Efficient Subgraph Indexing for Biochemical Graphs*

Chimi Wangmo[a] and Lena Wiese[b]
Goethe University Frankfurt, Institute of Computer Science, Robert-Mayer-Str. 10, 60629 Frankfurt am Main, Germany

Keywords: Subgraph Indexing, Transaction Graph Database, Radix Tree, Path Compression.

Abstract: The dynamic nature of graph-structured data demands fast subgraph query processing to solve real-world problems such as identifying spammers in social networks, fraud detection in the financial system, and finding motifs in biological networks. The need for an efficient subgraph search has motivated the study for filtering the candidate graphs using the filter-then-verify framework with minimal indexing size. This paper presents an efficient in-memory index structure for indexing the paths in the transaction graph database. Our radix tree-based index structure addresses the issue of high memory consumption related to trie for representing biochemical datasets. Furthermore, we also contrast various containers used in the radix nodes. We demonstrate empirically the benefits of compressing the common prefixes in the path by achieving 20% reduction in the indexing size than a trie-based implementation.

1 INTRODUCTION

Graphs are a natural representation of datasets, widely adopted in various areas such as mining chemical structure information for drug discovery, prediction of traffic accidents, and recommendation of products on e-commerce websites. Over the decades, the vast availability of interconnected graph datasets and the growing popularity of graph databases has motivated the study of graph indexing to speed up query processing. Other motivation came from improving data structures to efficiently store graph data for subgraph processing (or subgraph search) and algorithms to solve subgraph isomorphism (or subgraph matching) for large-sized and numerous numbers of small graphs, respectively. Subgraph searching and subgraph isomorphism problems are some of the classic NP-hard problems in graphs. To overcome the challenges of slow construction time and large space consumption of subgraph indexes, our main contributions are listed as follows:

1. The earlier works focused mostly on solving the problem of subgraph query processing without concerning themselves with space-efficient indexing techniques, while the number of data graphs in the database increases. In contrast to solely solving subgraph queries, we design a compressed suffix tree indexing structure to store paths of the graphs.
2. We propose an index structure in which each node contains graph occurrence information. The graph information comprised of the data graph identifier as the key and the number of occurrences of the path in the data graph as a value. These keys are compressed using path compression optimization.
3. We propose a detailed comparative study of the different index structure used for representing the suffix nodes in the compact trie data structure.

The paper is organized as follows. The section 2 provides the background on the subgraph indexing. The section 3 briefly defines some terminologies associated with graphs. Section 4 describes the compressed suffix tree data structure for graph. The section 5 discusses the experimental setup and the analysis of the results. Finally, the section 6 provides the conclusion while, the section 7 discusses regarding the challenges and objective for future works.

2 RELATED WORK

In the subgraph query processing, the main goal is to find all the data graphs that contain the query graph q. Generally, subgraph pattern matching (Katsarou, 2018; Giugno et al., 2013) is a straightforward approach to finding all the subgraphs in a graph, that are

[a] https://orcid.org/0000-0002-8921-398X
[b] https://orcid.org/0000-0003-3515-9209
*Supported by a DAAD PhD scholarship.

Wangmo, C. and Wiese, L.
Efficient Subgraph Indexing for Biochemical Graphs.
DOI: 10.5220/0011350100003269
In *Proceedings of the 11th International Conference on Data Science, Technology and Applications (DATA 2022)*, pages 533-540
ISBN: 978-989-758-583-8; ISSN: 2184-285X

isomorphic to the query graph q. One approach for subgraph searching could be to exhaustively perform subgraph matching for all the graphs g_i in the graph database G and verify using subgraph isomorphism between q and g_i, where $g_i \in G$ (as defined below). Such a naive method is however computationally expensive.

The recent heuristics methods developed to solve the subgraph isomorphism (or subgraph matching) problem have shown significant performance improvement. Most existing algorithms for subgraph isomorphism are based on the idea of backtracking, where the query vertices are matched to vertices in the data graph incrementally (Sun et al., 2022a; Sun et al., 2022b; Kim et al., 2021; Min et al., 2021; Han et al., 2019). The general framework for the subgraph isomorphism consists of two tasks: filtering and matching. In the filtering phase, the number of vertices in the data graph mapped to the vertices in the query graph is reduced, thereby producing the potential candidate vertex sets. In the matching phase, the backtracking approach is performed by recursively extending each vertex in the candidate sets and then checking for the subgraph isomorphism of the query graph. Nevertheless, the subgraph isomorphism approaches are generally evaluated for small-sized query graphs and only a single data graph.

