SuMExplorer: Summarisation-based Frequent Subgraph Mining for Visual Exploratory Subgraph Searching

Chimi Wangmo[1], Lena Wiese[1]

[1]Goethe University Frankfurt, Robert-Mayer-Str. 10, 60629, Frankfurt am Main, Germany

Abstract

Graphs are ubiquitous with their vast application in various disciplines, including bioinformatics, security, and social sciences. One of the most popular queries involves searching for subgraphs and similarities on large numbers of small-to-medium-sized graphs. The visual exploratory subgraph search is proposed to be a search paradigm to support visual analysis and to ease user interaction. In this paper, we address the problem of reducing computational complexity during the indexing stage of visual exploratory subgraph searching. We propose (1) a compact summarisation of our input graph to reduce the index size and construction time. In addition, (2) we introduce weighted frequent subgraph mining for our novel graph summarisation-based visual subgraph searching. Based on these, (3) we propose a novel index structure to represent the subgraphs extracted from the summary graphs in a compact form. Finally, (4) we empirically demonstrate that the proposed framework provides higher efficiency than the baseline by benchmarking on three real-world biological datasets.

Keywords

visual exploratory subgraph searching, graph summarisation, graph indexing, graph database

CEUR-WS.org/Vol-3630/LWDA2023-paper10.pdf

1. Introduction

Subgraph containment and similarity searching are critical problems in analysing social networks, computational biology, collaboration networks, etc. In the case of subgraph containment, our interest lies in finding all the labelled graphs that contain a given query graph i.e. a subgraph to a given supergraph. In subgraph similarity, earlier works such as [1] have investigated the concept of missing edges and the need for a distance measure to identify similar graphs. The naive approach to solving this problem involves adopting a subgraph isomorphism algorithm. However, [2] proved that performing vertex-to-vertex mapping in the subgraph isomorphism problem is *NP-complete*. As a result, we introduce subgraph-based graph indexing to prune unrelated candidate labelled graphs and perform subgraph isomorphism (also known as "verification"). We call this process subgraph-based graph indexing because it requires embedding subgraphs within the graph database to filter non-candidate labelled graphs.

LWDA'23: Lernen, Wissen, Daten, Analysen. October 09–11, 2023, Marburg, Germany

*Corresponding author.

†These authors contributed equally.

wangmo@uni-frankfurt.de (C. Wangmo); lwiese@cs.uni-frankfurt.de (L. Wiese)

http://wiese.free.fr/ (L. Wiese)

0000-0002-8921-398X (C. Wangmo); 0000-0003-3515-9209 (L. Wiese)

© 2022 Copyright © 2023 by the paper's authors. Copying permitted only for private and academic purposes. In: M. Leyer, Wichmann, J. (Eds.): Proceedings of the LWDA 2023 Workshops: BIA, DB, IR, KDML and WM. Marburg, Germany, 09.-11. October 2023, published at http://ceur-ws.org.

CEUR Workshop Proceedings (CEUR-WS.org)

The use of current graph databases require knowledge of either programming or query language; hence the usage of modern databases is limited to specific, knowledgeable users. On the other hand, users such as chemists may lack an understanding of complex knowledge regarding query formulation using a particular query language. As a result, visual exploratory subgraph searching helps to overcome such issues [3]. To summarise, the manipulation operations available to users in visual exploratory subgraph searching comprise: (1) the addition of a new vertex and its connection to an existing vertex or the addition of a new edge between existing vertices, (2) the deletion of either an edge or of both a vertex and the corresponding edges, and (3) finally, the execution of query to find a matching subgraph in the transactional graph database. Operations (1) and (2) lead to modifying a query graph and, hence, require a maintenance algorithm for an index structure. Visual exploratory subgraph searching relies on the inner workings of the index structure to perform the subgraph query processing efficiently. We have looked, in particular, at solving issues related to the efficiency of index construction and memory usage and improving the effectiveness of filtering power and query processing.

In this paper, we propose a new framework for efficiently representing the graph as a summary graph and improving pruning power using weighted-frequent subgraph mining (Sec. 4). We introduce frequent subgraph mining on the summary graph with weighting functions. We devise a new index structure, called the **W**eighted-**Index** (W-Index), for compactly storing frequent and discriminative in-frequent subgraphs in the summary graph (Sec. 5 and 6). Finally, we verify the efficiency of the proposed solutions with extensive experiments (Sec. 7).

