(IJCSIS) International Journal of Computer Science and Information Security,
Vol. XIII, No. XII, December 2015

GCTrie for Efficient Querying in Cheminformatics

Yu Wai Hlaing
Ph.D candidate
University of Computer Studies
Yangon, Myanmar
yuwaihlaing.1987@gmail.com

Kyaw May Oo
Faculty of Computing
University of Information Technology
Yangon, Myanmar
kmayoo19@gmail.com

***Abstract*— The field of graph indexing and query processing has received a lot of attention due to the constantly increasing usage of graph data structures for representing data in different application domains. To support efficient querying technique is a key issue in all graph based application. In this paper, we propose an index trie structure (GCTrie) that is constructed with our proposed graph representative structure called graph code. Our proposed GCtrie can support all types of graph query. In this paper, we focus on index construction, subgraph query and supergraph query processing. The experimental results and comparisons offer a positive response to the proposed approach.**

Keywords-graph indexing and querying; graph representative structure; index; subgraph query; supergraph query

I. Introduction

Many scientific and commercial applications urge for patterns that are more complex and complicated to process than frequent item sets and sequential patterns. Such sophisticated patterns range from sets and sequences to trees, lattices and graphs. As one of the most general form of data representation, graphs easily represent entities, their attributes and their relationships to other entities. The significant of using graphs to represent complex datasets has been recognized in different disciplines such as chemical domain [6], computer vision [7], and image and object retrieval [8]. Various conferences over the past few years on mining graphs have motivated researchers to focus on the importance of mining graph data. Different applications result in different kinds of graphs, and the corresponding challenges are also quite different. A graph describes relationships over a set of entities. With nodes and edges labels, a graph can depict the attributes of both the entity set and the relation. For example, chemical data graphs are relatively small but the labels on different nodes (which are drawn from a limited set of elements) may be repeated many times in a single molecule (graph).

Storing the graphs into large datasets is a challenging task as it deals with efficient space and time management. Over the years, a number of different representative structures have been developed to represent graphs more and more efficiently and uniquely. Developing such structures is particularly challenging in terms of storage space and generation time. Among many representative structures adjacency list [9] and adjacency matrix [10] are the most common. We have already proposed a new graph representative structure called graph code [1]. Graph code is a new way of representing graphs to support all kinds of graph queries without verifying between graph structures. A good graph indexing and querying approach should have compact indexing structures and has a good power of pruning the false graphs in the dataset. The strategy of graph indexing is to move high costly online query processing to off-line index construction phase [2]. Chemical graphs in datasets are undirected labelled graphs. So, graph code is developed to process undirected labelled graphs. Graph code can retain the structural information of original graph such as which two edges are connected on which vertex.

To effectively understand and utilize any collection of graphs, an approach that efficiently supports elementary querying mechanism is crucially required. Given a query graph, the task of retrieving related graphs as a result of the query from a large graph dataset is a key issue in all graph based applications. This has raised a crucial need for efficient graph indexing and querying approaches. A primary challenge in computing the answers of graph queries is that pair-wise comparisons of graphs are usually really hard problems. It is apparent that the success of any graph based application is directly dependent on the efficiency of the graph indexing and query processing mechanisms. Recently, there are many techniques that have been proposed to tackle these problems.

In principle, queries in graph datasets can be broadly classified into the four categories: graph isomorphism query, subgraph query, supergraph query, and similarity query. Most of the existing graph indexing and querying approaches proposed to deal with only one type of the query problem. Our proposed approach allows the chemical compound dataset to be queried chemical structures in terms of XML file format. Using proposed approach, all types of graph queries can be processed. After entering a chemical structure as a query, user can process their desired query types. In this paper, we describe our proposed graph code structure, and GCTrie, and also perform subgraph query and supergraph query processing by probing GCTrie. We also perform experimental analysis on index construction and on these queries using proposed approach and other existing approaches.

II. Preliminaries

For simplicity, we present the key concepts, notations, and terminology used in our proposed approach which includes labeled undirected graph, graph automorphism, subgraph query, supergraph query, and graph code.

As a general data structure, labeled graphs is used to model complicated structures and schemaless data. In labeled graph, vertex and edge represent entity and relationship, respectively.

