An Indexing Framework for Efficient Visual Exploratory Subgraph Search in Graph Databases

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
[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—**Although exploratory search has received significant attention recently in the context of structured data, scant attention has been paid for graph-structured data. In this paper, we present two novel index structures called VACCINE and ADVISE to efficiently support exploratory subgraph search in a visual environment (VESS). 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. 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. These indexes engender more efficient and scalable visual exploratory subgraph search framework compared to a state-of-the-art technique.**

Index Terms—**exploratory search, graph database, visual interface, indexing**

I. Introduction

Exploratory search [7], [11] has received increasing attention in recent times. In the database community, a growing number of efforts have focused on building search and exploration frameworks for *structured* data (*e.g.,* relational) [5]. However, there is a dearth of work for realizing such search paradigm on graph-structured data [4], [9], [12]. Exploratory graph search 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 underlying data space.

We advocate that it is paramount to provide a visual 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 connected subgraphs) for efficient formulation of query components.

PICASSO [4] is the first tool to crystallize *visual exploratory subgraph search* (VESS) on graph databases centered around a collection of small- or medium-sized data graphs. It leverages the visual subgraph query processing framework of [6] to evaluate each formulation/reformulation incrementally by *blending* (*i.e.,* interleaving) visual query construction and processing.

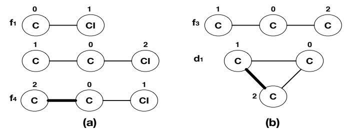


Fig. 1. **Examples of node and edge transformers.**

Our initial investigation, however, revealed that the visual subgraph query processor [6] of PICASSO is not efficient and scalable to support iterative query evaluation in a VESS environment as it was originally designed for look-up (*i.e.,* "one-shot") queries. Specifically, this is due to the following two scenarios that are prevalent in a VESS environment: (a) the initial query graph evolves to a large query during exploration and (b) template patterns may be leveraged during query 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 [6] for *each* newly constructed edge at *each* formulation step within the available GUI *latency* (*i.e.,* time to construct a query edge visually). That is, a query graph with n edges creates n SPIGs in PICASSO. Although this is effective in a look-up querying environment where query graphs are typically small in practice [1], the VESS environment demands efficient blending of a larger set of edges (*e.g.,* template patterns) to support larger query graphs. In particular, 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 inefficient in PICASSO. Hence, it is important to design novel indexes that can efficiently support the VESS paradigm.

In this paper, we present two novel indexing schemes, VACCINE and ADVISE, for a VESS framework to address the aforementioned challenges. VACCINE is an offline, *feature-based* index structure. It stores rich 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. 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, only one instance of ADVISE is necessary to support VESS. Here we present an overview of our indexing framework. Details on various aspects of this framework are available at [10].

II. Background

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.

A graph G is a *subgraph* of 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 say that G' *contains* G. G is called a *proper subgraph* of G', denoted as $G \subset G'$, iff $G \subseteq G'$ and $G \not\supseteq G'$.

Given 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 [6]. 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 $fsgIds(g)$.

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. Specifically, we classify infrequent fragments into two types, *discriminative* and *non-discriminative* [6]. Given $g \in \mathcal{I}$, let $sub(g)$ be the set of all subgraphs of g. If $sub(g) \subset \mathcal{F}$ or $|g| = 1$, 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 [6].

Lastly, we introduce the following set of GUI *actions* (*actions* for brevity) that a user takes to formulate an exploratory subgraph search query in any VESS interface: (a) 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 q, and returns the augmented query. (b) modify(q,g): This action denotes that a user revokes (deletes) a query fragment or a node g, and returns the modified query. (c) run(q): The run action models the execution of a query fragment by clicking on the Run icon (or equivalent of Run) in the GUI.

We refer to a sequence of such GUI actions taken by a user as *exploration action sequence* (EAS). 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).

Remark. We do not model low level actions (*e.g.,* mouse click, mouse hover, drag-and-drop) as different GUIs may follow different sequences of low level actions to realize the aforementioned actions. Consequently, this enables us to design a visual exploratory subgraph search framework that *underpins any* GUI.

III. 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 a 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).*

At each run(q), it is imperative for our framework to support *fast* subgraph containment or subgraph similarity search [13] of q. In particular, similar to [4], [6], we adopt maximum connected common subgraphs (MCCS)-based *subgraph similarity distance*, denoted as $dist(G, Q)$, to measure similarity between a data graph G and a query graph Q.

Definition 1: Let $\mathbb{A}$ be an EAS *undertaken by a user on a visual 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.*

We utilize the visual interface of PICASSO [4] for VESS. Similar to PICASSO, we interleave (*i.e.,* blend) the visual 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 present novel offline and online indexes to facilitate efficient evaluation of run(q). Note that the indexing framework is not tightly coupled to any specific VESS GUI. Any superior GUI can be used on top of our indexes.

IV. VACCINE Index

In this section, we briefly 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.

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 ℓ. 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 $(v_i, v_j) \in E'$.