2. Notations and Problem Statement

In this section, we first present the fundamental concepts related to graph theory, focussing on identifying the topological components that correspond to real-world entities. To maintain consistency, we use the term "vertex" to refer to an entity in the graph, while the term "node" is reserved for objects in the index graph. We now formally define the *labelled graph* as follows.

Definition 1 (Labelled Graph). *A* labelled graph *is a triplet* $G = (V_G, E_G, L_G)$ *where* V_G *denotes the set of vertices,* $E_G \subseteq \{\{u, v\} \,|u, v \in V_G\}$ *is the set of edges, and* $L_G : V_G \mapsto \sigma_G$ *assigns a label to a vertex where* $\sigma_G = \{\sigma_G^{(1)}, \sigma_G^{(2)}, ...\}$ *is the set of labels. We refer to the number of edges* $|E_G|$ *in the given graph* G *as its size and the number of vertices and labels as* $|V_G|$ *and* $|\sigma_G|$*, respectively.*

We work with a *transactional graph database*, which contains a set of n medium-sized labelled graphs, denoted as $\mathcal{G} = \{G_1, G_2, G_3, \dots, G_n\}$. Further, we denote the cardinality of the graph database as $|\mathcal{G}| = n$, the average number of vertices, edges, and labels of all the labelled graphs in the *transactional graph database* as $|V_{\mathcal{G}}|$, $|E_{\mathcal{G}}|$, and $|\sigma_{\mathcal{G}}|$, respectively. In this paper, our motivation is to query for a *query graph*; this is a subgraph of a labelled graph in the *transactional graph database.* Hence, it is necessary to understand and conduct *subgraph isomorphism* iteratively for each labelled graph in the graph database to solve the query. Yet, the major downside of subgraph isomorphism is its expensive operation time.

Definition 2 (Subgraph Isomorphism [4]). *Given a query graph, Q, and a labelled graph, G, an embedding of Q in G is a mapping, where* $M : V_Q \to V_G$ *such that: (1) M is injective, i.e.*

$M(u) \neq M(v)$ for $u \neq v \in V_Q$, (2) the labels match, i.e. $L_Q(u) = L_G(M(u))$ for every $u \in V_Q$, and (3) and all edges of Q exist in G, i.e. $(M(u), M(v))(or(M(v), M(u))) \in E_G \ \forall(u, v)(or(v, u)) \in E_Q$. Graph Q is subgraph-isomorphic to G, denoted by $Q \subseteq G$ if there exists an embedding of Q in G.

Example 1. *In Fig. 1 (a), we illustrate an example of labelled graphs representing molecules in the transactional graph database, denoted by $\mathcal{G}$. The transactional graph database contains 5 molecular graphs, $\mathcal{G} = \{G_1, G_2, G_3, G_4, G_5\}$. Each labelled graph contains several vertices (12 in G_1, 5 in G_2, 6 in G_3, 8 in G_4, and 3 in G_5) denoted by v_l where $1 \leq l \leq |V_G|$, and its corresponding label denoted by a letter, $\sigma_l \in \sigma$; here, $\sigma = \{C, H, O\}$. The vertex v_1 in G_1 contains label C. Furthermore, there exists an edge from one node to another, such as an edge from v_1 to v_2.*

2.1. Problem Statement

Prior to providing a formal definition of *visual exploratory subgraph search* (*VESS*), we formally define the query graph in VESS, Q_t, as follows.

Definition 3 (Query Graph in VESS). *The* query graph, *Q_t, is a labelled graph (see Definition 1), which we start with an insertion of a single edge, $Q_1 = (V_{Q_1}, E_{Q_1}, L_{Q_1})$, where $E_{Q_1} = \{e_{Q_1}\}$. We then sequentially perform incremental updates based on the user's update operation, op, on the source and destination vertices, denoted by v_s and v_d, respectively. The user can perform an operation on the previous instance of the query graph at time $t-1$, where $1 \leq t \leq \mathcal{N}$. Here, op is either "+" or "-", indicating an add or remove operation, respectively. The resulting graph from the addition of a v_{Q_t} and an edge, e_{Q_t}, to Q_{t-1} is $Q_t = \{V_{Q_t}, E_{Q_t}, L_{Q_t}\}$. It has V_{Q_t} nodes and E_{Q_t} edges.*

