Maximal Frequent Subgraph Mining

Thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Engineering

by

Lini Teresa Thomas
200599002
lini@research.iiit.ac.in

Center for Data Engineering
International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2010

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Maximal Frequent Subgraph Mining" by Lini Teresa Thomas, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Advisor: Prof. Kamalakar Karlapalem

Acknowledgments

First and foremost, I thank my advisor Prof. Kamalakar Karlapalem. It has been an honor to be his Ph.D. student. His strong support has been a constant assurance to me throughout my academic life under him by making everything look solvable and possible. My colleague Satyanarayana Valluri with whom I have spent many hours discussing and working with, has contributed greatly in making this thesis fun working on. Many thanks to my family who encouraged my joining the program and have strongly backed my completing the work. The International Institute of Information Technology, Hyderabad has brought me in touch with a new world of ideas, research and exceptional people who have contributed in a big way to my academic and personal growth. Finally, I thank my silent most and youngest supporter - my month old daughter Meera.

Abstract

The area of graph mining deals with mining frequent subgraph patterns, graph classification, graph clustering, graph partitioning, graph indexing and so on. In this thesis, we focus only on the area of frequent subgraph mining and more precisely on maximal frequent subgraph mining.

The exponential number of possible subgraphs makes the problem of frequent subgraph mining a challenge. The set of maximal frequent subgraphs is much smaller to that of the set of frequent subgraphs, providing ample scope for pruning. MARGIN is a maximal subgraph mining algorithm that moves among promising nodes of the search space along the "border" of the infrequent and frequent subgraphs. This drastically reduces the number of candidate patterns in the search space. Experimental results validate the efficiency and the utility of the technique proposed. MARGIN-d is the extension of the MARGIN algorithm which can be applied to finding disconnected maximal frequent subgraphs. A theoretical comparison with Apriori like algorithms and analysis are presented in this thesis.

Further, sometimes frequent subgraph mining problems can find simpler solutions outside the area of frequent subgraph mining by applying itemset mining to graphs of unique edge labels. This can reduce the computational cost drastically. A solution to finding maximal frequent subgraphs for graphs with unique edge labels using itemset mining is presented in the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is common to model complex data with the help of graphs consisting of nodes and edges that are often labeled to store additional information. Representing these data in the form of a graph can help visualise relationships between entities which are convoluted when captured in relational tables. Graph mining ranges from indexing graphs, finding frequent patterns, finding inexact matches, graph partitioning and so on. Washio and Motoda[63] states the five theoretical bases of graph-based data mining approaches as subgraph categories, subgraph isomorphism, graph invariants, mining measures and solution methods. The subgraphs are categorized into various classes, and the approaches of graph-based data mining strongly depend on the targeted class. Subgraph isomorphism is the mathematical basis of substructure matching and/or counting in graph-based data mining. Graph invariants provide an important mathematical criterion to efficiently reduce the search space of the targeted graph structures in some approaches. Furthermore, the mining measures define the characteristics of the patterns to be mined similar to conventional data mining.

For a graph of e edges, the number of possible frequent subgraphs can grow exponentially in e. Further, the core operation of subgraph isomorphism being NP-complete, it is critical to minimize the number of subgraphs that need to be considered for finding the frequency counts or use other strategic methods such as information of the occurrences of the subgraph in the database that could avoid subgraph isomorphism. These factors make graph mining challenging. Given a set of graphs $\mathbb{D} = G_1, G_2, \ldots, G_n$, the support of a graph g is defined as the fraction of graphs in $\mathbb{D}$ in which g occurs. The graph g is frequent if its support is at least a user specified threshold. Mining frequent structures in a set of graphs is an important problem with applications such as chemical and biological data, XML documents, web link data, financial transaction data, social network data, protein folding data, etc. In molecular data sets, nodes correspond to atoms and edges represent chemical bonds. In the area of drug discovery, the chemical compounds are modeled as graphs and graph mining is used in classification [26] and to find frequent substructures [25]. Interesting patterns like web communities can be mined using the links information in web as a graph [44].

One performance issue in mining graph databases is the large number of recurring patterns. The large number of frequent subgraphs reduces not only efficiency but also the effectiveness of mining

since users have to process a large number of subgraphs to find useful ones. A subgraph X is a closed subgraph if there exists no subgraph X' such that X' is a proper supergraph of X and every transaction containing X also contains X'. A closed subgraph X is frequent if its support is greater than the user given support threshold. All frequent subgraphs along with their corresponding support values can be derived from the set of closed subgraphs. However, the closed subgraphs are a lot fewer than frequent ones. Also, association rules extracted from closed sets have been proved to be more concise and meaningful, because redundancies are discarded. However, the number of closed frequent subgraphs too is much larger as compared to the number of maximal frequent subgraphs. Many applications require the maximal frequent subgraphs alone. This gave rise to the area of mining maximal frequent subgraphs.

A frequent subgraph is maximal if none of its super graphs are frequent. Mining only maximal frequent subgraphs offer the following advantages in processing large graph databases[36].

- It significantly reduces the total number of mined subgraphs. The total number of frequent subgraphs is up to one thousand times greater than the number of maximal frequent subgraphs. Thus, the time and space is significantly reduced on mining maximal frequent subgraphs.

- All the frequent subgraphs can be generated from the set of maximal frequent subgraphs. The support of the frequent subgraphs is certain to be at least as high as the frequency of the maximal subgraphs. The actual support can be counted by scanning the graph database. Even otherwise, techniques used in [15] can be easily adapted to approximate the support of all frequent subgraphs within some error bound.

- For certain applications, the set of maximal frequent subgraphs itself are of most interest. Maximal frequent substructures may provide insight into the behaviour of a molecule. For example, researchers exploring immuno deficiency viruses may gain insight into recombinants via commonalities within initial replicants. In addition, it can be used to group previously unknown groups, affording new information. Graph-based methods can be applied on proteins to predict folding sequences and also maximal motifs in RNAs. Other applications that require the computation of maximal frequent subgraphs are mining contact maps [34], finding maximal frequent patterns in metabolic pathways [45], and finding set of large cohesive web pages. Similarly, given a set of web communities (eg,orkut) the maximal frequent subgraphs would generate all the largest groups of users and their relationships who together belong to many communities and hence might be users of similar interests.

A typical approach to frequent subgraph mining problem has been to find frequent subgraphs incrementally in an Apriori manner. Canonical labeling has been adopted to reduce the number of times each candidate is generated [65, 35]. The Apriori based approach has been further modified to suit closed subgraph mining [66] and maximal subgraph mining [36] with added pruning.

