query-response Paradigm. *Synth. Lec. on Inf. Conc., Retr., and Serv. 1*, 1, 2009.

[37] M. Yahya, K. Berberich, et al. Exploratory Querying of Extended Knowledge Graphs. *In PVLDB*, 9(13), 2016.

[38] X. Yan and J. Han. gspan: graph-based substructure pattern mining. *In ICDM*, 2002.

[39] X. Yan, P. S. Yu, J. Han. Graph Indexing: A Frequent Structure-Based Approach. *In SIGMOD*, 2004.

[40] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *In ACM SIGMOD*, 2005.

[41] P. Yi, B. Choi, et al. AutoG: A Visual Query Autocompletion Framework for Graph Databases. *The VLDB Journal*, 26(3), 2017.

Appendix

Proof of Lemma 1 (Sketch). Algorithm 2 builds a VACCINE index by adding all frequent fragments one by one and connecting them together by their transformation relationships via two types of primitive transformers. So the number of vertices in a VACCINE index is the total number of frequent fragments and DIFs (*i.e.,* N). For a frequent fragment f, there are at most N_{fmax} nodes. Hence, we can add $C^2_{N_{fmax}}$ new edges for connecting current nodes in f. In addition, there are at most N_{fe} different ways to add a new frequent edge of a new labeled node to a current node in f. So it can create at most $N_{fmax}N_{fe}$ edges. Thus, there are $O(N(C^2_{N_{fmax}} + N_{fmax}N_{fe}))$ edges at most in a VACCINE index.

Proof of Theorem 1 (Sketch). In Algorithm 2, the process for creating VACCINE index can be divided into three major steps. The first step (Line 1) is to mine all frequent fragment from $\mathcal{D}$, whose time complexity is denoted as C_{ff}. The second step (Line $5-10$) is to fetch all frequent edges in $\mathcal{D}$ and store them in a matrix. Its time complexity is $O(|E|)$. The third step (Line $11-14$) is to iterate through all frequent fragments to create the index by utilizing the node and edge transformers. Assume the time complexity of the canonical labeling process is C_{cl}. Then the time complexities of node and edge transformers are $O(N_{fmax}|\mathcal{L}|C_{cl})$ and $N^2_{fmax}C_{cl})$, respectively. Hence the overall complexity is $N_fC_{cl}(N_{fmax}|\mathcal{L}| + N^2_{fmax})$. The final step (Line $15-17$) is to compute data graph identifier set of DIFs. Its complexity is $O(N_{dif}N_{dmax})$. Thus, the time complexity for building VACCINE index is $O(C_{ff} + |\mathcal{E}| + N_fC_{cl}(N_{fmax}|\mathcal{L}| + N^2_{fmax}) + N_{dif}N_{dmax})$.

Proof of Theorem 2 (Sketch). First, we prove the following lemma, which we shall be using subsequently.

Lemma 2: Given a VACCINE *index $G_I = (V_I, E_I)$, the time complexity of processing a new edge e_ℓ to the current query fragment $q = (V_q, E_q)$ is $O(|V_I|C_{CAM}+min(|V_I|, x_f)(|V_q|+|E_q|))$, where x_f is the number of frequent fragments and DIFs of q in G_I that contains e_ℓ and C_{CAM} is the time complexity of comparing the* CAM *codes of a pair of graphs.*