On the other hand, some approaches answer the subgraph query by typically utilizing the two phase filter-then-verify (FTV) framework to build the subgraph indices (Licheri et al., 2021; Luaces et al., 2021). The first phase, index construction phase, involves enumeration of the graph patterns (such as paths or trees or cycles, etc.) either through mining frequent patterns or exhaustive enumeration. This is then followed by query processing phase, which includes filtering out the data graphs in the graph database that does not contain query graph, thereby generating pruned candidate graphs. Finally, in the verification step, the subgraph matching is performed on the candidate graphs. Therefore, the pruning power of the indexes reduces the number of subgraph isomorphic verification to the less number of candidate graphs. However, the works on subgraph indexing are mainly targeted towards setting where there are many numbers of small data graphs. Further, the algorithms those uses mining-based approach to generate frequent patterns and then index them produces stable indexing structure but consumes longer construction time. While the methods that perform exhaustive enumeration of small patterns, require more memory space to store all the permutations of patterns in the data graphs. Our approach is developed to provide a compact suffix tree representation for subgraph indexing.

3 PRELIMINARIES AND PROBLEM DEFINITION

In this paper, we consider undirected, connected, vertex-labelled graphs in transaction graph database. The transaction graph database is a set of large number of small graphs (or connected components), called *data graphs*, $G = \{g_1, g_2, g_3, ..., g_n\}$. Each graph g_i is defined as a triplet $g = \{V_g, E_g, L_g\}$, composed of three elements where V_g is the set of vertices, E_g is the set of edges between vertices in the graph, and L_g is the set of labels associated with the vertices in the graph. More precisely, L_g is a mapping function that maps vertex to a label in σ, where σ is the distinct label list. Subgraphs correspond to a subset of nodes of the original graphs. A graph h is said to be induced subgraph of a data graph g if the vertices in h is the subset of g, $V_h \subseteq V_g$, and the corresponding edge set consists of all the edges in E_g that have both the endpoints in V_h. The label of the vertices in h is the same as the label of the corresponding vertex in g.

Definition 1 (Subgraph isomorphism (Kim et al., 2021))**.** *Given a query graph q and data graph g, an embedding of q in g is a mapping $M : V_q \rightarrow V_g$ such that: (1) M is injective (i.e. $M(u) \neq M(u')$ for every $u \neq u' \in V_q$, (2) $L_q(u) = L_g(M(u))$ for every $u \in V_q$ (3) $(M(u), M(u')) \in E_g$ for every $(u, u') \in E_q$. q is said to be isomorphic to g, denoted by $q \subseteq g$ if there exists an embedding of q in G.*

Our proposed indexing structure aims to speed up the search for subgraphs while maintaining small-sized indexes.

Definition 2 (Subgraph searching)**.** *For a given transaction graph database with n data graphs $G = \{g_1, g_2, g_3, ..., g_n\}$ and a query graph q, the subgraph search is to find all the graphs g_i in the database that contain q.*

4 OUR COMPRESSED TRIE

We now derive our construction of a suffix tree that supports the two above-mentioned inner node structures.

4.1 Compressed Suffix Tree Construction

In this section, we discuss the representation and the process of building the compressed suffix tree to index the paths in the graph for obtaining the candidate graphs. Inspired by the idea of text compression, our method implements a radix tree like index structure

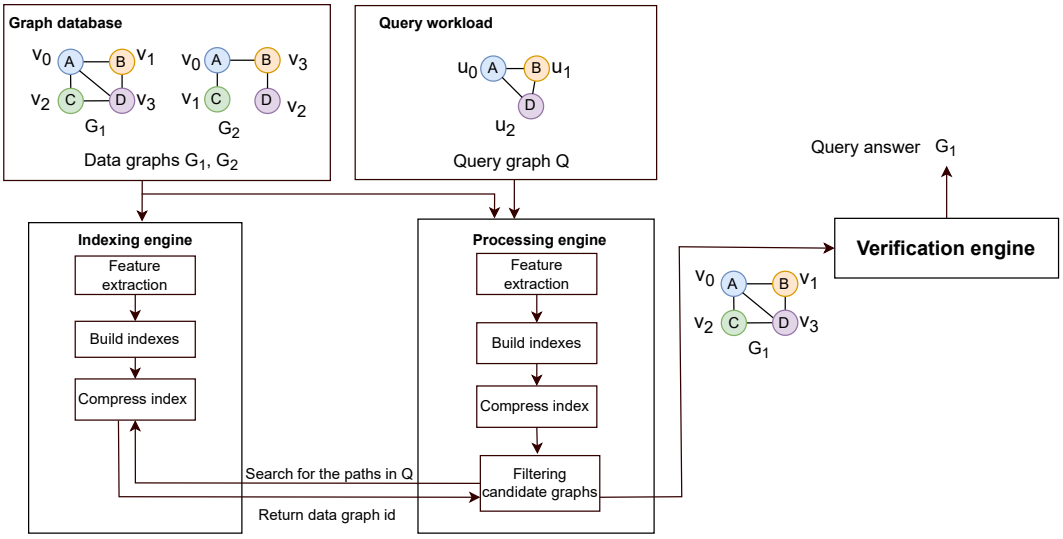

Figure 1: System architecture.