SPIN[36] is the only algorithm in the literature of graph mining for mining maximal frequent subgraphs explicitly apart from MARGIN. It is of course possible to generate all the frequent or closed

frequent subgraphs and then prune them to get the maximal frequent subgraphs. This can be expected to be highly inefficient as compared to algorithms developed specifically for the problem of maximal frequent subgraph mining.

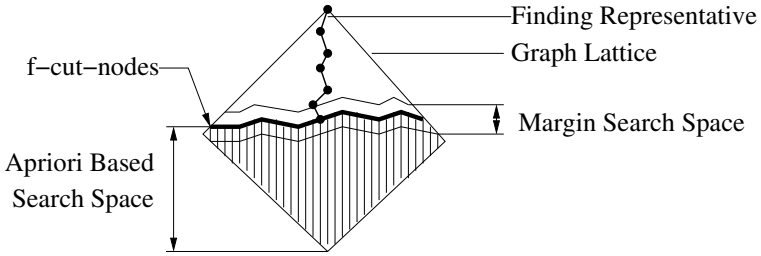

Figure 1.1 Search Space Explored

The maximal frequent subgraphs lie in the middle of the lattice such that all the subgraph below the maximal frequent subgraphs are frequent and the ones above are infrequent. Finding maximal subgraphs using Apriori methods would require bottom-up traversal of the lattice wherein the frequent subgraphs that are not maximal, though are of no interest, need to be explored and processed in order to reach the maximal frequent subgraphs. It was hence of interest to check whether it is possible to avoid the exploration of the space above or below the lattice and yet somehow attain the set of maximal frequent subgraphs which form something like a border between the set of frequent and infrequent subgraphs in the lattice. We developed a technique that would, in some manner, locate one maximal frequent subgraph, and then jump from one maximal frequent subgraph to another, exploring all the maximal frequent subgraphs of the lattice. Such a method should minimise the exploration of any node that is not a member of the set of maximal frequent subgraphs.

For a graph G, the graph lattice of G is a graph where the connected subgraphs of G form the nodes of the lattice. An edge occurs between two nodes of a lattice if the subgraph represented by one node is the immediate subgraph of the subgraph represented by the other node. We can visualise the graph lattice to have levels where the bottom-most level is the empty subgraph and the next higher level has all single node subgraphs of G and so on where the top-most node will be G itself. If level l has subgraphs of n edges then level $l + 1$ has subgraphs with $n + 1$ edges. Edges exist between two subgraphs in the lattice if one is the supergraph of the other formed by adding one edge. Hence, edges exist between two consecutive levels of a lattice. A graph lattice is defined in detail in the next chapter.

The set of subgraphs which are likely to be maximally frequent are the set of n-edge frequent subgraphs that have a $(n + 1)$-edge infrequent supergraph. We refer to such a set of nodes in the lattice as the set of $f(\dagger)$-$nodes$ (Figure 1.1) which form our candidate set. The set of maximal frequent subgraphs is a subset of this candidate set such that all its $(n + 1)$-edge supergraphs are infrequent. We propose the MARGIN algorithm that computes such a candidate set efficiently. By a post-processing step our algorithm then finds all maximally frequent subgraphs by retaining only those subgraphs that have all its immediate supergraphs to be infrequent. The $ExpandCut$ step invoked within the MARGIN algorithm recursively finds the candidate subgraphs by jumping from one candidate node to another.

Apriori based algorithms enumerate the frequent subgraphs or the maximal frequent subgraphs by a bottom-up approach and prune using the property that an infrequent subgraph cannot have a frequent supergraph. Hence, in order to find maximal frequent subgraphs, the frequent subgraph space needs to be explored. The SPIN[36] algorithm applies further pruning on the set of frequent subgraphs in order to find the maximal frequent subgraphs. The search space of Apriori based algorithms corresponds to the region below the $f(\dagger)$-$nodes$ in the graph lattice as shown in Figure 1.1. On the other hand, MARGIN explores a much smaller search space by visiting the lattice around the $f(\dagger)$-$nodes$. It can be clearly seen in the figure that if the border separating the frequent and infrequent subgraphs which is formed by the $f(\dagger)$-$nodes$ lies in the lower sections of the lattice, then there is a possibility of the space explored by both MARGIN and Apriori based algorithms being approximately the same. However, for lower supports where the border between the frequent and infrequent subgraphs lies higher up the lattice, the search space is distinctly much more for Apriori based algorithms. Table 1.1 shows the number of nodes visited by Margin and SPIN[36] for the two input parameters to the algorithm, namely, database size and support value. The variable D in column one refers to the number of graphs in the database and "support%" refers to the percentage of graphs in the database a subgraph should occur in inorder to qualify as frequent. Experimental results as in Table 1.1 show that our algorithm explores one-fifth of the nodes explored by SPIN[36].

$E = 10, V = 10, L = 10, I = 5, T = 6$		
DataSet	Lattice Nodes Visited	
D(**Support%**)	**SPIN**	**MARGIN**
100 (2)	43,861	9,311
200 (2)	54,026	10,916
300 (2)	57,697	14,954
400 (2)	58,929	42,201
100 (5)	42,584	9,930
200 (5)	49,767	12,318
300 (5)	52,118	24,660
400 (5)	54,726	44,686
500 (2)	32,556	12,619
500 (3)	21,669	10,078
500 (4)	9,187	9,912
500 (5)	4,162	8,264

Table 1.1 Lattice Space Explored

We prove that our algorithm finds all the nodes that belong to the candidate set. We show that any two candidate nodes are reachable from one another using the $ExpandCut$ function used to explore the candidate nodes. Given that the graph lattice is dense and there exists several paths that connect two candidate nodes, one of the main challenges of the proof is the abstraction of the sublattice that contains the path taken by the $ExpandCut$ function. By the construction of such a sublattice we show that such

a sublattice is guaranteed to exist. A reachability sequence is then shown that starting at any candidate node would reach another candidate node.

We study the behaviour of the MARGIN algorithm on synthetic datasets for a combination of dataset parameters and support values. We present results with our experiments on real life datasets namely, the records of page views by the users of msnbc.com and stock market data. We further compare the performance of the MARGIN algorithm with known standard techniques like gSpan, SPIN[36] and CloseGraph[66].