Proof of Lemma 2. When a new edge e_ℓ is added to the current query fragment q, Algorithm 4 first compares the CAM code of e_ℓ with all fragments in G_I to check whether the new edge is a frequent fragment or a DIF. If it is, then we can get the corresponding matching vertex for e_ℓ (Line 1). The time complexity for this task is $O(|V_I|C_{CAM})$. Next, the algorithm finds and indexes all frequent fragments and DIFs that contain e_ℓ gradually by utilizing the primitive transformers associated with the edges of the matching vertex. For each fragment, it performs three tasks: (a) compare the transformer information with all children of the matching vertex for finding the next one via MATCHINGINVACCINE function (Line 11), (b) update/add vertex for matched fragments (Lines 12–15) and its parental relationships (Lines 16), and (c) push itself to the queue (Line 17). The time complexities of these three tasks are $O(1)$ (using a suitable hash function), $O(|V_q|+|E_q|)$ (there are at most $|E_q|-1$ parent-child relationships in G_I for a fragment) and $O(1)$, respectively. The upper bound of the number of frequent fragments and DIFs is the minimum value of $|V_I|$ and x_f. Thus, the complexity of processing each new edge during query formulation is $O(|V_I|C_{CAM} + min(|V_I|, x_f)(|V_q| + |E_q|))$.

Proof of Theorem 2. From Lemma 2, we know that the time complexity for building ADVISE index by adding an edge e_ℓ to the current query graph $q_c = (V_{q_c}, E_{q_c})$ is $O(|V_I|C_{CAM} + min(|V_I|, x_f)(|V_{q_c}| + |E_{q_c}|))$ where x_f is the number of frequent fragments and DIFs of q_c in G_I that contains e_ℓ. The whole query q is formulated gradually, thus $|E_{q_c}| \leq |E_q|$ and $|V_{q_c}| \leq |V_q|$. So the worst-case cost for adding a query edge is $O(|V_I|C_{CAM}+min(|V_I|, x_{fq})(|V_q|+|E_q|))$. Because

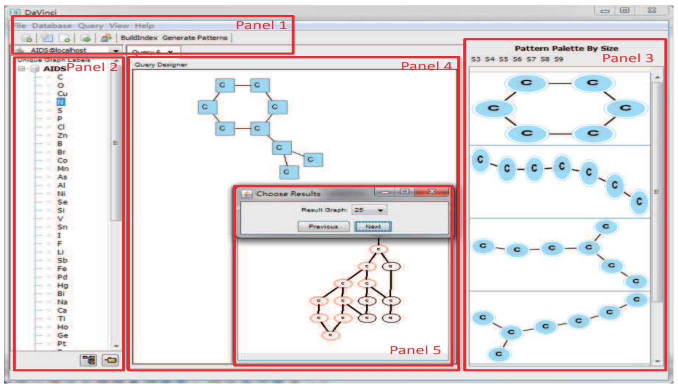


Fig. 17. **GUI of FERRARI and PICASSO.**

there are at most E_q different edges with distinct labels to be added during the formulation of q, the total time complexity is $O(|E_q| * (|V_I|C_{CAM} + min(|V_I|, x_{fq})(|V_q| + |E_q|)))$.

The upper bound of the number of vertices in G_A is the minimum value of $(|V_I|$ and $2^{|E_q|} - 1)$. Thus, the space complexity of ADVISE index is $m * min(|V_I|, 2^{|E_q|} - 1)$.

Figure 17 depicts the direct-manipulation interface of PICASSO and FERRARI. It consists of the following panels.

- An *Attribute Panel* (Panel 2) to display a set of labels or attributes of nodes or edges of the underlying data.
- A *Pattern Panel* (Panel 3) to display a set of template patterns that can aid query formulation.
- A *Query Panel* (Panel 4) for constructing a graph query graphically by leveraging the *Attribute* and *Pattern Panels*.
- A *Results Exploration Panel* (Panel 5) that displays the query results during exploration.

A typical query would be constructed using the interface by performing the following sequence of steps.

1) Move the mouse cursor to the *Attribute* or *Pattern Panel*.
2) Scan and select a label or pattern (*e.g.,* label C, benzene ring pattern).
3) Drag the selected item to the *Query Panel* and drop it. Each such action represents formulation of a single node or a query fragment in the query graph.
4) Repeat, if necessary, Steps 1–3 for constructing another node or a query fragment.
5) Construct edges (if necessary) between relevant nodes in the constructed subgraphs by clicking on them.
6) Repeat Steps 4 and 5 until the query graph is executed by clicking on the Run icon.