for multiple paths in the data graphs. The repetition of labels in the graph path is similar to the text repetition. The general observation is that the existing trie-based index structure for labelled paths consists of the repeating suffix nodes in multiple paths and suffix node's graph identifier and occurrences information. As a result, our compressed suffix tree provides two-fold compression. The first one is in terms of compressing the suffix tree by merging the common suffix nodes into one. The second one offers a compact representation of the graph identifier and occurrences information in the suffix nodes. Our compressed suffix tree will help to compress the Trie in terms of a common prefix, common labels in the single path, and the single labelled path to one node.

4.2 Index Building Phase

The construction of the radix tree involves the insertion of the extracted paths during depth-first search (DFS) traversals in an unsorted and incremental manner. We execute a DFS up to a certain maximum length l_d starting from each vertex $V_g = (v_1, v_2, ..., v_k)$ in each data graph g_i. Thus, all the subsequent prefix and suffix paths are generated and indexed. During each path traversal in the graph, the corresponding radix node is built, containing vertex labels as a partial key and a corresponding list of graph identifiers and occurrences as a value. Formally, path indexing can be defined as follows:

Definition 3 (Path indexing)**.** *Path indexing is the process of building indexes of the labelled paths being extracted by traversing each vertex* $\{v_1, v_2, ..., v_k\}$ *in each data graph* $\{g_1, g_2, ..., g_n\}$ *using depth-first search up to maximum length* l_d.

4.2.1 Structure of Compressed Suffix Tree

Generally, the issue with Trie-based data structures is the space overhead due to the large number of pointers associated with each inner node which are often found to be empty. On the other hand, the radix tree utilized fewer nodes as opposed to trie by providing a mechanism to merge common suffix and prefix partial keys. The radix tree structure consists of a rooted tree starting with an empty node.

Our method includes storing all the possible suffixes generated during depth-first search traversal of each data graph in the radix tree. Normally, the suffixes generated are often redundant, and the labels are already indexed, hence only the occurrences information and graph identifier are updated. Unlike the GRAPES (Giugno et al., 2013) method, our method implements path compression and lazy expansion.

In our implementation, each DFS generated path inserts a new node and upon encountering common prefix paths, only the graph identifier and occurrences information are updated. Further, if a new node is a single child of a node, then the child is merged with the parent node.

Additionally, the choice of data structure used for the inner node should maintain the trade-off between faster retrieval and space optimization. The typical representations are based on arrays, linked lists, binary search tree, and hashmaps. In contrast to GRAPES which represents inner nodes using a linked list only, our implementation includes for both hashmap-based radix tree and linkedlist-based radix tree in order to enable a comparison in a uniform environment. We now specify the basic data structures. Hashmap-based radix tree is the simplest form

