Indexing Frequent Subgraphs in Large graph Database using Parallelization

Swati C. Manekar[1], Manish Narnaware[2]

[1, 2]Computer Science & Engineering Department, G. H. Raisoni College of Engineering, Nagpur, Maharashtra, India

Abstract: ***Plenty of structural patterns in real world have been represented as graph like molecules, chemical compounds, social network, road network etc. Mining this graph for extracting some useful information is of special interest and has many applications. The application includes drug discovery, compound synthesis, anomaly detection in network, social network analysis for finding groups etc. One of the most interesting problems in graph mining is graph containment problem. In graph containment problem ,given a query graph q ,it is asked to find all graph in given graph dataset containing this query (query graph as subgraph).This means finding all graph which is isomorphic to query graph. As in real world there is vast number of graph in graph dataset so this task of subgraph isomorphism test become tedious, complex, time and space consuming. So it is necessary to create an index of graphs present in dataset for cost efficient query processing. In this paper we proposed a time efficient graph indexing technique using discriminative frequent subgraph as indexing feature for molecular datasets using parallel approach. We proposed a method which will find frequent subgraphs using better pruning capability and executed in multithreaded environment in parallel manner. Our experimental studies conceal that parallelization method for graph indexing which has a condensed index structure, achieves an order of degree better performance in index construction, and significantly, outperforms state-of-the-art graph based indexing methods.***

Keywords: Graph indexing, graph mining, frequent structure based approach, parallelization approach.

1. Introduction

Graph data has grown steadily in various scientific and commercial areas. Chemical molecules, proteins and three-dimensional mechanical parts are modeled as graphs. Graphs also have broad applications in such areas as computer vision and image processing. In recent years, a number of data mining and management applications have been designed in the context of graphs and structural data .Several graph mining techniques have been developed to extract useful information from graph representation .In order to speed up graph queries, usually an index of the graph is derived according to some predefined index features.

Graph indexing is often utilized by graph search algorithms that look for a sub-graph within a graph database. Subgraph Search is one of the most popular graph retrieval models. In a graph dataset D, given a query graph q, a subgraph search algorithm retrieves all graphs in D containing q as a subgraph. This process of finding the subgraph isomorphic to query graph in graph database is nontrivial, as subgraph isomorphism problem is known to be NP-complete as shown in [1]. To solve the subgraph search problem, subgraph (subtree) features are commonly mined using several methods to build a graph index. As shown in Figure 1, in a 9-graph dataset, three subgraph features are mined to build a graph index. Given a query q containing features P1 and P3, any supergraph of q should have both P1 and P3 as subgraphs. Therefore, only graphs {G1, G2} = {G1, G2, G4, G8} ∩ {G1, G2, G5, G6, G7} are candidate graphs that need to be evaluated with subgraph isomorphism tests, and all the other database graphs are directly filtered out. The query processing time depends upon the number of subgraph isomorphism tests, which, in turn, depends on the filtering power of the feature set. As such an important aspect is a choice of good features. One more example can be given as a typical graph of router-level Internet consists of millions of nodes making it impractical to perform many operations on the whole graph. In such cases, graph indexing allows operations to be more efficient.

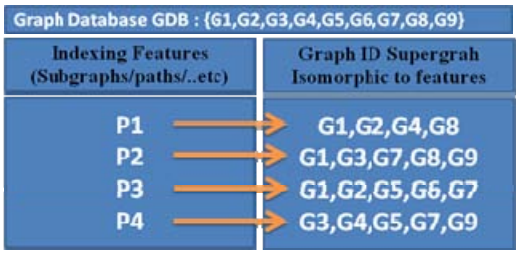

Figure 1: Graph Index

As it is like given a graph database GDB (a set of graphs) and a query graph Q, find those graphs in GDB that contain Q. We can test graph containment (subgraph isomorphism) for each graph in GDB that is sequentially scanning each graph in database for checking isomorphism. But the number of graphs in GDB can be extremely large, and also the graphs may be large in size. Therefore, this sequential approach is unfeasible .Unfeasible in the sense will require much more time, so not efficient solution. It is inefficient to perform a sequential scan on the graph database and check whether Q is a subgraph of GDB. Sequential scan is very costly because one has to not only access the whole graph database but also check subgraph isomorphism which is NP-complete.