Definition 4 (Visual Exploratory Subgraph Search). *Given a batch of update operations, B, on an initial query graph Q_1, and a transactional graph database, $\mathcal{G}$, the* visual exploratory subgraph search *(VESS) finds all the graphs, $G_i \in \mathcal{G}$, that contain the query graph, $Q_t = \{B, Q_1\}$.*

Example 2. *In Fig. 1 (b), we provide an example of exploratory subgraph searching in a transactional graph database. In this figure, we have denoted time with t_i, where $1 \leq i \leq \mathcal{N}$. Initially, the user will begin the exploratory search with a single-edge addition at time t_1. The exploratory search involves a series of add() operations on new vertices and connections to existing vertices, as observed in time $t_2, t_3, t_5, t_8, t_9, t_{10}, t_{11}$. In addition, the add() operation may be simply an addition of a new edge between two existing vertices, as observed at time t_6. Furthermore, the user may also remove certain edges, as seen at time t_4 and t_7. Finally, after some time, say t_{11}, the user may issue the run() operation to find answers to the searched query.*

In this paper, we focus on the subgraph containment problem formally defined as follows.

Definition 5 (Subgraph Containment). *Given a query graph Q and labelled graphs, $G_i \in \mathcal{G}$, the* subgraph containment (also called "Subgraph search") *problem is to find all the labelled graphs that contain Q.*

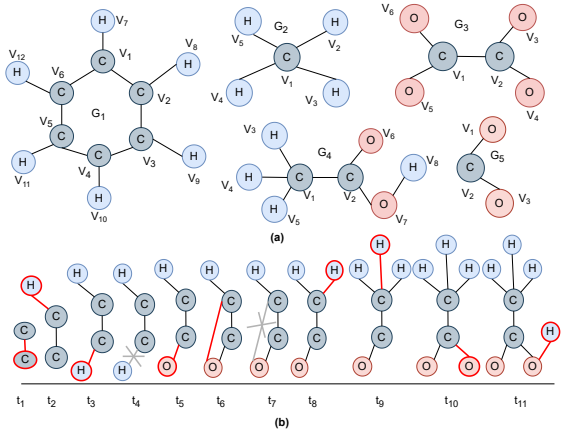

Figure 1: Running example of (a) a transactional graph database and performing (b) visual exploratory searching through add(), modify(), and run() operations. We have highlighted (red for add() and grey for modify()) the vertices and edges to indicate the current modification made.

3. Related work

In a graph database system, one of the main components of management is query processing, which includes an operation to facilitate efficient searching of the query graph. The **P**rogress**I**ve **C**onnected Subgr**A**ph **S**ubstructure **S**earch T**O**ol (PICASSO) [3] platform is the first experiment that supports the VESS. The framework was mainly utilised for lookup operations, primarily emphasising streams of edges forming the query graph. PICASSO proposed two index structures namely, the *action-aware frequent index* and *action-aware infrequent index*, denoted as A^2F and A^2I, respectively. A^2F indexes frequent subgraphs while A^2I indexes discriminative infrequent subgraphs. Furthermore, the author(s) of PICASSO maintain another online index known as *SPIG*. The work requires the construction of *SPIG* sets, which are mainly time and memory intensive due to the underlying index structure, for every query formulation.

The PICASSO indexes exhibits a large size and requires a significant amount of time for construction. This is because of the many subgraphs (ranging from a few thousand to millions of molecular compounds) in the transactional graph database. In particular, memory consumption increases greatly with the increasing numbers of labelled graphs present in the transactional graph database. Additionally, the index construction time is substantial due to the requirement of detecting numerous frequent subgraphs within each labeled graph. In the case of pruning power, the prior index structure treats each subgraph as a disconnected component. As a result, we lose the neighbourhood information associated with a frequent subgraph, despite it containing structural information. To avoid expensive graph operations and space usage, we adopt graph summarisation to represent a general graph and perform weighted frequent subgraph mining on the summary graphs.