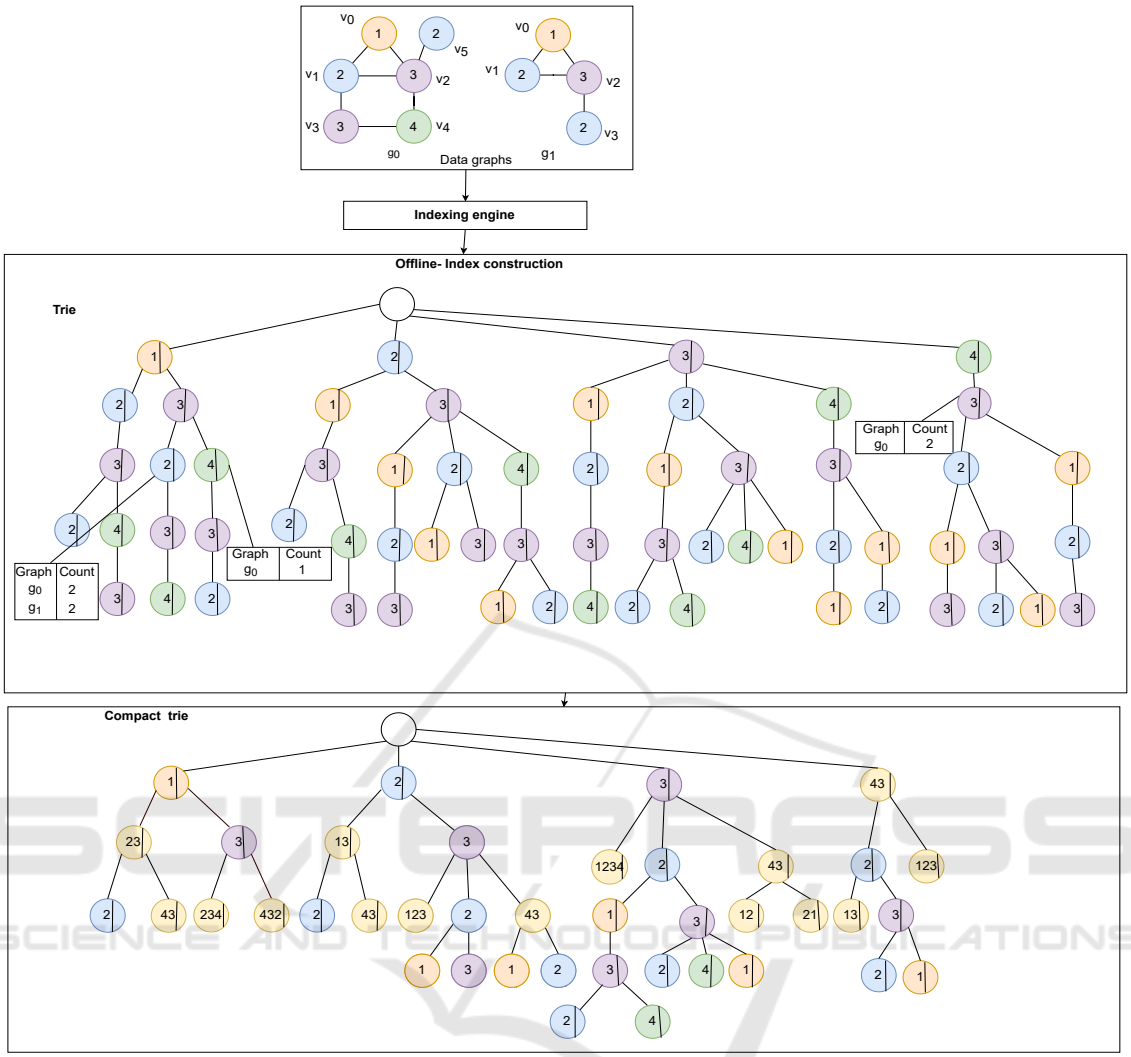

Figure 2: Index construction.

of node providing linear search time as well as compact storage space. However, it does not support predecessor-successor relationship.

Definition 4 (Indexed node of the Hashmap-based radix tree)**.** *An* indexed node of the hashmap-based radix tree $\gamma_i(HRTree)$ *is a quadruplet* $\langle \lambda, C, G_I, 1 \rangle$*, which is node* γ_i *of a hashmap-based radix tree HRTree (see Definition 5). The element* λ *is a node identifier, which consists of ordered sequence of integers* $\lambda = \{\lambda_1, \lambda_2, ..., \lambda_j\}$*, where the length of the node identifier belonging to an indexed node in HRTree is length*$(\gamma_i(HRTree))$*. The term "node identifier" is used in conjunction to an indexed node of the hashmap-based radix tree, which should not be confused with the term "label" for a vertex in a graph. The element* C *is the set of children keys* $C = \{c_1, c_2, ..., c_m\}$*, where* $c_k = \{\lambda_1 \in \lambda(child_k(\gamma_i(HRTree))) \mid 1 \leq k \leq m\}$*.* $child_k(\gamma_i(HRTree))$ *is the* k^{th} *child of the current indexed node* $\gamma_i(HRTree)$ *inserted at the* k^{th} *position. Each child key is mapped to the children of the indexed node* $M : c_k \rightarrow child_k(\gamma_i(HRTree))$*, where* $1 \leq k \leq m$*. The element* G_I *is the graph information consisting of graph identifiers and corresponding occurrences count. The element* 1 *has a boolean value associated with it; with* 1 *means that the indexed node is a leaf whereas* 0 *means it is not.*

Definition 5 (Hashmap-based radix tree)**.** *The* hashmap-based radix tree *HRTree is a triplet* $\langle \mathscr{R}, \mathscr{I}, \mathscr{L} \rangle$*.*

1. *HRTree is rooted with an indexed node called as a root node, denoted by* $\mathscr{R}$*. Formally,* $\mathscr{R}$ *is defined as,*

$$\mathscr{R} = \{\gamma_i(HRTree) \mid \lambda_1(\mathscr{R}) = -1\} \quad (1)$$