The attributes associated with entities and relationships are called labels. XML is a kind of directed labeled graph. The chemical compound shown in Fig. 1 is labeled undirected graph.

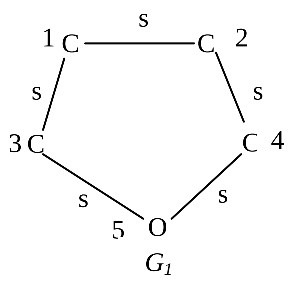

Figure 1. A Labeled Undirected Graph

Definition 1. Labeled Undirected Graph

A labeled undirected graph G is defined as 5-tuple, (V, E, L_V, L_E, l) where V is the non-empty finite vertex set called vertices, and E is the unordered pairs of vertices called edges. L_V and L_E are the set of labels of vertices and edges and l is a labelling function assigning a label to a vertex l: V to L_V and an edge l: E to L_E.

Definition 2. Graph Isomorphism

Let $G = (V, E, L_V, L_E, l)$ and $G' = (V', E', L'_V, L'_E, l')$ be two graphs. An automorphism between two graphs G and G' is an isomorphism mapping where $G = G'$. An isomorphism mapping is a mapping of the vertices of G to vertices of G' that preserve the edge structure of the graphs. That is, it is a graph isomorphism from a graph G to itself.

Definition 3. Subgraph Query

This category searches for a specific pattern in the graph dataset. The pattern can be a small graph. Therefore, given a graph dataset $D = \{G_1, G_2,\ldots, G_i\}$ and a subgraph query q,the answer set $A = \{G_i|q \subseteq G_i, G_i \in D\}$.

Definition 4. Supergraph Query

Given a graph dataset $D = \{G_1, G_2,\ldots, G_i\}$ and a supergraph query q, if a query q is a supergraph of a dataset graph, the answer set $A = \{ G_i| G_i \subseteq q, G_i \in D \}$.

Definition 5. Graph Code

For a graph G_i, the code of G_i, denoted by $c(G_i)$ is the list of the form $e_{id}[(v), e_{id\_adj}]\ldots$ depending on adjacent edges. e_{id} is the edge id, v is vertex label on which two edges are connected, e_{id_adj} is adjacent edge id for this edge.

III. RELATED WORKS

Graphs are used to represent many real life applications. Graphs can be used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. Many graph datasets (e.g., chemical compounds) have more than one vertex with the same label. Same graph is stored more than once in the graph datasets leading to adverse results of mining. To ensure the consistency of graph datasets, required a mechanism to check whether two graphs are automorphic or not. So, detection and elimination of automorphic graphs is required. In proposed approach, a graph is represented via its graph code generated by using adjacent edge information and edge dictionary. Instead of expensive graph automorphism test, automorphic graphs can be detected by matching codes of two graphs [3].

GraphGrep [4] was proposed that is a path-based technique to index graph datasets. It has three basic components: building the index to represent graphs as sets of paths, filtering dataset based on query and computing exact matching. GraphGrep enumerates paths up to a threshold length (l_p) from each graph. An index table is constructed and each entry in the table is the number of occurrences of the path in the graph. Filtering phase generates a set of candidate graphs for which the count of each path is at least that of the query. Verification phase verifies each candidate graph by subgraph matching. However, the graph dataset contains huge amount of paths and can have an effect on the performance of the index.

OrientDB [5] is an open source NoSQL database management system written in java. It is a multi-model database, supporting graph, document, key/value, and object models, but the relationships are managed as in graph databases with direct connections between records. It supports schema-less, schema-full and schema-mixed modes. It has a strong security profiling system based on users and roles and supports querying with SQL extended for graph traversal.

IV. PROPOSED APPROACH

In our proposed approach, there are three main phases: code generation phase, subgraph query and supergraph query processing phase, and graph isomorphism query and similarity query processing phase. There are three sub-steps in code generation phase. These are preprocessing, code generation and automorphism checking, and index construction. In subgraph query and supergraph query processing phase, there are four sub-steps: preprocessing, code generation, subgraph querying and supergraph querying. In graph isomorphism query and similarity query processing phase, there are also four sub-steps, preprocessing, code generation, and graph isomorphism querying and similarity querying. In this paper, we focus on index construction step, and subgraph query and supergraph query processing phase.