We further show in our work that the proposed algorithm can be generalised to be applied to a wider range of lattice based problems. We study the properties that makes such a horizontal sweep of the lattice by moving along the "border" of the frequent and infrequent subgraphs possible and generalise the solution that can be adapted to various other applications.

It is common to model complex data with the help of graphs consisting of nodes and edges that are often labeled to store additional information. The problem of maximal frequent subgraph mining can often be complex or simple based on the kind of input dataset. If the graph database is known to satisfy some constraint, like constraints on the size of graphs, the node or edge labels of graphs and the degree of nodes, it might be possible to solve the problem of subgraph mining without using any graph techniques by exploiting the properties of the constraint. Though the class of such graphs might be small, this problem is worth investigating since it could lead to immense reduction in run time.

The complexity of itemset mining algorithms is much less than that of graph mining algorithms because of the following reasons: (i) In itemset mining algorithms, an itemset can be generated exactly once by using a lexicographical order on the items. However, in order to avoid multiple explorations of the same subgraph, graph mining algorithms use expensive canonical normal form computation, (ii) the frequency of an itemset can be determined based on tid-list intersection of certain subitemsets. On the other hand, even the presence of all subgraphs of a graph g_1 in another graph g_2 does not guarantee the presence of the graph g_1 in g_2. Hence, detecting the presence of a subgraph in a graph requires additional operations, and (iii) Subgraph isomorphism being NP-Hard makes it critical to minimize the number of subgraphs that need to be considered for finding the frequency counts or use other strategic methods such as information of the occurrences of the subgraph in the database that could avoid subgraph isomorphism. Finding whether an itemset is a subitemset of another is trivial as compared to determining whether a graph is a subgraph of another.

Since itemset mining algorithms are known to be less complex compared to graph mining algorithms, we show how this aspect can be used effectively used to reduce the time and memory of graph mining algorithms. In Chapter 6 we present an itemset based subgraph mining algorithm(ISG) for graphs of unique edge labels which finds the maximal frequent subgraphs by invoking a maximal itemset mining algorithm.

1.1 Organisation and Contributions

- In Chapter 2, we give details of related work in the area of subgraph mining and the utility of maximal frequent subgraph mining.

- In Chapter 3, we first present the preliminary definitions and terminology that will be used throughout the MARGIN algorithm followed by a brief discussion in Chapter 3.2 on the two models in which the graph database is viewed and processed in our work. Next we give the intuition to the algorithm in Chapter 3.3 with a running example. We present the naive MARGIN algorithm in Chapter 3.4. The MARGIN algorithm presented in Chapter 3.4 gives connected maximal frequent subgraphs. The MARGIN algorithm can be modified to give disconnected maximal frequent subgraphs which is presented in Chapter 3.5

- In Chapter 3.6, we present proof showing that all maximal frequent subgraphs are found by our technique. The proof requires the generalisation of a graph lattice to show that given $f(\dagger)$-$node$, all the $f(\dagger)$-$nodes$ can be reached. The pruning step to find all the maximal frequent subgraphs from the set of $f(\dagger)$-$nodes$ is trivial.

- The MARGIN algorithm presented in Chapter 3 is the naive version of the algorithm which can be further optimised to give a significant saving in the running time. In Chapter 3.7, we give the optimisation techniques that can be applied. We further present the performance results on applying the optimisation techniques to the naive algorithm.

- Given a RNA database, it is interesting to identify different characteristics that are common across RNA molecules. RNA motifs are believed to be able to provide the ultimate basis for the understanding of the RNA structure and function. Motif identification is closely related to finding maximal frequent subgraphs of the RNA database. We apply the MARGIN algorithm to the Ribonucleic Acid database in Chapter 5.2.

- In Chapter 5.1, we discuss further applications of the MARGIN algorithm. It can been seen that the framework of the $ExpandCut$ function can be applied to the problems that meet a certain set of criterions which is explored in Chapter 5.1.1. This widens the scope of application of the MARGIN algorithm. The MARGIN algorithm itself can be applied to disconnected maximal frequent subgraph mining as well. The problem of finding disconnected maximal frequent subgraphs is more complex than when the reported maximals are to be connected. The modification of MARGIN to compute disconnected maximal frequent subgraphs is discussed in Chapter 3.5.

- In Chapter 4, we present experimental results on both synthetic and real life datasets. We first analyse the behaviour of MARGIN for varying support values, database sizes, graph sizes, number and size of frequent subgraphs, edge and vertex labels, etc. We then compare MARGIN with other maximal frequent subgraph mining algorithms like SPIN[36], gSpan and CloseGraph[66]. We also present results of the extension of MARGIN to finding maximal disconnected frequent

subgraphs. We further apply MARGIN to real life datasets like page views from msnbc.com, stock market data from the Yahoo Finance Stock site[8] and chemical datasets used widely available at [2].

- In Chapter 6, we propose a new algorithm, ISG that mines maximal frequent subgraphs of a constraint-based graph database using a maximal itemset mining algorithm. The algorithm opens the possibility of avoiding expensive subgraph mining techniques for databases that have constraints. The experimental results show that ISG performs significantly better than existing methods showing the utility of our approach.

Chapter 2

Related Work

Research on pattern discovery has advanced from itemset and sequence mining to mining trees, lattices and graphs. The initial work towards applying Apriori techniques towards structured data was done in R. Agrawal and R. Srikant [12] for sequence mining.

The earliest work on finding subgraph patterns was a greedy approach to avoid the high complexity of subgraph isomorphism, developed in SUBDUE [23] and K. Yoshida et al. [69]. These did not report the complete set of frequent subgraphs. SUBDUE [23] compresses the original graph using the minimum description length principle. However, due to heuristic and greedy solution based approaches, they miss significant patterns.

WARMR [24] reported the complete set of frequent substructure in graphs. WARMR used ILP-based methods along with the non-monotone Apriori property. WARMR however cannot be extended to disconnected frequent subgraph mining. Also, the approach has very high computational complexity due to equivalence checking done by them. In 1999, A. Inokuchi et al. [38] came up with an approach to find frequent subgraphs in a graph database. This scope of this work was limited to graphs that have unique node labels. Apriori-based Graph Mining (AGM) [37] was a computationally efficient algorithm proposed to find frequent induced subgraphs. AGM uses Apriori like technique to extend subgraphs by adding one vertex at a time. The algorithm does a join operation based candidate generation and finds the normalised form of the adjacency matrices of each generated subgraph. However, since the same graph can have multiple normalised forms based on the subgraphs from which it was generated, the normalised form cannot be used to uniquely represent a subgraph. Hence, the algorithm gives a method to convert normalised graphs into its canonical form in order to efficiently find the frequency of the subgraph in the graph database. Frequent Subgraph Mining Algorithm (FSG) [47] enumerates all the frequent subgraphs and not just the induced frequent subgraphs, by the join operation. It generates supergraphs by adding one edge at a time. FSG has to however address both graph isomorphism as well as subgraph isomorphism. Hence, when the number of edge and vertex labels are low, pruning becomes difficult and hence the performance deteriorates. One main reason for AGM's slow computation is that AGM produces all possible combinations of induced frequent subgraphs while FSG generates only frequent connected subgraphs [47].