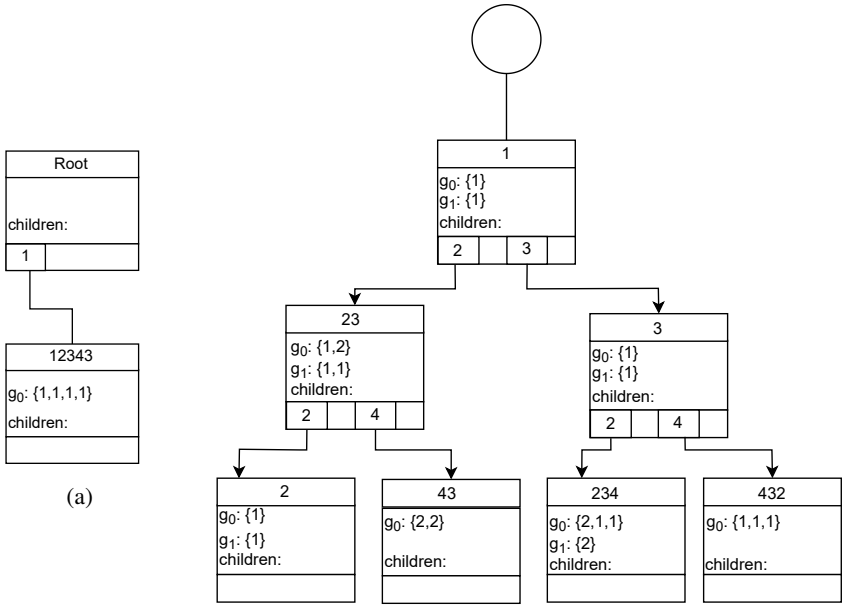

Figure 3: (a) Initial insertion of the labelled path (b) Indexing of the labelled paths originated from vertex v_0.

2. *The element $\mathscr{I}$ is an indexed node referred to as an inner node. Formally, $\mathscr{I}$ is defined as,*

$$\mathscr{I} = \{\gamma_i(HRTree) \mid \mathscr{I} = \neg\mathscr{R} \wedge \neg\mathscr{L}\} \quad (2)$$

3. *The element $\mathscr{L}$ is an indexed node referred to as a leaf node containing the value of the boolean leaf as* 1*. Formally, $\mathscr{L}$ is defined as,*

$$\mathscr{L} = \{\gamma_i(HRTree) \mid 1 = 1\} \quad (3)$$

LinkedList-based radix tree is a dynamic data structure composed of node label, run-time created dynamic children list, and next sibling pointer. We implemented two options, namely Hashmap-based radix tree *HRTree*, and Linkedlist-based radix tree *LRTree* with the path compression and lazy expansion optimization. The data structures are implemented in C++, and the current implementation supports the vector of integers for the labelled node. The maximum size of each node in the radix tree is dependent on the type of data structure utilized for the node representation and whether the node stores common prefixes.

4.2.2 HashMap-based Radix Tree

Definition 6 (Labelled path)**.** *A* labelled path *LP is an ordered sequence of integers $LP = (\sigma_1, \sigma_2, ..., \sigma_k + 1)$, where each vertex identifier $v_i \in V_g$ is mapped to exactly one label $\sigma_i \in L \; \forall \; 1 \leq i \leq k+1$. Formally, LP is defined as,*

$$LP = \{\sigma_1, \sigma_2, ..., \sigma_k + 1 \mid M : v_i \rightarrow \sigma_i, \sigma_i \in L, \forall 1 \leq i \leq k+1\} \quad (4)$$

where, k is the maximum length of the labelled path.

Definition 7 (Indexed path)**.** *An* indexed path *IP is an ordered sequence of integers in node identifiers λ associated with each indexed node γ_i from a root node $\mathscr{R}$ to a leaf node $\mathscr{L}$. Formally, IP is defined as,*

$$IP = \{\lambda(\gamma_1), \lambda(\gamma_2), ..., \lambda(\gamma_n) | 1 \leq n \leq N\} \quad (5)$$

p refers to as the length of the indexed path, which is the total number of integers in the node identifier λ for each indexed nodes γ_i in the indexed path IP. Formally, p is defined as,

$$p = \sum_{i=1}^{n} length(\gamma_i) \quad (6)$$

Example 1. *In the Figure 2, the depth-first search traversed id-path $\{v_0, v_1, v_2, v_4, v_3\}$ of g_0 is represented as a labelled path $LP = \{1,2,3,4,3\}$ obtained from graph translate to one of the indexed path IP in the index structure. Here, each value in $\{1,2,3,4,3\}$ is the label mapped to the corresponding vertex identifier $\{v_0, v_1, v_2, v_4, v_3\}$.*

Example 2. *Assuming that the aforementioned labelled path LP has not been indexed, the Figure 3a shows the labelled path $LP = \{1,2,3,4,3\}$ will be inserted as a child to the root node $child_k(\mathscr{R})$ with* 1 *as a key c_k. The value will be the new node $\gamma_i(HRTree)$ containing $\{1,2,3,4,3\}$ as a node identifier λ.*