A. Preprocessing, CodeGeneration and Automorphism Checking

In preprocessing, the graph information such as vertex information, edge information, and adjacent edge information are generated by parsing input xml files with xml parser. The edge information of the graph is defined as (V_{id},L,V_{id}) where V_{id} is the vertex id, L is the edge label. Then adjacent edge information is generated. Fig. 2 shows graph information for graph G_1 in Fig. 1.

Vertex id :	1	2	3	4	5
Vertex Info :	C	C	C	C	O

(a)

V_{id},L,V_i:	<1,s,2>	<1,s,3>	<2,s,4>	<3,s,5>	<4,s,5>
Edge Info:	<C,s,C>	<C,s,C>	<C,s,C>	<C,s,O>	<C,s,O>

(b)

Edge:	**Adjacent Edges:**
<1,s,2>	<1,s,3>, <2,s,4>
<1,s,3>	<1,s,2>, <3,s,5>
<2,s,4>	<1,s,2>, <4,s,5>
<3,s,5>	<1,s,3>, <4,s,5>
<4,s,5>	<2,s,4>, <3,s,5>

(c)

Figure 2. Graph Information of G_1 (a) Vertex Information (b) Edge Information (c) Adjacent Edge Information

For each edge from the graph's edge information, check the edge dictionary to determine whether the edge is already existed in edge dictionary or not. If not, insert new edge into edge dictionary. Then, the edge ids are associated with their corresponding edges in graph's edge information. Edge dictionary is shown in Fig. 3.

Id	Edge
1	<C,s,C>
2	<C,s,O>
…	…

Figure 3. Edge Dictionary

A graph is represented holistically into a graph code that preserves the structural information of the graph. Every edge in the graph is assigned with global unique identifier already defined in the edge dictionary. Instead of using the edge itself, using the edge id of the edge dictionary can have advantages in three ways:

- Firstly, using the edge id in the code saves the amount of storage space.
- Secondly, using the same id for the duplicated edge is effective when constructing the graph code.
- Thirdly, using the edge id in the code reduces the time for finding automorphic or isomorphic graphs.

Most of the chemical graphs have a lot of common edges. So, edge dictionary uses little memory space. Edge dictionary and adjacent edge information are used to generate graph code. Graph code for graph G_1 is as follows:

$c(G_1)$=1[c,1],1[c,1],1[c,1],1[c,2],1[c,1],1[c,2],2[c,1],2[o,2],2[c,1],2[o,2]

After computing the graph code of G_k, compares it with each graph code G_i in code store (*CS*), $1 <= i < k$, to check graph automorphism. If the graph code of G_k has the same code as that of G_i, concludes that the two graphs are automorphic and append id of G_k to corresponding graph code of G_i. Otherwise, add the graph code of G_k to *CS* assuming as G_k is a new graph.

B. Index Construction

After generating graph codes for all dataset graphs and checking automorphism, the next step is to construct GCTrie for efficient querying. Instead of using path or subgraph decomposition to support subgraph query type which has result in strctural information lost and exhaustive enumeration time problems, we propose an index trie structure called GCTrie for supporting all types of graph query. We put the graph codes of all dataset graphs in GCTrie. A GCTrie is a trie where each node except the root node is a string array that represents an edge id or an vertex label on which two edges are connected. There are five levels in the GCTrie. The second and fourth level is for edge ids. The third level is for vertex labels and the last is for leaves which are implemented by hashmaps of graph ids and their frequencies. Procedure for index construction is shown as follows. We represent one edge's adjacent code, e.g; 1[o,2] as feature *f*.

```
Procedure. IndexConstruction(CS)
For each $c(G_i) \in CS$
    For each feature $f \in c(G_i)$
        Put $f$ in GCTrie
return GCTrie
```

In index GCTrie construction, for the graph code of G_1 from *CS*, $c(G_1)$: 1[c,1],1[c,1],1[c,1],1[c,2],1[c,1],1[c,2],2[c,1],2[o,2],2[c,1],2[o,2], there are four occurrences of features 1[c,1]. There are two occurrences of features 1[c,2], 2[c,1] and 2[o,2]. So, the GCTrie after putting $c(G_1)$ is as shown in Fig. 4.