Enumerating all the subgraphs can be done either by the join operations as in AGM [37] and FSG [47] or by the extension operations as in C. Borgelt and M. R. Berthold [13], J. Huan et al. [35], GSpan [65] or both as in J. Huan et al. [35]. Join-based algorithms suffer from the very expensive candidate generation phase. Frequency computation is also computationally very expensive due to subgraph isomorphic checks. Itemset mining uses hash trees in order to find the presence of an itemset in a transaction. It is challenging to build such a hash tree for graphs [47]. Apriori based algorithms map each pattern to a unique canonical label. By using these labels, a complete order relation is imposed over all possible patterns and ensures that only one isomorphic form is extended [65]. Graph-Based Substructure Pattern Mining (GSpan) [65] follows a depth first approach where a generated subgraph is checked with previously generated subgraphs to know whether it has been previously generated using the DFS-code. The DFS-code is a five-tuple (i,j,l_i,$l_{i,j}$,l_j) with i and j being the identifier numbers of its incident nodes (assigned by their order of appearance in the sequence), l_i and l_j being their labels and $l_{i,j}$ being the label of the edge. The smallest DFS code representation of a graph is defined as its minimum DFS code, which is unique and is used as canonical form for the mining process. gSpan traverses the graph database in a depth-first manner by extending every subgraph that was found to have a min-DFS code and then was found to be frequent. gSpan combines frequent subgraph extension and subgraph isomorphism into one procedure which is the reason for its speed up. Also, the expensive candidate generation done by join-based algorithms is completely avoided. The memory usage for breadth first search(BFS) based algorithms is high as the subgraphs of every level have to be saved in order to generate the next level of subgraphs. Further, GSpan [65] mention a clear speed up due to the shrinking of graphs in each iteration. This is done by not considering edges of lower lexicographic order during the depth first extensions of some single edge.

J. Huan et al. [35] uses a canonical ordering on the sequence of lower triangular entries which prunes isomorphic graphs. M. Cohen and E. Gudes [22] formulates frequent graph mining framework in terms of reverse depth search and a prefix based lattice. GASTON [50] efficiently finds frequent subgraphs in the order of increasing complexity by searching first for frequent paths, then frequent undirected trees and finally cyclic graphs. J. Wang et al. [61] uses a partition based approach to find frequent subgraphs for databases that do not fit into main memory. G. Buehrer et al. [14] shows how allowing the state of the system to influence the behavior of the algorithm can greatly improve parallel graph mining performance in the context of CMP architectures.

M. Kuramochi and G. Karypis [49] attacks the tougher problem of frequent subgraph mining where frequency of a graph is the total number of occurrences in the graph database (single-graph setting) unlike GSpan [65], J. Huan et al. [35], AGM [37], J. Pei et al. [53] wherein frequent refers to the number of graph transactions that a subgraph occurs in(graph transaction setting). Solutions of subgraph mining in the single graph setting can be applied to the problem of graph transaction setting but not vice versa. Y. Ke et al. [43] and H. Tong et al. [59] propose a new problem of correlation mining for graph databases given a query graph. ORIGAMI [31] mines a representative set of graph patterns. T. Washio

and H. Motoda [63] provides a comprehensive survey of graph based data mining approaches since the mid 1990's.

Considerable work has gone into closed itemset mining [54, 72]. The exponential size of frequent subgraphs has led to an interest in closed graphs [66, 62, 74] which is a much smaller set as compared to the set of frequent graphs. An even smaller set is the set of maximal frequent subgraphs.

Finding maximal frequent itemsets has been a well explored problem [15, 28, 29]. The maximal frequent subgraph mining problem has been addressed in SPIN [36]. Spanning tree based maximal graph mining (SPIN) algorithm[36] first mines all the frequent tree patterns from a graph database and then constructs the maximal frequent subgraphs from the frequent trees. This approach offers asymptotic advantages compared to using subgraphs as building blocks, since tree normalization is a simpler problem than graph normalization. Tree normalization is a sequence of transformations that transforms an original expression form one form into an equivalent one which helps in comparing whether two subgraphs/trees are the same and hence has been visited earlier. There are two important components in the framework. The first one is a graph partitioning method through which all frequent subgraphs are grouped into equivalence classes based on the spanning trees they contain. The second important component is a set of pruning techniques which aim to remove some partitions entirely or partially for the purpose of finding the maximal frequent ones only. Trees are then expanded to cyclic graphs by searching their search spaces and the maximal frequent subgraphs are constructed from the frequent ones. SPIN [36] also introduces some optimization techniques to improve the search for the maximal frequent subgraphs. S. Srinivasa and L. BalaSundaraRaman [57] uses a filtration based approach that starts by assuming all graphs to be isomorphic. It removes edges from graphs that contradict this assertion until it converges to the maximal frequent subgraphs.

In our work(Chapter 3), we focus on undirected connected labeled graphs but it is trivial to fit our algorithm to directed graphs, disconnected graphs(3.5) and partially and unlabeled graphs. The MARGIN algorithm can be easily modified to find the maximal cliques by changing the lattice definition and the definition of frequent and infrequent subgraphs in the lattice.