4. The SuMExplorer Architecture

4.1. The Overall Framework

Our novel framework, **Sum**marisation-based frequent subgraph mining for visual **exploratory** subgraph searching, *SuMExplorer*, consists of six modules: (1) a *user-interface module*, where we provide a visual interface for a user to formulate queries through add(), modify() and run() operations, (2) a *graph loader*, representing the textual format in an appropriate graph structure, (3) a *graph summariser*, representing the input graph structures in the summarised form, (4) a *frequent subgraph miner*, mining frequent subgraphs efficiently, (5) a *graph Indexer*, storing the frequent and in-frequent subgraphs of the summarised graphs, and (6) a *manipulator*, performing update operations on a query graph. In Section 5 and 6, we will provide the discussion on modules (5) and (6). The SuMExplorer is depicted in the Fig. 2. In this paper, we propose a

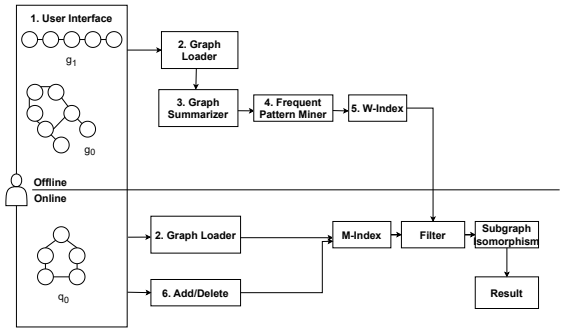

Figure 2: SuMExplorer Framework.

novel method of utilising a *summary graph* for minimal memory usage by indexes and faster query processing. Firstly, we will provide a formal definition of the *intermediate summary graph* (see Definition 6), which we obtain from the general graph represented by the *graph loader* module. The *graph summariser* module converts a simple, undirected, labelled graph to its intermediate summary graph through "vertex-based summarisation".

Definition 6 (Intermediate summary graph). *Let $G = (V_G, E_G, L_G)$ be a labelled graph. We denote an* intermediate summary graph *of G as $IG' = (V_{IG'}, E_{IG'}, L_{IG'}, W_{IG'})$. The set $V_{IG'}$ comprises the* supernodes *containing distinct vertices, $V_{IG'} \subset V_G$, and we refer to the edges between these supernodes as* superedges*, $E_{IG'}$. $L_{IG'} : V_{IG'} \mapsto \sigma_{IG'}$ assigns a label to a vertex where $\sigma_{IG'} = \{\sigma^{(1)}_{IG'}, \sigma^{(2)}_{IG'}, ...\}$ is the set of labels. With the graph summarisation algorithm, we assign the weights on the* superedges *as $W_{IG'}$. The weight on edge, $W_{IG'}$ is a real number that indicates the number of merged vertices sharing a common property and that are connected to the same destination vertex. The $W_{IG'}$ is a metric that shows the interconnectivity and, thus, the importance of the nodes in the graph.*

We generate an intermediate summary graph for each graph in the transactional graph database. Given a labeled graph in the transactional graph database, $G \in \mathcal{G}$, we can now prove the following result.

Lemma 1. *The size of an intermediate summary graph IG' is at the most the size of the original graph, $|IG'| \leq |G|$.*

We will use SSumM, the *Sparse Summarisation of Massive Graphs [5]* proposed to obtain a space-efficient summary graph with a lower reconstruction error. The reconstruction error is the difference between the original graph G and the reconstructed graph from the summary graph IG'. The *intermediate summary graph* can provide compactness and structure preservation by reducing nodes and edges, however, the result of the intermediate summary graph is an edge-weighted multi-graph. This is an issue since there need to be more studies for frequent subgraph mining on the multi-weighted graph. Due to the space constraints, we omit computing the weighted frequent mining here.

Our index structure is responsible for maintaining the extracted weighted frequent and in-frequent subgraphs (or edges). We have two index structures designed to facilitate efficient query processing. The first is an offline-based index structure, which we call a **W**eighted-**Index**, denoted by W-Index. In W-Index, we index the subgraphs extracted from the labelled graphs within the transactional graph database. We build this index only once. The second index is an online-based index structure, which we call a **M**aintainable-**Index**, denoted by M-Index. In the case of M-Index, the input graph is incrementally built from a single edge through the add() operation induced by the end user. Therefore, we require an index structure to maintain the existing result in order to facilitate efficient query processing efficiently. We provide the detailed discussion in Section 6.