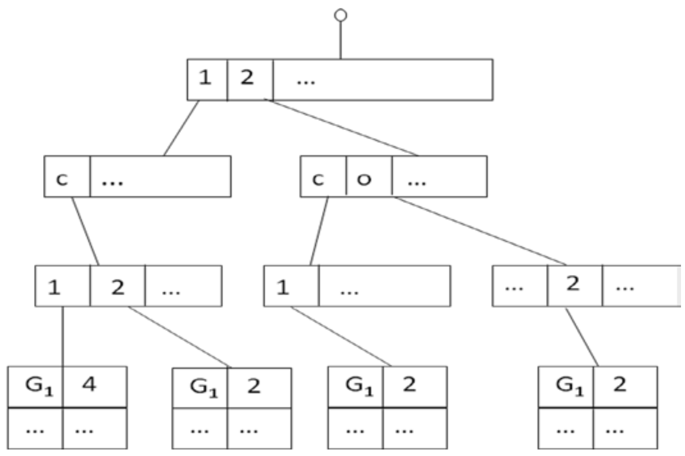

Figure 4. GCTrie After Putting $c(G_1)$

Then we put all dataset graph codes from *CS* into this GCTrie. When the query graph enters into the system, vertex information, edge information and adjacent edge information of query graph is generated in preprocessing step. Then the query graph code is generated in code generation phase using system's edge dictionary and query adjacent edge information. Each feature from query graph code is probed in GCTrie.

C. Subgraph Querying

In the core of many graph-related applications lies a common and critical problem of how to efficiently process subgraph query. In some cases, the success of an application directly relies on the efficiency of the query processing system. The classical graph query problem can be described as follows: given a graph dataset $D = \{G_1,G_2,\ldots,G_n\}$ and a graph query q, finds all the graphs in which q is a subgraph. If all of the query features are matched with features of data graph codes in the GCTrie but the size of dataset graphs are larger than query graph size, then the dataset graphs are returned as answer set. The following algorithm 1 describes the step-by-step process for subgraph query.

Algorithm 1 SubgraphQuery

```
Input : GCTrie, and Query $q$
Output : Answer set $D_q$.
1.  Generate graph code for query $q$.
2.  Let $D_q = D$
3.  For each feature $qf \in c(q)$
4.      Probe $qf$ in GCTrie.
5.      If $qf \in$ GCTrie
6.          Intersect $D_q$ and $D_{qf}$.
7.  For each $G_i \in D_q$
8.      If size($G_i$) < size($q$)
9.          Remove $G_i$ from $D_q$.
10. Return $D_q$;
```

Assume that we have generated the graph code of the query graph. We establish a necessary condition that forms the basis for processing subgraph query. Thus we state the following theorem.

Theorem 1 Given a query graph q, if q is a subgraph of a dataset graph G, then $c(q) \subseteq c(G)$.

Proof. By definition, if q is a subgraph of G, then every feature of q appears in G. Therefore, if parametric quantities of $c(q)$ are contained in $c(G)$, then $c(q) \subseteq c(G)$.

The intuition is as follows. If a query q is a subgraph of a dataset graph, then all of its features are a subset of the features of the dataset graph. Therefore, the adjacent edges of each edge that appear in the graph code of the query will definitely appear in the graph code of the dataset graph.

D. Supergraph Querying

Supergraph query searches for the graph dataset members of which their whole features are contained in the input query. Formally, given a dataset $D = \{G_1,G_2,\ldots,G_n\}$ and a supergraph query q, if q is a supergraph of the dataset graphs then all of its features form a superset of the features of the resulted dataset graphs. The large number of graphs in datasets and the NP-completeness of subgraph isomorphism testing make it challenging to efficiently processing supergraph queries.

In our propose approach, when the query graph enters, it is represented as a query graph code. Each feature from query' graph code is probed in GCTrie. If all of the query features are matched with features data graph codes in the GCTrie but the query graph size is larger than the dataset graphs' size. Then the dataset graphs are returned as answer set that are contained in query as subgraph. The step-by-step process of supergraph query is described as the following algorithm 2.