Closely related is the work on the frequent subtree mining problem [20, 19, 21, 71], frequent graphs with geometric constraints [48], discovering typical patterns of graph data [60] and graph indexing [67, 42, 32, 75, 39, 64]. The frequent subtree mining problem addresses the subtree isomorphism problems which is in P and simpler than the subgraph isomorphism problem that is NP complete. An uncertain graph [77, 76] is a special edge-weighted graph, where the weight on each edge (u, v) is the probability of the edge existing between vertices u and v, called the existence probability of (u, v). The support of a subgraph pattern S in an uncertain graph database D is a probability distribution over the supports of S in all implicated graph databases of D. Given an uncertain graph database D and an expected support threshold [77, 76] find all frequent subgraph patterns in D. Mining significant patterns is another variant of frequent subgraph mining where significant graphs are the one formed by transforming regions of the graphs into features and measuring the corresponding importance in terms of p-values. MbT [27] builds a model based search tree to find significant graphs where they use the divide and conquer method

to find the most significant patterns in a subspace of examples. Further, in order to provide a good understanding of the patterns of interest, it is important to reduce the redundancy in the pattern set. ORIGAMI [31] reports frequent graph patterns only if the similarity is below a threshold and for every non-reported pattern , at least one pattern can be found in for which the underlying similarity to is at least a threshold . Frequent mining problem becomes more challenging with the huge size of data graphs and the large number of graphs in a database. C. Chen et al. [16] uses randomized summarization in order to reduce the data set to a much smaller size. This summarization is then used to determine the frequent subgraph patterns from the data. Bounds are derived in C. Chen et al. [16] on the false positives and false negatives with the use of such an approach. Another challenging variation is when the frequent patterns are overlaid on a very large graph, as a result of which patterns may themselves be very large subgraphs. An algorithm called TSMiner was proposed in R. Jin et al. [40] to determine frequent structures in very large scale graphs.

The graph database can however come with certain constraints, the simplest of them being a database of graphs with unique node labels. M. Koyuturk et al. [46] finds maximal frequent subgraphs in a database of graphs with unique node labels. Their work mines metabolic pathways to discover common motifs of enzyme interactions that are related to each other. Each enzyme is represented by a unique node, independent of the number of times the enzyme appears in the underlying pathway. They hence get a graph database where each graph has unique node labels. This framework simplifies the problem considerably as it completely bypasses subgraph isomorphism. A unique node pair automatically implies a unique edge and a set of unique edges can be put together to form only one graph structure given that the node labels are unique. Hence finding the frequency of a subgraph does not require subgraph isomorphism. The authors address the problem by applying the technique as in gSpan with some additional optimizations. Since each node label is unique, the set of node labels in a graph is treated as an itemset. The paper follows a depth first approach wherein each sub-itemset is extended if the set of labels on extension form a connected subgraph of the original graph. The connectivity is maintained by only adding edges that are connected to the current subgraph and avoid redundancy by keeping track of already visited edges. A. Inokuchi et al. [38] addresses the problem in a slightly different manner where again frequent itemset mining is used for mining maximal frequent subgraphs. They do not bother looking at edge extensions by maintaining connectivity but instead treat each graph as an itemset to apply maximal frequent itemset mining and remove all disconnected graphs. We extend this work to develop an algorithm for maximal frequent subgraph mining for the class of graphs containing unique edge labels. This problem is more difficult. A frequent itemset cannot be put back together uniquely to rebuild the subgraph though the edge labels are unique. This is because of the possible multiple occurrences of a node label. Hence the algorithm would require additional information and a slightly different approach. We explore the scope of using itemset mining techniques in solving maximal frequent subgraph mining problems. We show that doing this helps substantially in reducing the run-time and the complexity involved. Itemset mining has the advantage of being less complex than subgraph mining in terms of frequency computation and candidate generation.

Frequent pattern mining was first proposed by R. Agrawal et al. [9] for market basket analysis in the form of association rule mining. The huge number of possible frequent itemsets in a large transaction database makes developing an algorithm for frequent itemset mining challenging. R. Agrawal and R. Srikant [11] observed the Apriori property for mining frequent itemsets where for an itemset to be frequent all its subitemsets have to be frequent. They use this property to efficiently find the frequent itemsets. This led to a lot of interest in other improvements or extensions of Apriori, e.g., hashing technique [51], partitioning technique [56], incremental mining [18] and parallel and distributed mining [52, 10, 17, 73].

Mining of the complete set of frequent itemsets without candidate generation was proposed by J. Han et al. [30]. They devised an FP-growth algorithm that mines long frequent patterns by finding the shorter ones recursively and then concatenating the suffx. This method reduces the search time substantially.

Equivalence CLASS Transformation (Eclat) algorithm proposed by M. J. Zaki [70] mines the frequent itemsets with vertical data format. After computing the TID-set of each single item by a single database scan, Eclat algorithm generates (k+1)-itemsets from k-itemsets generated previously using a depth first computation order similar to FP-growth [30]. M. Holsheimer et al. [33] explores the potential of solving data mining problems using the general purpose database management systems (dbms).

MaxMiner proposed by R. J. B. Jr et al. [41] studies the problem of finding maximal frequent itemsets. MaxMiner is performs level-wise bread first search to find maximal frequent itemsets using the Apriori property. It adopts superset frequency pruning and subset infrequency pruning for search space reduction. D. Burdick et al. [15] proposed MAFIA algorithm that finds maximal frequent itemsets where in frequency counting is done efficiently by the use of vertical bitmaps to compress the TID list. Theoretical analysis of the complexity of mining maximal frequent itemsets is discussed in G. Yang [68] where it is shown to be NP-hard. The length distribution of the frequent and maximal frequent itemsets is studied by G. Ramesh et al. [55].

Chapter 3

MARGIN: Maximal Frequent Subgraph Mining

In this chapter, we introduce the MARGIN algorithm. We first present the preliminary definitions and terminology that will be used throughout the MARGIN algorithm followed by a brief discussion in Chapter 3.2 on the two models in which the graph database is viewed and processed in our work. Next we give the intuition about the running of the algorithm in Chapter 3.3 with a running example. We present the MARGIN algorithm in Chapter 3.4. The MARGIN algorithm presented in Chapter 3.4 gives connected maximal frequent subgraphs. The MARGIN algorithm can be modified to give disconnected maximal frequent subgraphs which is presented in Chapter 3.5

3.1 Preliminary Concepts

In this chapter, we first provide the necessary background and notation. We adopt the same definitions as in [65] for Labeled Graph, Isomorphism and Frequent Subgraphs.

Definition 1 ***Labeled Graph****: A labeled graph can be represented as a tuple,* $G = (V, E, \Lambda, \lambda)$*, where*

V *is a set of vertices's,*

$E \subseteq V \times V$ *is a set of edges,*

Λ *is a set of labels,*

$\lambda : V \cup E \rightarrow \Lambda$*, is a function assigning labels to the vertices and edges.*

Definition 2 ***Isomorphism, Subgraph Isomorphism****[65]: An isomorphism is a bijective function* $f : V(G) \rightarrow V(G')$ *such that*

$$\forall u \in V(G), \lambda_G(u) = \lambda_{G'}(f(u))$$
$$\forall (u, v) \in E(G), (f(u), f(v)) \in E(G'), \textit{and}$$
$$\lambda_G(u, v) = \lambda_{G'}(f(u), f(v))$$

A subgraph isomorphism from G to G' is an isomorphism from G to a subgraph of G'.