Clearly, it is necessary to build graph indices in order to help processing graph queries. The indexing process in a graph matching methodology creates an index of the reference graph vertices along with their attributes so that the future referencing of the vertices for matching purpose becomes an efficient process. The data structures used for indexing usually determine the flow of the process. Many features, such as frequent and discriminative subgraph (subtree) features, δ−TCFG features and MimR (maximum information and minimum redundancy) features are mined to build the graph index and have certain significant filtering capabilities.

2. Graph Indexing Techniques

As subgraph isomorphism test is NP Complete problem [1], a filter and verification methods is generally applied to speed up the query processing task. As filtering process is key issue in most to improve the search efficiency many indexing method have been proposed. Most of these indexing methods can be grouped into three categories: path based indexing, frequent subgraph based indexing and graph decomposition based indexing.

Path-based indexing approach take path as the basic indexing unit, categorized as path-based indexing approach. For example GraphGrep [2] and Daylight [3] are Path-based graph indexing methods. They use path expressions as indexing features such as GraphGrep enumerates all paths in the graph up to the length maxL. This means enumerating all the existing paths in a database up to maxL length and indexes them, where a path is a vertex sequence, V1, v2, ……, vk, s.t.,$\forall 1 \leq i \leq k-1,\ (v_i, v_{i+1})$ is an edge. It uses the index to identify every graph gi that contains all the paths (up to maxL length) in query q. A significant feature of path-based approaches is that paths can be manipulated easier than trees and graphs and the index space is predefined: all the paths up to maxL length are selected. For answering to the structured or tree queries path based approach divide the queries into paths of different length and then look into each graph for these paths .Finally it collect all the graphs containing this path and display results. Since the structural information could be lost when breaking such queries apart. So Yan et. al. indicated, path is a simple structure loosing structural information of a graph [4]. In addition, the number of paths in a graph database increases exponentially making path-based methods impractical for very large graphs.

On the other hand structured based approach identifies subgraphs to be indexed as an indexing feature. As as Yan et. al. indicated that false positive ratio of path based methods would be very high alternatively, structure-based graph indexing approaches gIndex[4] first searches for the frequent subgraphs in the graph, then indexes these frequent structures**.** A case in the above paper discussed is that frequent subgraph discovery increases complexity and exponential number of frequent fragments may exist under low frequency support. Therefore, in their study, they limit the number of nodes and index frequent structures up to 10 nodes.

An alternative structural indexing approach to search and process queries efficiently even in very large graphs As indexing features, used commonly observed graph structures: star, complete bipartite, triangle and clique. An important feature of these structures is that each one is comprised from the previous one where clique contains complete bipartite structures and complete bipartite contains star structures.

In structural indexing, they have indexed predefined structures that are commonly observed in complex networks. In particular, index star, complete bipartite, triangle and clique structures in a given graph G = (V, E). An important difference of their approach from the previous studies is that they do not limit the size of subgraph considered in indexing. They have indexed all maximal graphs that match the structure formulation. For instance, a maximal clique is a clique that cannot be extended by adding one more vertex from the graph. However, the substructure size in indexing may be limited when needed since maximal clique search is known to be NPcomplete .In order to reduce computational complexities; they have indexed the structures within the original graph in a consecutive manner. They first identified star structures, and then the complete-bipartite, triangle and clique structures from the preceding one. But problem with this method is that finding complete clique is a NP complete problem also finding these types of structures having limited applications [5].

The tree-based indexing involves a tree data structure. Graph partitioning algorithms are used in order to obtain the vertices at various levels of the tree. For example, Top-k based subgraph matching algorithm uses a G-tree for index construction. This algorithm uses a heap structure for the matching process and to store the final result.

One of the recent indexing technique is neighborhood based method, which is employed by TALE [9], GADDI [8] and SAPPER [7]. During indexing, it is ensured that not only the vertex labels are stored, but the neighborhood of a vertex is also stored thus ensuring that the structural information is taken into consideration. Since the number of neighbors can be large in a very dense graph, normal storage strategies may prove to be inefficient. To tackle the storage issue, a hashing based methodology called bloom filter is used. TALE and SAPPER use bloom filter [6] for storing the index. GADDI focuses on a slightly different neighborhood approach called NDS (Neighbor Discriminating Substructure) distance for indexing. The NDS distance of a pattern P is defined as the number of matches of P present in an induced subgraph of neighboring vertices. An array is used for each of the vertices of the induced subgraph to store the NDS distances.