Algorithm 2 SupergraphQuery

```
Input : GCTrie, and Query $q$
Output : Answer set $D_q$.
1.  Generate graph code for query $q$.
2.  Let $D_q = D$
3.  For each feature $qf \in c(q)$
4.      Probe $qf$ in GCTrie.
5.      If $qf \in$ GCTrie
6.          Intersect $D_q$ and $D_{qf}$.
7.  For each $G_i \in D_q$
8.      If size($G_i$) > size($q$)
9.          Remove $G_i$ from $D_q$.
10. Return $D_q$;
```

Assume that we have generated the graph code of the query graph. We establish a necessary condition that forms the basis for processing supergraph query. Then we state the following theorem.

Theorem 2 Given a query graph q, if q is a supergraph of a dataset graph G, then $c(G) \subseteq c(q)$.

Proof. By definition, if q is a supergraph of G, then every feature of G appears in q. Therefore, if parametric quantities of c(G) are contained in c(q), then $c(G) \subseteq c(q)$.

The intuition is as follows. If a query q is a supergraph of a dataset graph, then all of its features form a superset of the feature of the resulted dataset graphs. Therefore, the adjacent edges of each edge that appear in the graph code of the dataset graph will definitely appear in the graph code of the query.

V. Experimental Analysis

A performance analysis for proposed approach is presented in this section. The main goal of the experiment is to represent the performance evaluation of our proposed approach apply on AIDS antiviral screen compound dataset, NCI yeast anticancer drug screen dataset, and primary screening dataset for Formylpeptide Receptor.

Index Construction times for graph indexing approaches such as GraphGrep, OrientDB and our proposed approach are analyzed. All of the approaches are implemented in java on Intel(R) Core (TM) i3-4010U CPU with 2GB memory and Window7 34-bit operating system. Fig. 5, Fig. 6, and Fig. 7

shows the index construction times vary for different approaches on three datasets respectively. Various numbers of chemical graphs are tested and take the average index construction times to compare GraphGrep, OrientDB and our proposed approach. For GraphGrep, We use two values 4 and 10 for parameter: the length of path (l_p). It can be seen that our proposed approach consumes at least 10 times less than OrientDB in index construction and at least 10^2 times less than GraphGrep (l_p = 4) and GraphGrep (l_p = 10) respectively.

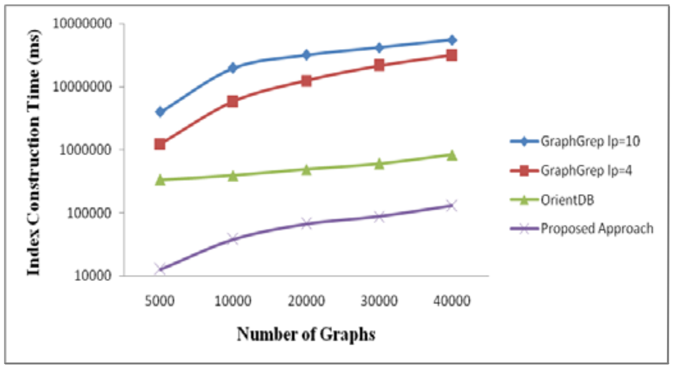

Figure 5. Analysis of Index Construction Time of Three Different Approaches for AIDS Antiviral Screen Dataset

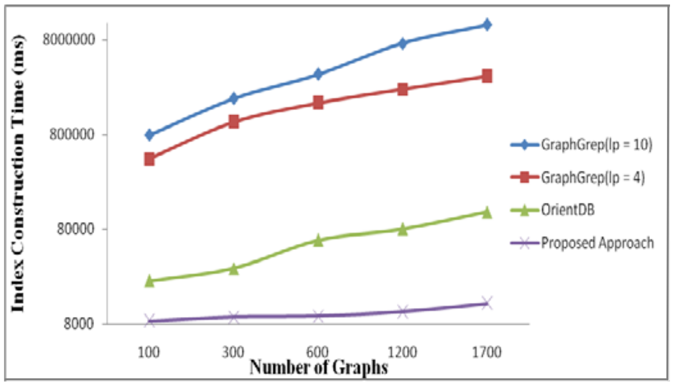

Figure 6. Analysis of Index Construction Time of Three Different Approaches for NCI Yeast Anti-cancer Drug Screen Dataset