For the ease of presentation, in this work we assume undirected connected labeled graphs. We denote the relationship "subgraph of" using $\subseteq_{graph}$ and "proper subgraph of" using $\subset_{graph}$. $\mathbb{D}$ is a database of n graphs $\mathbb{D} = \{G_1, G_2, .., G_n\}$.

Given a graph database $\mathbb{D}$ of n graphs and minimum support, $minSup$ let

$$\sigma(g, G) = \begin{cases} 1 & \text{if } g \text{ is isomorphic to a subgraph of } G, \\ 0 & \text{if } g \text{ is not isomorphic to any subgraph of } G \end{cases}$$

$$\sigma(g, \mathbb{D}) = \sum_{G_i \in \mathbb{D}} \sigma(g, G_i)$$

$\sigma(g, \mathbb{D})$ denotes the occurrence frequency of g in $\mathbb{D}$, *i.e.*, the support of g in $\mathbb{D}$. A frequent subgraph is a graph g, such that $\sigma(g, \mathbb{D})$ is greater than or equal to $minSup$. Let $\mathbb{F} = \{g_1, g_2, \ldots\}$ be the **set of all frequent subgraphs** of $\mathbb{D}$ for a given support $minSup$. A graph $g_i \in \mathbb{F}$ is said to be *maximal* frequent if there exists no subgraph $g_j \in \mathbb{F}$ such that, $g_i \subseteq_{graph} g_j$. Let $\mathbb{MF} \subseteq \mathbb{F}$ be the set of all maximal frequent subgraphs of $\mathbb{D}$. *Given the graph database $\mathbb{D}$ and a minimum support $minSup$, the problem of maximal frequent subgraph mining is to compute all maximal frequent subgraphs in $\mathbb{D}$.* We conceptualize the search space for finding $\mathbb{MF}$ in the form of a graph lattice.

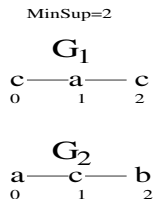

Figure 3.1 Graph Database $\mathbb{D}$=$\{G_1, G_2\}$

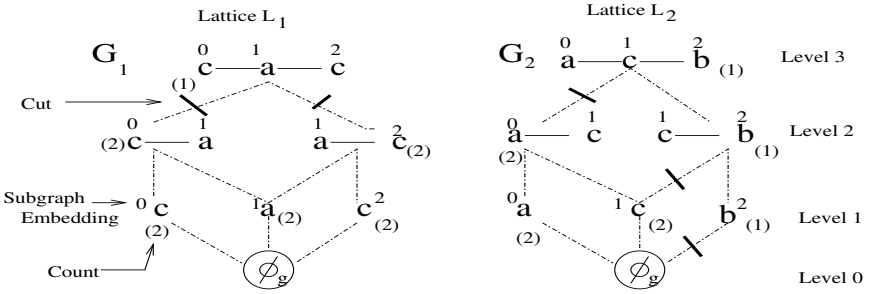

Figure 3.2 Lattice L_1,L_2

The lattice L_1 and L_2 as shown in Figure 3.2 are the graph lattices of the graphs $G_1, G_2 \in \mathbb{D}$. To keep the example simple, we assume that all the edge labels are identical and hence are not shown in the figure. Every node in the lattice is one connected subgraph of G_i. Every subgraph in G_i is represented exactly once in the lattice. In Figure 3.2, the graph $a - c$ occurs twice in the Lattice L_1 since there are two subgraphs of G_1 that are isomorphic to $a - c$. The bottom most node corresponds to the empty subgraph ϕ_g and the top most nodes correspond to G_i. A node C is a child of the node $P \neq \phi_g$ in the lattice L_i, if $P \subseteq_{graph} C$ and C and P *differ by exactly one edge*. The node P is a parent of such a node C in the lattice. For instance, for both isomorphic forms of the graph $a - c$ in L_1, the child would be the subgraph $c - a - c$ (by adding the edge $c - a$). Similarly, the subgraphs $a - c$ and $c - b$ are the parents

of $a - c - b$ in L_2. All single node subgraphs are the children of the node ϕ_g. ϕ_g is thus the parent of all the single node subgraphs and is considered to be always frequent. An edge exists in the lattice between every pair of child and parent nodes.

For a given graph G, the size of the graph (denoted by $|G|$), refers to the number of edges present in G. All the subgraphs of equal size form a $level$ in the lattice L_i of G_i. The node corresponding to ϕ_g forms level 0, singleton vertex graphs form level 1 and the nodes of size i form level $i + 1$ for $i > 0$ (Figure 3.2).

Definition 3 ***Cut:*** *A cut between two nodes in a lattice represented by $(C \dagger P)$ is defined as an ordered pair (C, P) where P is the parent of $C \in L_i$ and C is not frequent while P is frequent. The frequent subgraph P of a cut is represented by $f(\dagger)$ (frequent-†) and the infrequent subgraph C is represented by $I(\dagger)$ (infrequent-†). The symbol † is read as 'cut'.*

Note that different isomorphic subgraphs of a graph g in the Lattice L_i will thus have the same support in $\mathbb{D}$. However the subgraphs corresponding to the children of each isomorphic form might be different. Also while one isomorphic form of a subgraph might become a $f(\dagger)$-$node$, the other might not.

Example: Consider Figure 3.2 with $minSup = 2$. The node in L_2 that corresponds to the subgraph c is a $f(\dagger)$-$node$ since it is frequent with count 2. Its parent node that corresponds to $c - b$ is infrequent with count 1 and thus is an $I(\dagger)$-$node$. Hence, this pair is marked as cut. Figure 3.2 shows the frequency count of each node in the example lattice along with all the existing cuts in the lattice L_1 and L_2 respectively.

3.2 Lattice Representation Models

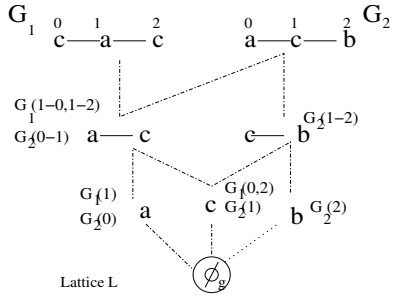

Figure 3.3 Embedded Model

The lattice structure to the MARGIN algorithm can be represented using two models, namely the replication model and the embedded model.