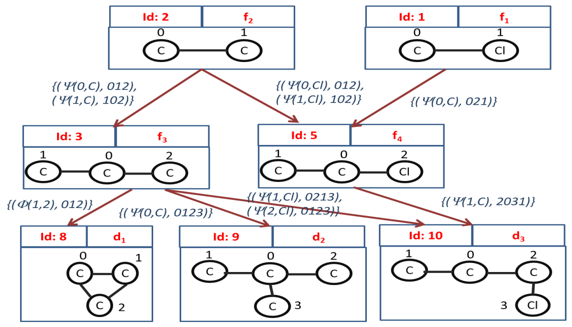


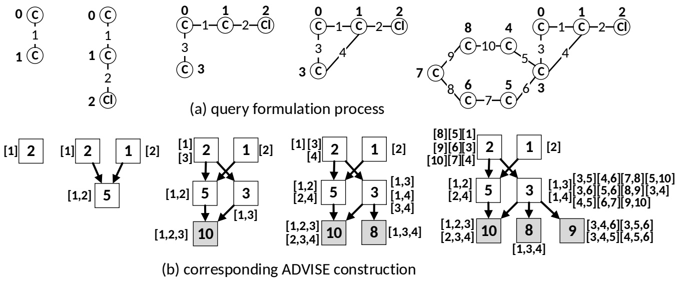
Fig. 2. **VACCINE index. For clarity, we do not show the CAM code and $\mathcal{L}_g$ associated with each vertex.**

Example 1: Consider the graphs f_1 and f_4 in Figure 1(a). Assume that these fragments are frequent or DIF. Then f_4 is generated from f_1 by utilizing $\Psi_{f_1}(0, \texttt{C})$. 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 1(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). ∎

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. That is, they capture the add(q,g) action taken by a user in a QFS where g is an edge.

Given $\mathcal{D}$ 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 [3] of g, and $\mathcal{L}_g$ is a set of identifiers of the data graphs which are subgraph isomorphic to g (*i.e.,* $\forall g'_i \in \mathcal{L}_g, g \subseteq g'_i$). 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 [8] from $n_1.g$ to $n_2.g$. Consider the two 3-nodes graphs in Figure 1(a). The canonical labeling from the middle graph to f_4 (in bottom) is $(\texttt{0,2,1})$ (*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). We also refer a vertex in VACCINE as a *leaf* if it has no outgoing edges.

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


(a) query formulation process
(b) corresponding ADVISE construction
Fig. 3. **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.**

V. ADVISE Index

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

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. Let $q = (V_q, E_q)$ be a visual graph query fragment on $\mathcal{D}$ 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: (a) 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 $id(n_m) = n.id$. (b) 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 $|f(m)| = |f(m')|+1$. (c) 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 $f(m)$.

We describe the construction of the ADVISE index due to add actions with an example in Figure 3. Its maintenance strategy due to modify actions is given in [10]. Since a user may add an edge or a template pattern during an add action, we consider the latter as an *edge stream* (a collection of edges). Consequently, a single new edge (referred to as *seed*) is processed iteratively to construct the index. Suppose we use the VACCINE index in Figure 2 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.

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 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 3 and 1. 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 user adds the benzene ring pattern. Each edge in this pattern is added sequentially into the query automatically and is processed by following the aforementioned strategy. *Observe that we create only one instance of* ADVISE.

VI. Supporting Run Action

Since q for each run action can be either a subgraph containment or subgraph similarity search [4], [13] we briefly discuss how they can be realized by leveraging the indexes. Our strategy follows the PICASSO approach [4] but replaces its indexing schemes with more efficient VACCINE and ADVISE. For subgraph containment search, if a query graph is a frequent fragment or a DIF, then its matches can be directly retrieved from the 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 [2]. On the other hand, 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 [4], [6]. In our framework, we utilize our more efficient ADVISE index instead of SPIGs.

VII. Summary of Performance Study

We have investigated the performance of our indexes in the VESS environment with real-world datasets containing up to 1.3 million data graphs. 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. 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.,* 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.
- 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 indexing techniques reduces with increasing dataset size.

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] A. Bonifati, W. Martens, T. Timm. An Analytical Study of Large SPARQL Query Logs. *In PVLDB*, 11(2), 2017.

[2] L. Cordella, et al. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE PAMI*, 26(10):1367–1372, 2004.

[3] J.P. Huan, W. Wang, J. Prins. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.

[4] K. Huang, S. S. Bhowmick, S. Zhou, B. Choi. PICASSO: Exploratory Search of Connected Subgraph Substructures in Graph Databases. *PVLDB*, 10(12): 1861-1864, 2017.

[5] S. Idreos, O. Papaemmanouil, S. Chaudhuri. Overview of Data Exploration Techniques. *In SIGMOD*, 2015.

[6] C. Jin, et al. PRAGUE: A practical framework for blending visual subgraph query formulation and query processing. In *In ICDE*, 2012.

[7] G. Marchionini. Exploratory Search: from Finding to Understanding. *Commun. ACM*, 49(4): 41-46, 2006.

[8] B. D. McKay, A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60:94 – 112, 2014.

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

[10] C. Wang, et al. FERRARI: An Efficient Framework for Visual Exploratory Subgraph Search in Graph Databases. *Technical Report*, http://www.ntu.edu.sg/home/assourav/TechReports/FERRARI-TR.pdf, 2018.

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

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

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