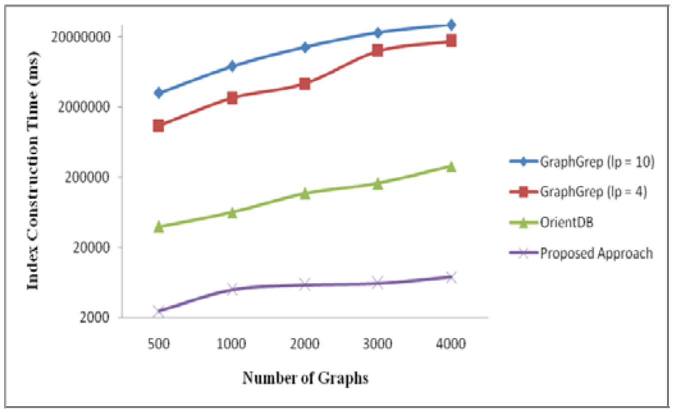

Figure 7. Analysis of Index Construction Time of Three Different Approaches for Primary Screening Dataset for Formylpeptide Receptor

We evaluate the query response time of our proposed approach with GraphGrep on AIDS antiviral screen dataset, NCI yeast anti-cancer drug screen dataset and primary screening dataset for Formylpeptide Receptor. Since GraphGrep only support subgraph isomorphism query, we can evaluate subgraph isomorphism query response time with it. For GraphGrep, we use two values 4 and 10 for parameter; the length of path (l_p). Fig. 8 shows the analysis of subgraph isomorphism query response time over AIDS antiviral screen dataset. It can be seen that our proposed approach significantly reduces at least 10^3 times for subgraph isomorphism query response time when compare to GraphGrep.

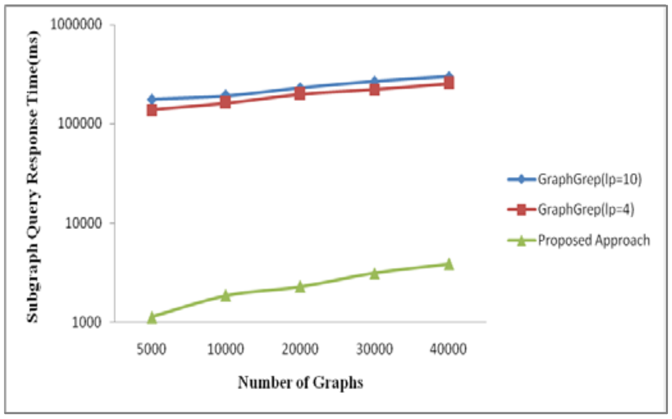

Figure 8. Analysis of Subgraph Query Response Time Between GraphGrep and Proposed Approach on AIDS antiviral screen Dataset

Fig. 9 shows the analysis of subgraph isomorphism query response time over NCI yeast anti-cancer drug screen dataset. Fig. 10 shows the analysis of subgraph isomorphism query response time over primary screening dataset for Formylpeptide Receptor. It can be seen that our proposed approach significantly reduces at least 10^2 times and 10^3 times of subgraph isomorphism query response time less than when compare to GraphGrep over NCI yeast anti-cancer drug screen dataset and primary screening dataset for Formylpeptide Receptor respectively.

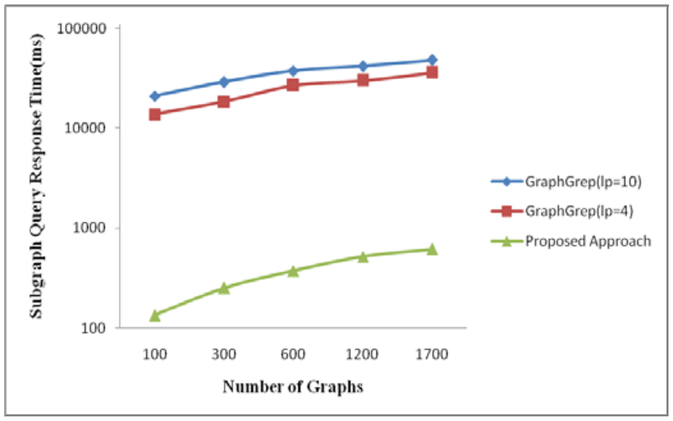

Figure 9. Analysis of Subgraph Query Response Time Between GraphGrep and Proposed Approach on NCI Yeast Anti-cancer Drug Screen Dataset