5. W-Index

In this section, we will discuss the offline-based index structure, W-Index, and its index construction process. The W-Index is a hybrid Trie-based data structure that efficiently supports subgraph search queries by using subgraph containment searches and the intersection of graph identifiers. Each indexed node, denoted by $\gamma_u \in \gamma$, represents frequent (discriminative in-frequent) subgraphs. We formally define the indexed node, $\gamma_u \in \gamma$, as follows.

Definition 7 (Index node in W-Index). *The indexed node in the W-Index is a quadruple $\gamma_u = \langle id, C_h, p, G_I \rangle$. In order to identify uniquely the indexed node, we assign a unique identifier to the indexed node, denoted as id, based on its insertion order. Now, to support the storing of the frequent (or in-frequent) subgraphs, we store the label, denoted as C_h. C_h is a canonical labelling of the extracted subgraphs, which we represent using the Depth First Search (DFS) code. DFS code is a graph sequentialisation method, which represents a graph as a string sequence translating to the edge sequences in the graph. We can obtain DFS code through the depth-first search method and is ordered based on the lexicographic order of the vertices label. We chose the minimal DFS code for this study. Furthermore, we store a boolean value p in each node, which we use to differentiate between frequent and in-frequent subgraphs (or edge). Finally, we store the encoded list of graph identifiers G_I associated with the extracted subgraphs.*

In order to define the tree formally, we must first define the necessary conditions that constitute the W-Index. Therefore, let us denote the set of indexed nodes in the tree, denoted as $\mathcal{T}$, as $U \in \mathcal{T}$. We denote the total number of index nodes in the W-Index as N_w.

Definition 8 (W-Index). *We call a tree $\mathcal{T}$ a W-Index iff $\mathcal{T}$ satisfies the six conditions: (1) The root of $\mathcal{T}$ has id $= 0$ and C_h as an empty sequence ( ) with no G_I. (2) The subgraphs h_j inserted in the $\mathcal{T}$ contain weights and are frequent, which we denote by DF, and in-frequent subgraphs, denoted by DI. (3) The index nodes contain p as either 0 or 1, which we can use to differentiate between frequent and in-frequent subgraphs. (4) The indexed nodes $\{u_1, u_2, \dots, u_s \in U\}$ follow the lexicographic order, where $1 \leq s \leq N_w$. (5) The size of the subgraph in the indexed node is level−1. (6) Similar to general trees, leaf nodes have no children.*

By default, we have the W-Index as a rooted tree as described earlier. The insertion of the new indexed nodes includes the checking of whether the indexed nodes are either frequent or in-frequent subgraphs; hence, we will assign their p as either 0 or 1. We can create a new indexed node in the W-Index by adding a single edge in the level 1 of the tree and we update the W-Index by adding more labelled graphs. Indexing includes the canonical labelling of the frequent (or in-frequent) edges and insertion of the entries to the identifier of the labelled graph (denoted as G_I).

Theorem 1. *Given an intermediate summary graph IG_i' for each labelled graph in the transactional graph database, assumes that there are discriminative Num_{DF} frequent and Num_{DI} infrequent subgraphs. Assume that the time complexity of generating an intermediate summary graph is $T_{IG_i'}$, then consider that the time complexity for frequent subgraph mining is T_h, and the time complexity for canonical labelling is T_c. We denote the maximum number of edges in the intermediate summary graph by $E_{IG_i'}$. Now, consider that the maximum number of vertices in the frequent subgraph is M_{DF}. Therefore, the overall time complexity for building the W-Index is $\mathcal{O}(|\mathcal{G}| \cdot T_{IG_i'} + |E_{IG_i'}| + T_h + Num_{DF} \cdot T_c \cdot (M_{DF}^2) + Num_{DI} \cdot T_c)$.*

6. M-Index

In this section, we describe an auxiliary structure called the M-Index, which is an online-based structured organisation that we build dynamically through various graph update operations such as add() and modify(). M-Index is a maintainable index that aims to enhance the query processing during the refinement of an evolving query graph and numerous query executions. The main idea is to store compactly the partial results in the context of the currently executed query, which will then aid in the faster processing of future queries. Now, let us turn to the structural representation of an index node in the M-Index. Note here that the query graph undergoes refinement on one edge at a time. We formally define an index node in the M-Index as follows.