In indexing for multileveled graph using SAPPER algorithm plus some enhancement to accommodate multiple labels for vertices and edges given in paper [10]. Data structure used here is List. Indexing done in the five part 1)Labels of the vertex 2)Degree of the vertex 3)labels of neighbors 4)Labels of edges to neighbors 5)the labels of second level neighbors stored in bloom filter. Approached used here in this paper can be explained as follows;

Firstly the entire reference graph is loaded into primary memory then indexing process is stated. Data Structure used for storage is array, array (1) – Storing vertex information, array (2) – Storing edge information. Two phases for indexing is used first is vertex processing and second is edge processing. In vertex processing firstly a data structure is initialized, then for each vertex, vertex labels are inserted along with initialization of neighbor list structure. Now in edge processing each edge entry is traversed first then updates the neighbor lists corresponding vertices with the label of its neighbors .Update labels of the edge connecting these neighbors and also update vertex identifiers of these neighbors. After this one label for edge and neighbor vertex stored in neighbor list and finally edge processing completed. Labels of second level neighbors are stored in bloom filter for each vertex.

In graph decomposition based indexing technique, graph decomposition is applied directly to replying queries of isomorphism and subgraph isomorphism [11], [12] .Graph decomposition is applied before going for graph isomorphism testing. Two short coming with these method are one is that they have to enumerate all connected subgraphs and hence complexity is exponential to graph size which is to be decomposed and other is frequent information in decomposition results are not improving the efficiency of graph similarity search.

One more method uses both graph decomposition method and frequent subgraph method for indexing graph[13] Here a graph is decomposed into set of *k*-Adjacent trees and decomposed result are indexed by a *K*-AT index. It store more structural information as compared to normal graph decomposition (breakdown) methods.

So in this paper we proposed an indexing method which uses frequent subgraphs an indexing feature for very large database .By using a proper pruning strategy and parallel approach it minimize the time required to construct an index.

3. Proposed Indexing Method

The main aim of Graph Indexing is to reduce the set of graphs without loss of result, means graph indexing must promise pruning non promising graphs.

The graph indexing technique must not be very expensive i.e. it should not take much time and memory to create index. Secondly it should have higher pruning capability without loss of result, and finally it should not generate false answer.
To deal with the problem of space we used parallel and distributed approach with multiple processors with their individual memory running in parallel to give good speed. For high pruning capability we used close fragment pruning with the loss less result.

User can define the support to index all frequent subgraphs usually low-support large fragments may be indexed well by their smaller subgraphs. Especially, it will be always chosen the (absolute) minSup to be 1 for size-0 fragment to ensure the completeness of the indexing. This method having two rewards first one is that the number of frequent fragments so obtained is much less than that with the lowest uniform minSup, and second one is low-support large fragments may be indexed well by their smaller subgraphs; thereby it will not miss useful fragments for indexing by using frequent fragments with the size-increasing support constraint, one has a smaller number of fragments to index. However, the number of indexed fragments may still be huge when the support is low. For example, 1,000 graphs may easily produce 100,000 fragments of that kind. It is both time and space consuming to index them. To overcome this parallel and distributed approach has been used in our approach. This can be explained as follows,

- Identify Frequent Structures in the database; the frequent structures are sub graphs that appear quite often in the graph database.
- Prune redundant frequent structures to maintain a small set of discriminative structures.
- Create an inverted index between frequent structures and graphs in the database.
- Enumerate structures in the graph database build an inverted index between structures and graphs.
- A graph or structure is frequent if its support i.e. occurrence frequency no less than the minimum support threshold.

Indexing feature used here is frequent structures as they are the quality candidates not losing the structural information giving lossless results. This whole process of indexing is parallelized to achieve time efficiency.

We are using molecular database as an input. First step is to represent molecules as attributed graph. DFS is used to get into the search tree which will ensure that each vertex get covered in the graph. Now the next step is generation of candidate subgraphs i.e. so called pattern growth. This can be done by extending a fragment by adding node or edge. If it is closing ring then adding an edge only otherwise adding a node.