Example 3. *Given the labelled path LP in Figure 3a is traversed from g_0. The indexed node with the node identifier $\lambda = \{1,2,3,4,3\}$ will have graph occurrence information G_I. G_I is represented using unordered map which contains g_0 as the key, denoting*

the first data graph in the transaction database. Further, g_0 is mapped with $\{1,1,1,1\}$ as the occurrences count associated with each labelled sub path. Here, the first value in $\{1,1,1,1\}$ implies that occurrences count associated with 1 sub-path, the second value 1 implies the occurrences count associated with $1,2$ sub-path, the third value implies 1 the occurrences count associated with $1,2,3$ sub-path.

Definition 8 (Key containment)**.** *Given a new labelled path LP_{new} and a root node $\mathscr{R}$, the* key containment *is true, if the first integer of the labelled path is contained in the children key of the root node. Formally, $Key(LP_{new}) \in \mathscr{R}$ is defined as,*

$$Key(LP_{new}) \in \mathscr{R} = \begin{cases} 1, & if\ \sigma_1(LP_{new}) \in C(\mathscr{R}) \\ 0, & otherwise. \end{cases} \quad (7)$$

Definition 9 (Full path containment)**.** *Given a new labelled path LP_{new} and an indexed path IP in the hashmap-based radix tree HRTree, the* full path containment *$Full(LP_{new}) \in HRTree$ is true, if there exist one-to-one ordered, mapping from each integer in LPnew to each integer in IP. Formally, $Full(LP_{new}) \in HRTree$ is defined as,*

$$Full(LP_{new}) \in HRTree = 1,\ M : \sigma_i(LP_{new}) \rightarrow \beta_i(IP) \mid 1 \leq i \leq j\} \quad (8)$$

Definition 10 (Longest common prefix)**.** *Given a newly traversed labelled path LP_{new} and an indexed node $\lambda(\gamma_i)$, the* longest common prefix *$LCP(LP_{new}, \gamma_i)$ is a sub-path from 0^{th} to min length of LP_{new} that maps to $\lambda(\gamma_i)$. Formally, $LCP(LP_{new}, \gamma_i)$ is defined as,*

$$LCP(LP_{new}, \gamma_i) = M : \sigma_i(LP_{new}) \rightarrow \lambda_i(\gamma_i), \quad \forall\ 0 \leq i \leq min. \quad (9)$$

Definition 11 (Prefix of the labelled path)**.** *Given a newly traversed labelled path LP_{new} and a specified position i, the* prefix of the labelled path *$Pre(LP_{new}, i)$ is a sub-path that starts from 0^{th} to the i^{th} position of the the labelled path. Formally, the $Pre(LP_{new}, i)$ is defined as,*

$$Pre(LP_{new}, i) = \{LP_{new}[0,..,i] | i = 1, if\ i = NULL\} \quad (10)$$

Definition 12 (Suffix of the labelled path)**.** *Given a newly traversed labelled path LP_{new} and indexed node γ_i, the* suffix of the labelled path *$Suf(LP_{new}, \gamma_i)$ is a sub-path that starts from j^{th} to k^{th} position of the labelled path LP_{new}. Formally, the $Suf(LP_{new}, \gamma_i)$ is defined as,*

$$Suf(LP_{new}, \gamma_i) = \{LP_{new}[j,.,k] \mid j = |LCP(LP_{new}, \gamma_i)|\} \quad (11)$$

The insertion process involves performing hop constrained depth-first search traversal and the addition of the *LP* as a node γ_i to the hashmap-based radix tree *HRTree*. Typically, there are three scenarios concerning the insertion of children nodes:

1. If the newly traversed labelled path LP_{new} and root node $\mathscr{R}(HRTree)$ is not able to satisfy the *key containment*, create new node with LP_{new} as the node identifier λ. The child of the root node is represented using an unordered map container. The child will contain the first character $\lambda_1 \in \lambda$ as key and the new child node as the value. Moreover, we will update the G_I by incrementing the occurrences count of the labelled path *LP* in the g_i.
2. If the newly traversed path LP_{new} is fully contained in the hashmap-based radix tree $Full(LP_{new}) \in HRTree$, then only the $g_i \in G_I$ associated with indexed node $\gamma_i(LRTree)$ is incremented. However, if the length of the newly traversed path $k(LP_{new})$ is longer than the lenght of the indexed path $p(IP)$. In such scenario, we would have to increment the occurrences count associated with $g_i \in G_I$ for the longest common prefix between the $\lambda_1, \lambda_2, ..., \lambda_j$. In addition, we would perform recursive insertion with the suffix of the newly traversed path $Suf(LP_{new}, \gamma_i)$ as a new child's node identifier λ.
3. If a prefix of the newly traversed path is in the indexed node of the hashmap-based radix tree $Pre(LP_{new}, i) \in \gamma_i(HRTree)$, but only partially shares a certain common prefix $LCP(LP_{new}, \gamma_i(HRTree)) = \{\sigma_1, \sigma_2, ..., \sigma_j\}$ to the label λ of an existing indexed node $\gamma_i(HRTree)$. This causes an splitting of the indexed node into two nodes: new node γ_{new} and the existing indexed node $\gamma_i(HRTree)$. The new node γ_{new} contains the common prefix $\lambda(\gamma_{new}) = LCP(LP_{new}, \gamma_i(HRTree))$. Further, the existing indexed node $\gamma_i(HRTree)$ is updated as the child to the new node $\lambda(\gamma_i(HRTree))child_{k+1}(\gamma_{new}) = \gamma_i(HRTree)$. In addition, the node identifier of an existing indexed node is revised to its suffix $Suf(LP_{new}, \gamma_i(HRTree))$. Similarly, the graph information G_I is updated by incrementing the occurrences count for the common prefix label of the newly indexed node γ_{new} in g_i.

The lookup operation is similar to how insertion occurs.

4.2.3 Linkedlist-based Radix Tree

In case of a linkedlist-based radix tree *LRTree*, each indexed node $\gamma_i(LRTree)$ has a node identifier λ rep-

resented using the vector of integers. The node identifier of the indexed node $\lambda(\gamma_i(LRTree))$ corresponds to the traversed labelled path LP_{new}. Further, the pointer to a child node is stored in the indexed node. In addition, pointer to a next sibling node, if any, is as well stored in the indexed node. Most importantly, graph occurrences information G_I associated with the indexed node $\gamma_i(LRTree)$ is inserted. The insertion process is similar to the implementation for the hashmap-based radix tree $HRTree$. Initially, the linkedList-based radix tree $LRTree$ contains only the root node $\mathscr{R}$ with node identifier as -1. The children C and sibling S pointers of the root node will contain null values.

5 PERFORMANCE EVALUATION

This section shows the experimental results to evaluate the effectiveness of our indexing structure on datasets of increasing size. The experiment is run on a machine with Seven Intel Core 1185G7@ 3.00 GHz 1.80 GHz CPUs and 32 GB of RAM running Windows operating system.

5.1 Datasets

The input dataset is a text file that contains V_g and E_g related to each $g_i \in G$. The first line starts with the character 't', which denotes a transaction as a data graph. The character 't' is followed by integer i, which represents the graph identifier g_i. The following lines which begin with the character 'v' indicate the vertices. The letter 'v' is followed by a vertex identifier $v_i \in V_g$ and its corresponding label $l_i \in L$. Additionally, the next lines which are preceded by 'e' indicate edges $\in E_g$, which contain the source vertex identifier $v_i \in V_g$ and the destination vertex identifier $v_j \in V_g$. We ran our experiments on the two datasets obtained from (Sun and Luo, 2019). The first dataset, AIDS, comprises of 40,000 small graphs with 62 unique labels. Each graph contains on average 45 vertices and 46.95 edges. The second dataset, PDBS, has 600 data graphs that represent DNA, RNA, and proteins. Each data graph has an average of 2939 vertices with an average degree of 2.06 per vertex, and 3,064 edges. Besides that, the graph consists of 10 unique labels.

5.2 Performance Measurement

We have measured the index construction time as well as the index size for three cases: an uncompressed index as well as our linkedlist- and hashmap-based suffix trees.

The barcharts in Figures 4 to 5 visualize the indexing size and timing measurements. In the figures, the Hashmap-based Radix tree and LinkedList-based Radix tree are denoted as $HRTree$ and $LRTree$ respectively. Moreover, for each dataset, we have generated its subgroup as small, medium, and large, which contains 20%, 50%, and 100% of the original data graphs. In addition, each experiment is run four times and the average result has been reported for the indexing size and construction time.

Our experiments in Figure 4 and 5 show that the choice of the data structure to represent the node impacts the performance. Further, the optimization using path compression can overcome the drawbacks of trie indexing structure, thereby reducing the memory consumption.

In the AIDS dataset, the finding indicates the benefits of path compression. For the large subgroup of the AIDS dataset, the hashmap-based radix tree occupies approximately 20% and 12% less space in hard disk than the trie and linkedlist-based radix tree respectively. In particular, the construction time for the linkedlist-based radix tree is the longest, followed by an uncompressed trie.