Definition 9 (Index node in M-Index). *The indexed node $\lambda_t \in \lambda$ in the M-Index is a tuple $\lambda_t = Q_t = \langle \varepsilon(Q_t), eid \rangle$, where Q_t is a new query graph resulting after the addition or modification of a new edge at timestamp t. We denote the total number of indexed nodes in the M-Index as $\mathcal{N}$. Each indexed node stores the identifier of the indexed node in the W-Index, whose C_h matches the canonical labelling of the query graph. As a result, we need a node matching function $\varepsilon(Q_t) = \chi(Q_{t-1}, op, v_s, v_d) = \gamma_{id}$. Furthermore, each indexed node in the M-Index contains a set of identifiers of the edges denoted as eid.*

Observe that the complete set of an indexed node generates the M-Index, which is merely the whole query graph at the time stamp Q_t. We now present the formal definition of the M-Index.

Definition 10 (M-Index). *M-Index is a rooted tree, $\mathcal{T}$, with a tuple $\langle \lambda, M \rangle$. The tree $\mathcal{T}$ is an M-Index, iff $\mathcal{T}$ satisfies the two conditions: (1) The index node λ is a set of unique nodes that correspond to the instance of the query graph Q_t. (2) The edge $m \in M$ indicates the parent-child relationship where we state that the child node derives from the parent due to an update operation on the parent node.*

We then run the operation to re-examine the subgraph query problem in the context of the partial results stored in the M-Index. The visual exploratory subgraph search problem, VESS, is now a variant of the *workload-aware subgraph search.* Formally, we define the *workload-aware subgraph search* as follows.

Definition 11 (Workload-aware subgraph search). *First, let us consider a new query Q_t obtained after the exploration process, an offline-based index, the W-Index, and an M-Index. The* workload-aware subgraph search *is to find the identifier of the indexed node in the W-Index using the matching node function, $\chi(Q_{t-1}, op, v_s, v_d)$, in the presence of Q_{t-1}.*

Therefore, we can interpret the execution of the query graph at a particular time as the intersection of all the graph identifiers contained in each leaf node of the M-Index tree.

Theorem 2. *Given the graph index W-Index, for each new edge addition to the query graph, which is Q_{t-1}, the time complexity of building the query index, M-Index is $\mathcal{O}(log|\lambda| \cdot T_c + min(|\lambda|, N_Q)(|V_{Q_t}| + |E_{Q_t}|))$. Assume that a user adds a new edge, denoted as e_{Q_t}, to the current query graph, denoted as Q_t. We will have to check for the matching node in the W-Index that has the same canonical label C_h as the DFS code generated for the new edge, $C_h(e_{Q_t})$. Hence, we would have to perform a $\mathcal{O}(|\lambda| \cdot T_c)$ canonical label comparison. Then, we would need to add an identifier of the matching index node present in the W-Index, denoted as id, to a new index node λ_t in the M-Index. Following that, we extend the λ_t by searching for connections from Q_t, which is the upper bound of the frequent fragments, $min(|\lambda|, N_Q)$. Now, it requires the addition or updating of a matching index node in M-Index with the complexity of $(|V_{Q_t}| + |E_{Q_t}|)$.*

6.1. Comparison with PICASSO and gIndex

In contrast to PICASSO, our SuMExplorer exploits the graph summarisation method to reduce the computational cost associated with large numbers of redundant vertices having the same properties (such as labels) and their connecting edges. Further, PICASSO utilise unweighted frequent subgraph mining, whereas SuMExplorer introduces the concept of weighted subgraph mining which improves the pruning power of frequent subgraph mining. In our index structure, W-Index, we capture the weighted frequent and infrequent subgraphs in a suffix tree. In addition, we employ a compressed bitmap to reduce the filtering time and indexing size. We observe that PICASSO does not address these issues. While PICASSO are limited to look-up operations, our SuMExplorer stores meta information regarding query formulation in the offline-based index structure, W-Index. Since PICASSO does not consider this meta information, they have to build a query index for each edge addition; this increases both storage size and building time.