For finding Frequent Subgraphs it is important to review how a graph database is searched for frequent subgraphs: Starting from the seed node or user defined node for which all possibilities must be tried a subgraph is extended by adding a node or edge in every step. Condition here is that in this stepwise extension process one requires that at least one node which is a part of an added edge must already be there in subgraph. These means that the search is restricted to connected subgraphs which are necessary for most applications. In its most basic form the search considers all possible extensions of the current subgraph by an edge and, if necessary, a node. The set of extensions can be reduced by exploiting a canonical description used in gSpan. Note that, as a consequence of the above, the search produces a numbering of the nodes in each subgraph: the steps in which these nodes are added corresponds to node indexes and similar in case of edges also edges are also associated with the indexes in the order in which they are added. This search generates a spanning tree of the subgraph, which is improved by additional edges. Depth-first and breadth-first search are straightforward systematic methods for constructing a spanning tree of a graph. Other alternatives include a spanning tree construction that first visits all neighbors of a node (like breadth-first search), but then chooses the next node to extend in a depth-first search manner.

As in extending search tree, may generate duplicate subgraphs, so to avoid redundant search Canonical form pruning, closed subgraph pruning is used.

For calculating the support embeddings are stored .The support of the fragment (Sub graph) is determined by simply counting the number of different molecules the embedding refers to. As for large data input frequent fragment can be very large in size so only discriminative fragments are derived which give compact index. Discriminative fragments are those fragments which are high in active molecules, low in inactive molecules.
After identifying all frequent discriminative subgraph an inverted index has been created between these subgraphs and the graphs in database.

This whole process can be summarized in the form of flow char as shown below;

The simultaneous use of more than one CPU or processor core to execute a program or multiple computational threads give higher speed of execution. So this process of finding frequent subgraphs in the database is divided into multiple threads and executed on different processors in parallel fashion .Ideally, parallel processing makes programs run faster because there are more engines (CPUs or Cores) running it. So we have divided a program of graph miner (finding frequent fragments) in such a way that separate CPUs or cores can execute different portions without interfering with each other. Parallel processing provide faster execution time so higher throughput. Overall execution time for creating the index of graph got reduced. The parallel processing architecture for creating graph index can be shown in fig. 3.