In contrast to the noticeable impact of compression on various AIDS data subgroups, the index structure size has levelled out for the PDBS data set. This can be because for the trie and compressed trie, as soon as all the enumeration of paths up to a certain maximum length has been indexed only the graph occurrences information is updated. A general trend is that both the index size and construction time increase as we add more data graphs.

6 CONCLUSIONS

Graph indexing is essential to enable efficient query processing for graph database. Hence, it is well studied and adopted in various fields ranging from bioinformatics to social network analysis. In this work we presented a compact radix tree for indexing biochemical datasets. We have demonstrated through benchmarking the benefits of path compression, over the regular trie data structure to reduce the index size for storing biochemical datasets.

7 FUTURE WORK

In future work, we consider the integration of query processing to enable run time indexing of a query

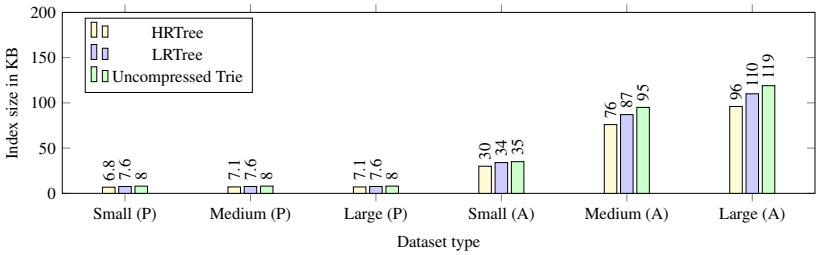

Figure 4: Average size of indexes (A: AIDS, P: PDBS).

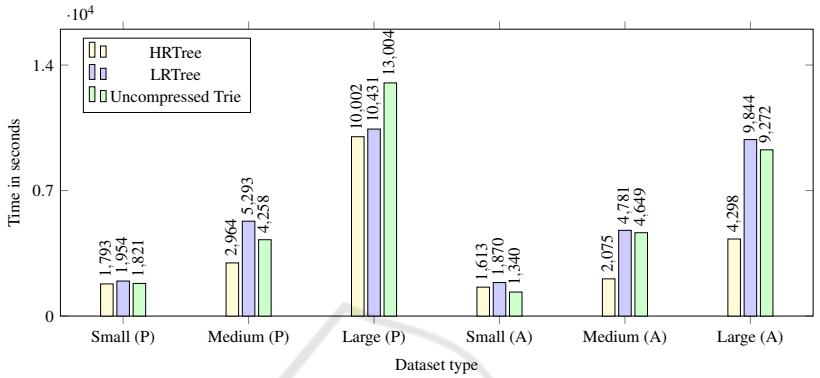

Figure 5: Average construction time of indexes (A: AIDS, P: PDBS).

graph and generation of candidate graphs with our index structure. On the technical side, it would be interesting to implement the maintenance algorithm to support an incremental update operation.

REFERENCES

Giugno, R., Bonnici, V., Bombieri, N., Pulvirenti, A., Ferro, A., and Shasha, D. (2013). Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one*, 8(10):e76911.

Han, M., Kim, H., Gu, G., Park, K., and Han, W. (2019). Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In Boncz, P. A., Manegold, S., Ailamaki, A., Deshpande, A., and Kraska, T., editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1429–1446, Amsterdam. ACM.

Katsarou, F. (2018). *Improving the performance and scalability of pattern subgraph queries*. PhD thesis, University of Glasgow, UK.

Kim, H., Choi, Y., Park, K., Lin, X., Hong, S., and Han, W. (2021). Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In Li, G., Li, Z., Idreos, S., and Srivastava, D., editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 925–937, China. ACM.

Licheri, N., Bonnici, V., Beccuti, M., and Giugno, R. (2021). GRAPES-DD: exploiting decision diagrams for index-driven search in biological graph databases. *BMC Bioinform.*, 22(1):209.

Luaces, D., Viqueira, J. R., Cotos, J. M., and Flores, J. C. (2021). Efficient access methods for very large distributed graph databases. *Information Sciences*, 573:65–81.

Min, S., Park, S. G., Park, K., Giammarresi, D., Italiano, G. F., and Han, W. (2021). Symmetric continuous subgraph matching with bidirectional dynamic programming. *Proc. VLDB Endow.*, 14(8):1298–1310.

Sun, S. and Luo, Q. (2019). Scaling up subgraph query processing with efficient subgraph matching. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 220–231, China. IEEE.

Sun, X., Sun, S., Luo, Q., and He, B. (2022a). An in-depth study of continuous subgraph matching (complete version). *CoRR*, abs/2203.06913.

Sun, Y., Li, G., Du, J., Ning, B., and Chen, H. (2022b). A subgraph matching algorithm based on subgraph index for knowledge graph. *Frontiers Comput. Sci.*, 16(3):163606.