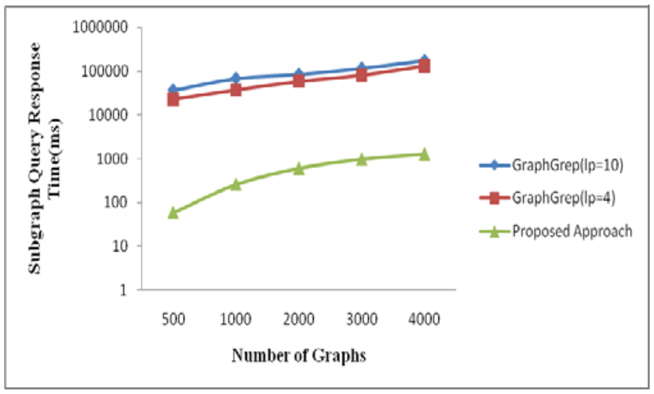

Figure 10. Analysis of Subgraph Query Response Time Between GraphGrep and Proposed Approach on Primary Screenin Dataset for Formylpeptide Receptor Dataset

VI. CONCLUSIONN AND ONGOING WORKS

Proposed graph code used edge dictionary and adjacent edges information to preserve the structural information of the original graph. Instead of expensive pair-wise comparisons, it can be efficiently used to detect automorphic graphs. Instead of path or subgraph decomposition process which could result in structural information lost and exhausted enumeration time, GCTrie is used to support all query types. From our experimental results, proposed approach outperforms the existing methods in index construction time and subgraph query response time. Similarity query processing is going to be observed as our ongoing work.

ACKNOWLEDGEMENT

This research was supported by University of Computer Studies, Yangon and University of Information Technology. We thank our colleagues who provided insight and expertise that greatly assisted the research.

REFERENCES

[1] Y. W. Hlaing and K. M. Oo, “A graph representative structure for detecting automorphic graphs,” in the Proceeding of 9th International Conference on Genetic and Evolutionary Computing, August 2015, pp.189-197.

[2] S. Sakr and G. Al-Naymat, “Graph indexing and querying : a review,” in International Journal of Web Information Systems, vol. 6 No.2, 2010, pp.101-120.

[3] Y. W. Hlaing and K. M. Oo, “Graph code based isomorphism query on graph data,” in the Proceeding of 2015 IEEE International Conference on Smart City/SocialCom/SustainCom together with DataCom 2015 and SC2 2015 (SmartCity 2015), Dec 2015, in press.

[4] D. Shasha, J. T. L. Wang, R. Giugno, “Algorithmic and Applications if Tree and Graph Searching”, 2002.

[5] OrientDB https://en.wikipedia.org/wiki/OrientDB

[6] R. N. Chittimoori, L. B. Holder, and D. J. Cook, “Applying the SUBDUE substructure discovery system to the chemical toxicity domain,” in Proceeding of the 12th international Florida AI, Research Society Conference, 2003, pp.90-94.

[7] D. A. Piriyakumar, and P. Levi, “An Efficient A* based algorithm for optimal graph matching applied to computer vision,” in GRWSIA-98, Munich, 1998.

[8] D. Dupplaw, and P. H. Lewis, “Content-based image retrieval with scale-spaced object trees,” in Proceeding of SPIE: Storage and Retrieval for Media Databases, Volume 3972, 2000, pp.253-261.

[9] Adjacency List http://en.wikipedia.org/wiki/Adjacency_list.

[10] Adjacency Matrix http://en.wikipedia.org/wiki/Adjacency_matrix.

AUTHORS PROFILE

Y. W. Hlaing received her bachelor degree in Computer Science (2006) from university of Computer Studies, Yangon and her master degree in Computer Science (2008) from Computer University, Mawlamyine. She is now a Ph.D candidate at University of Computer Studies, Yangon.

K. M. Oo received her B.Sc. in Mathematics from Yangon University; M.I.Sc. from Institute of Computer Science and Technology(ICST); and Ph.D. in Information Science from University of Computer Studies, Yangon(UCSY), Myanmar in 1994, 1996, and 2007 respectively. Currently, she is an associate professor in the Faculty of Computing, University of Information Technology (UIT), Myanmar. Her research interests include graph theory, and data mining.