In the replication model (Figure 3.2), each subgraph $g \subset_{graph} G_i \in \mathbb{D}$ is represented exactly once in the lattice L_i of G_i as described in Chapter 3.1.

The embedded model is shown in Figure 3.3 for the graphs in Figure 3.1. In an embedded model, the lattice $\mathbb{L}$ of the database $\mathbb{D}$ is common for all $G_i \in \mathbb{D}$. All isomorphic forms of any subgraph in $\mathbb{D}$ are represented using a single node in the embedded model. Each node in the lattice L stores the information about the occurrences of all of its isomorphic forms in the database. The bottommost node corresponds to the empty graph ϕ_g and the topmost nodes correspond to the graphs in $\mathbb{D}$. The graph $a-c$ in Figure 3.1 has three isomorphic forms in the database but will be represented exactly once along with the information about its occurrence in G_1 at location $1-0$ and $1-2$ and its occurrence in G_2 at location $0-1$.

In the rest of the work, we proceed using the replication model unless mentioned. With minor modifications the MARGIN algorithm can be applied to the embedded model.

3.3 Intuition

We start by defining the $Upper$-$\diamond$-$Property$ that holds in every lattice L_i of $G_i \in \mathbb{D}$ which we exploit in our algorithm.

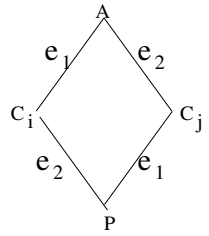

Figure 3.4 $Upper$-$\diamond$-$Property$

Property 1 ***Upper Diamond property*** ***(*$Upper$*-$\diamond$-property):*** *Any two children C_i, C_j of a node P, where $C_i, C_j \in$ Lattice L_k of G_k for $G_k \in \mathbb{D}$, will have a common child A.*

Proof: Let e_1 and e_2 be the edges incident on the vertices n_1 and n_2 in P respectively. Let $P \cup \{e_1\} = C_j$ and $P \cup \{e_2\} = C_i$ as in Figure 3.4. Hence e_1 would be incident on n_1 in C_i and e_2 would be incident on n_2 in C_j. Let $A = C_j \cup \{e_2\}$.
Hence, $A = (P \cup \{e_1\}) \cup \{e_2\} = (P \cup \{e_2\}) \cup \{e_1\} = C_i \cup \{e_1\}$.
Hence, $A = C_i \cup \{e_1\} = C_j \cup \{e_2\}$ is the common child of C_i and C_j proving the $Upper$-$\diamond$-property. □

Consider any two children A_1 and A_2 of a node B in a lattice. In a lattice of the replication model, by definition, A_1 and A_2 are obtained by extending B. However, in the embedded model, any two children of a node need not be the extensions of the same isomorphic form. Hence, in the embedded model, the $Upper$-$\diamond$-property holds similarly except that A_1 and A_2 should be the extensions of the same isomorphic form of the subgraph B in order to have a common child C of A_1 and A_2.

Next, we provide an intuition to the MARGIN algorithm. The set of candidate subgraphs that are likely to become maximally frequent are the $f(\dagger)$ nodes. This is because they are frequent subgraphs having an infrequent child. The remaining nodes of the lattice cannot be maximal frequent subgraphs. In this work, we present an approach that avoids traversing the lattice bottom up and instead traverses the cuts alone in each lattice L_i for $G_i \in \mathbb{D}$. We prune the set of $f(\dagger)$ nodes to give the set of maximal frequent subgraphs. MARGIN algorithm unlike the Apriori based algorithms goes directly to any one of the $f(\dagger)$ nodes of the lattice L_i and then finds all other $f(\dagger)$ nodes by cutting across the lattice L_i. We give an insight below into the approach developed.

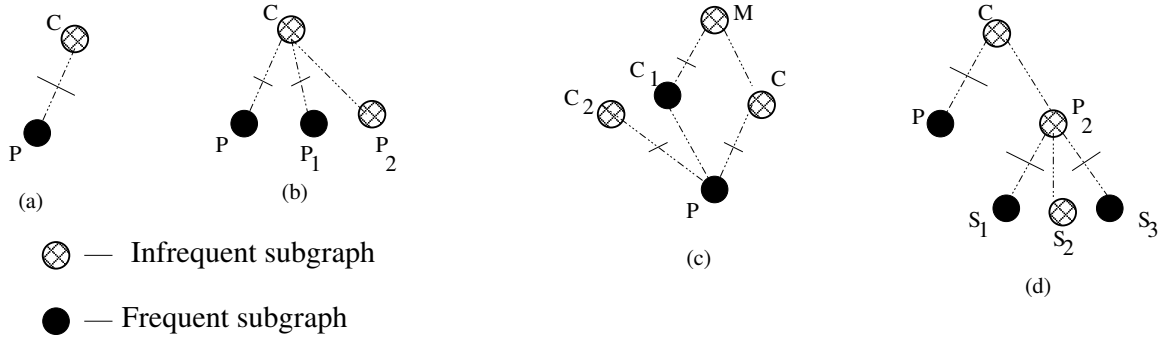

Figure 3.5 The Steps in $ExpandCut$

Finding the initial $f(\dagger)$ node is a trivial dropping of edges one by one from the initial graph $G_1 \in \mathbb{D}$, ensuring that the resulting subgraph is connected until we find the first frequent subgraph R_i. We call the frequent subgraph found by such dropping of edges as the $Representative$ R_i of G_i. Our initial cut is thus $(CR_i \dagger R_i)$ where CR_i is the infrequent child of R_i. Figure 1.1 shows the infrequent nodes visited in the infrequent space of the lattice in order to determine the $Representative$ that lies on the border. Thus, strictly speaking, in Figure 1.1, MARGIN explores the space marked as "Margin Search Space" and the set of infrequent nodes to find the $Representative$ R_i of G_i. Similarly, the $Representative$ can also be found by starting with the empty graph ϕ_g (assumed to be frequent) and extending it until an infrequent subgraph is found. For lower support values, as the candidate nodes are expected to lie higher up in the lattice (as discussed later), we find the $Representative$ starting at the topmost node of the lattice. On the other hand, for high support values, as the candidate nodes lie in the lower levels of the lattice, starting at ϕ_g incurs less computational cost.