7. Evaluation

In this section, we evaluate the performance of competing algorithms on some chemical datasets. Specifically, our experimental study aimed to answer the research question: how do the index size and construction time of our SuMExplorer compare to the state-of-the-art PICASSO?

We obtained the source code of *SSuM* from [5]. Our own SuMExplorer is implemented in Java, and its source code is available online [6]. Hence, we compare our work to PICASSO, which is also implemented in Java. We ran our experiment on a machine with Seven Intel Core 1185G7@ 3.00 GHz 1.80 GHz CPUs and 32 GB of RAM under Windows.

We used three real datasets, namely antiviral screen (AIDS) dataset ($|\mathcal{G}| = 40,000, |V_{\mathcal{G}}| = 45, |E_G| = 46.95, |\sigma_{\mathcal{G}}| = 62$) [7], P388 dataset ($|\mathcal{G}| = 41,472, |V_{\mathcal{G}}| = 45, |E_{\mathcal{G}}| = 41.8, |\sigma_{\mathcal{G}}| = 73$), and Yeast dataset ($|\mathcal{G}| = 79,601, |V_{\mathcal{G}}| = 45, |E_{\mathcal{G}}| = 40.7, |\sigma_{\mathcal{G}}| = 75$) to show the efficiency of our indexing algorithm. We received the remaining datasets (P388 and Yeast) from [8]. First, we parsed the chemical graphs using SmilesGraphParser and then serialise them into LineGraphParser format. Both the graph parsers are available in the *ParMol* package [9]. Then, we obtained frequent and infrequent subgraphs using our weighted frequent subgraph mining algorithm. In contrast to our approach of weighted frequent subgraph mining, PICASSO employed gSpan algorithm without assigning weights to the edge labels.

Now, we will describe the workload used for the evaluation of query processing. We utilised queries $Q_1 \rightarrow Q_4$ for the AIDS dataset, $Q_5 \rightarrow Q_6$ for the P388 dataset, and $Q_7 \rightarrow Q_8$ for the Yeast dataset. The queries are illustrated in Fig 3. Regarding the parameter selection for benchmarking, in the case of the graph summarisation module, we set the reconstruction parameter as the default value, as suggested in the original paper [5]. Furthermore, we set the minimum support, $min_{sup} = 0.1 \cdot |\mathcal{G}|$, as suggested by PICASSO.

Figure 3: Query workload for visual exploratory subgraph searching. The edge label indicates the sequence of insertion.

Next, we present the results using two metrics: index construction time, measured in milliseconds (*ms*), and index size, in megabytes (*MB*). To verify our claim that our W-Index is an efficient approach for reducing computational resources, we compared its index size and time to that of PICASSO. Our method utilises a summary graph with fewer vertices and edges and employs weighted frequent subgraph mining to improve its effectiveness. The results of our comparison demonstrate that our approach is, indeed, efficient, as evidenced by its smaller index size and faster building time. By demonstrating the efficiency of our W-Index, we can highlight that using an efficient graph summarisation and an effective weighted frequent subgraph mining can significantly impact the index building time. In Fig. 4 (a), we have shown the index time of our W-Index compared to PICASSO's A^2F and A^2I for the above-mentioned datasets. As observed in Fig. 4 (a), the index building time for W-Index is faster than A^2F and A^2I for all the datasets. The usage of optimisation such as graph summarisation and weighted frequent mining led to faster construction time and memory cost for W-Index in comparison to PICASSO. In Fig. 4 (b), we