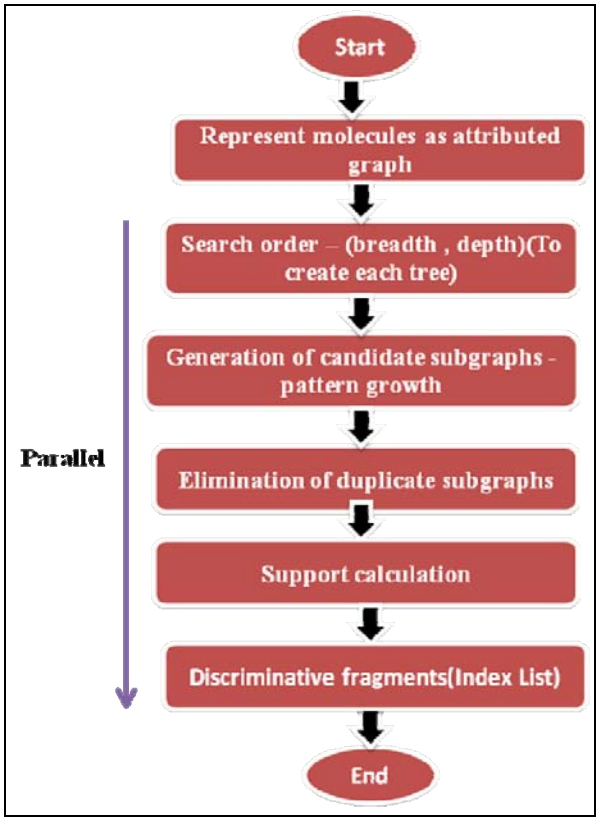

Figure 2: Graph Indexing Flow Chart

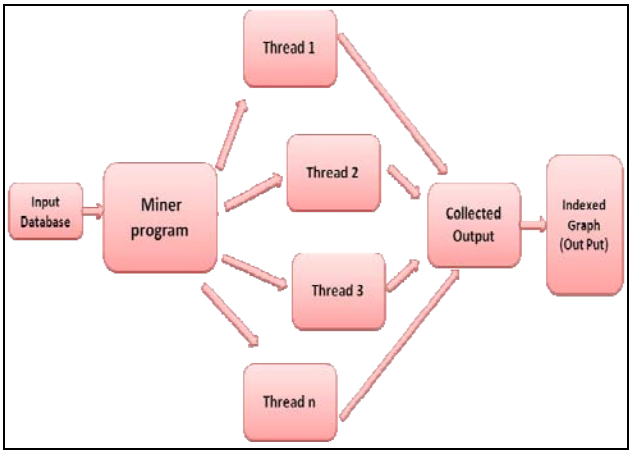

Figure 3: Parallel Processing Module

Here the miner program is collection of the different methods for finding frequent discriminative fragments in the graph along with the pruning strategies for avoiding irrelevant candidate answers.

4. Conclusion

The different approaches for graph indexing having some advantages and drawbacks. In order to speed up graph queries, usually an index of the graph is derived according to some predefined index features. Graph indexing is used for efficient graph mining. As many graph data sets are defined on massive node domains in which the number of nodes in the underlying domain is very large the indexing techniques implement require more time. The performance of the graph indexing has been enhanced and speeded up by using parallelization approach by running the program in parallel on two processors.

References

[1] S. A. Cook, “The complexity of theorem-proving procedures,” in STOC, 1971, pp. 151–158
[2] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In Symposium on Principles of Database Systems, pages 39–52, 2002.
[3] C. A. James, D. Weininger, and J. Delany. Daylight theory manual daylight version 4.82. Daylight chemical information systems, 2003.
[4] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach, 2004.
[5] Hakan Kardes¸ and Mehmet Hadi G¨unes. Structural Graph Indexing for Mining Complex Networks. 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops.
[6] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.
[7] S. Zhang, J. Yang, and W. Jin, “Sapper: subgraph indexing and approximate matching in large graphs,” Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 1185–1194, 2010.
[8] S. Zhang, S. Li, and J. Yang, “Gaddi: distance index based subgraph matching in biological networks,” in Proceedings of the 12th International Conference on

Extending Database Technology: Advances in Database Technology. ACM, 2009, pp. 192–203.
[9] Y. Tian and J. Patel, "Tale: A tool for approximate large graph matching," in Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. IEEE, 2008, pp. 963–972.
[10] Varun Krishna , NNR Ranga , Suri G Athithan, MuGRAM: An Approach for Multi-labelled Graph Matching. Centre for Artificial Intelligence and Robotics Bangalore, India. 978-1-4673-0255-5/12/$31.00c 2012 IEEE.
[11] D. Eppstein, "Subgraph Isomorphism in Planar Graphs and Related Problems," J. Graph Algorithms and Applications, vol. 3,no. 3, pp. 1-27, 1999.
[12] J.P. Kukluk, L.B. Holder, and D.J. Cook, "Algorithm and Experiments in Testing Planar Graphs for Isomorphism," J. Graph Algorithms and Applications, vol. 8, no. 3, pp. 313-356, 2004.
[13] Guoren Wang, Bin Wang, Xiaochun Yang," Efficiently Indexing Large Sparse Graphs for Similarity Search" IEEE TRANSACTIONS ON KNOWLEDGE ANDATA ENGINEERING, VOL. 24, NO. 3, MARCH 2012

Author Profile

Swati Manekar is undergoing her Maters Degree in Computer Science and Engineering in G H Raisoni College of Engineering, Nagpur. She has completed her undergraduate degree in year 2009 in Computer Engineering First Class. Her research interests are Graph Mining and Distributed & parallel processing.

Manish Narnaware has completed his Masters Degree in year 2010 from department of Computer Science and Engineering, VNIT Nagpur, with first class. He did undergraduate degree in year 2002 from VNIT Nagpur. He has around 4 years of professional experience. His research interests are distributed & parallel processing, Computational Mathematics. Best paper published by him is "practical approaches of image encryption/scrambling using 3D Arnolds Cat map" on CNC 2012, Springer Link Digital Library.