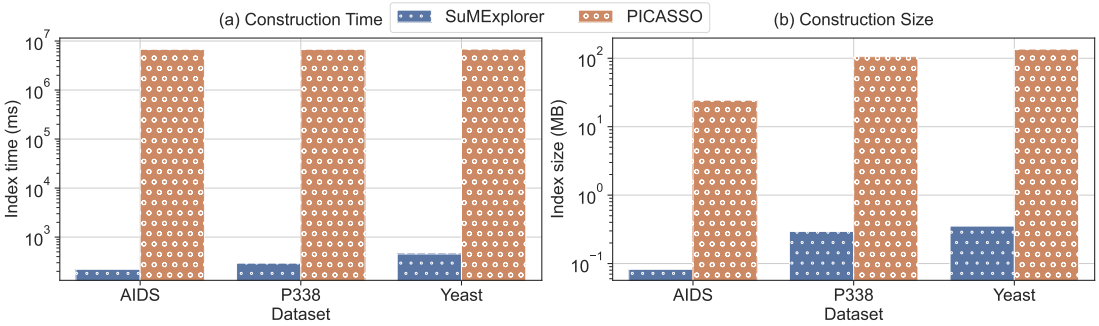

Figure 4: Comparison of: (a) construction time (*ms*) and (b) index size (*MB*) for various techniques (SuMExplorer and PICASSO) using a log-scale.

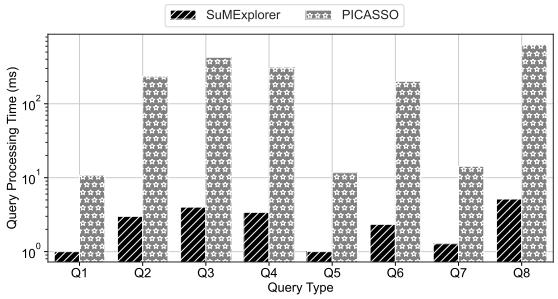

Figure 5: Comparison of query processing time (ms) for various techniques (SuMExplorer and PICASSO) on different query workloads using a log-scale.

presented the memory cost of the aforementioned index structures. We observed that W-Index requires noticeably less memory than PICASSO. Finally, we evaluated the performance of the query processing time when the user invoked the *run*() operation (Fig. 5).

8. Conclusions

In this paper, we have presented SuMExplorer, a framework for performing iterative subgraph searching on the summary graph. We have also introduced the notion of embedding subgraph-neighbourhood information to improve the pruning power of the frequent subgraph mining algorithm. Our results from diverse biological datasets show that the W-Index outperform PICASSO in terms of efficiency and scalability for all the three datasets. Specifically, we have extensively discussed generating summary graphs and indexing them in a compact representation known as the W-Index. In addition, we have provided an algorithm to update partial results and to detect changes in order to maintain an iterative query graph. In future work, we plan to extend the current framework for a similarity search query and use the notion of a graph neural network for continuous subgraph searching. An extensive evaluation of the M-Index will be produced in the extended work.

Acknowledgments

The authors would like to thank Deutscher Akademischer Austauschdienst (DAAD) for providing funds for the research on this project.

References

[1] R. H. Choi, C. Chung, Efficient processing of graph similarity search, World Wide Web 18 (2015) 633–659. doi:10.1007/s11280-014-0274-4.

[2] J. R. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM 23 (1976) 31–42. doi:10.1145/321921.321925.

[3] K. Huang, S. S. Bhowmick, S. Zhou, B. Choi, PICASSO: exploratory search of connected subgraph substructures in graph databases, Proceedings of the VLDB Endowment 10 (2017) 1861–1864. doi:10.14778/3137765.3137794.

[4] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, W. Han, Versatile equivalences: Speeding up subgraph query processing and subgraph matching, in: G. Li, Z. Li, S. Idreos, D. Srivastava (Eds.), SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, ACM, 2021, pp. 925–937. doi:10.1145/3448016.3457265.

[5] K. Lee, H. Jo, J. Ko, S. Lim, K. Shin, Ssumm: Sparse summarization of massive graphs, in: R. Gupta, Y. Liu, J. Tang, B. A. Prakash (Eds.), KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020, ACM, 2020, pp. 144–154. doi:10.1145/3394486.3403057.

[6] C. Wangmo, L. Wiese, Sumexplorer, 2023. URL: https://anonymous.4open.science/r/SuMExplorer-1137.

[7] AIDS, 2004. URL: https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data.

[8] R. Ayed, Aggregated search in Distributed Graph Databases. (Recherche d'information agrégative dans des bases de graphes distribuées), Ph.D. thesis, University of Lyon, France, 2019.

[9] T. Meinl, M. Wörlein, O. Urzova, I. Fischer, M. Philippsen, The parmol package for frequent subgraph mining, Electronic Communication of the European Association of Software Science and Technology 1 (2006). doi:10.14279/tuj.eceasst.1.85.