We devise an algorithm $ExpandCut$ which for each cut c discovered in $G_i \in \mathbb{D}$, recursively extends the cut to generate all cuts in G_i. We provide an intuition to the $ExpandCut$ algorithm used to find the nearby cuts given any cut $(C \dagger P)$ as input in the lattice L_i of G_i. Recursively invoking $ExpandCut$ on each newly found cut with the below three steps, finds all cuts in G_i. Figure 3.5 is through the explanation of the $ExpandCut$ algorithm. Consider the initial cut to be the cut $(C \dagger P)$ in Figure 3.5(a).
Step1: The node C in lattice L_i can have many parents that are frequent or infrequent, one of which is P. Consider the frequent parent P_1 in Figure 3.5(b). Cut $(C \dagger P_1)$ exists since P_1 is frequent while C is infrequent. Thus, for an initial cut $(C \dagger P)$, all frequent parents of C are reported as $f(\dagger)$ nodes.

Step2: Let P be the parent of C as in Figure 3.5(c). Let C_1, C_2 and C be the children of the frequent node P. Each of C_1, C_2 and C can be frequent or infrequent.
(a): Consider an infrequent child C_2. Cut (C_2 † P) exists since P is frequent while C_2 is infrequent. Thus, for an initial cut (C † P), for each frequent parent P_f of C that has an infrequent child C_i, the cut (C_i † P_f) is reported.
(b): Consider a frequent child C_1. By $Upper\text{-}\diamond\text{-}Property$, the nodes C and C_1 have a common child M. M is infrequent as its parent C is infrequent. Hence, cut (M † C_1) exists. Thus, for an initial cut (C † P), for each frequent parent P_f of C consider each of its frequent child C_i. The cut (M † C_i) is reported where M is the common child of C_i and C.
Step3: Consider all parents S_1, S_2, S_3 of an infrequent parent P_2 of C as in Figure 3.5(d). Each such parent can be frequent or infrequent. Consider frequent parents S_1, S_3 (Figure 3.5(d)) of an infrequent parent P_2 of C. Hence, the cuts (P_2 † S_1) and (P_2 † S_3). However, if step 1 is called on the cut (P_2 † S_1), the cut (P_2 † S_3) is found. Thus, for an initial cut (C † P), for each infrequent parent P_i of C, consider any one frequent parent S_f of P_i. The cut (P_i † S_f) is invoked.

Example: Consider the lattice L_2 of Figure 3.2. Let the cut ($c - b$ † c) be the initial cut. The node representing the subgraph $c - b$ has two parents, namely the subgraphs c and b of which the subgraph c is frequent and b is infrequent.
Step1: Among the parents of $c - b$, as the frequent parent is c which is the initial cut and there is no other frequent parent, we go to step2.
Step2: The subgraph c has one child $a - c$ apart from the child $c - b$ with which it formed the initial cut. Since $a - c$ is frequent, $ExpandCut$ goes to step2(b). In step2(b), the subgraphs $a - c$ and $c - b$ will have a common child $a - c - b$ by the $Upper\text{-}\diamond\text{-}Property$. As $a - c - b$ will be infrequent and $a - c$ is found to be frequent, the cut ($a - c - b$ † $a - c$) is found.
Step3: The infrequent parent b of $c - b$ is next considered and its frequent parents are found. As ϕ is a frequent subgraph, the cut (b † ϕ) is reported.
All the cuts found by the first iteration of the $ExpandCut$ function are reported and recursively called. In our example, a single call of the $ExpandCut$ technique on the initial cut has found all the cuts of the lattice. □

The detailed algorithm $ExpandCut$ is given in Chapter 3.4.

3.4 The MARGIN Algorithm

Table 3.1 shows the $MARGIN$ algorithm to find the globally maximal frequent subgraphs $\mathbb{MF}$. Initially, $\mathbb{MF} = \emptyset$ (line 1) and the graphs in $\mathbb{D}$ are unexplored. $\mathbb{LF}$ is the set of locally maximum subgraphs in each G_i which is initially ϕ (line 3). Initially, given the graphs $\mathbb{D} = \{G_1, G_2, \ldots, G_n\}$, for each $G_i \in \mathbb{D}$, we find the representative R_i for G_i (line 3). This is done by iteratively dropping an edge from G_i until a connected frequent subgraph is found. The $ExpandCut$ algorithm is initially

Input: Graph Database $\mathbb{D} = \{G_1, G_2, \ldots, G_n\}$,
Output: Set of Maximal Frequent Graphs $\mathbb{MF}$
Algorithm:
1 $\mathbb{MF}$=$\emptyset$
2. For each $G_i \in \mathbb{D}$ do
3. $\mathbb{LF}$=ϕ
4. Find the representative R_i of G_i
5. $ExpandCut(\mathbb{LF}, CR_i \dagger R_i)$ where CR_i is the infrequent child of R_i
6. Merge($\mathbb{MF}$,$\mathbb{LF}$)

Table 3.1 Algorithm Margin

invoked on the cut ($CR_i \dagger R_i$) with LF=ϕ where CR_i is the infrequent child of R_i. $ExpandCut$ finds the nearby cuts and recursively calls itself on each newly found cut. The algorithm functions in a manner that finding one cut in $G_i \in \mathbb{D}$ would find all cuts in G_i. In line 6, the globally maximal frequent subgraphs $\mathbb{MF}$ found so far are merged with the local maximal frequent subgraphs $\mathbb{LF}$ found in G_i. The merge function finds the new globally maximal set by removing all subgraphs that are not maximal due to the subgraphs in $\mathbb{LF}$ and adds subgraphs from $\mathbb{LF}$ that are globally maximally frequent after exploring G_i.

Table 3.2 shows the $ExpandCut$ algorithm which expands a given cut such that its neighboring cuts will be explored. The input to the algorithm are the set of maximal frequent subgraphs $\mathbb{LF}$ found so far (initially empty), and the cut $(C \dagger P)$.
For each parent Y_i of C, if Y_i is frequent, Y_i is added to $\mathbb{LF}$ (lines 3-4).
• If child(Y_i) be the set of children of Y_i, then for each infrequent child K in child(Y_i), $ExpandCut$ is called on the cut ($K \dagger Y_i$) (line 7-8).
• For each frequent child K in child(Y_i), let M be the common child of C and K. $ExpandCut$ is called on the cut ($M \dagger K$) (line 9-11).

On the other hand, if Y_i is infrequent and there exists at least one frequent parent K among the parents of Y_i, then, $ExpandCut$ is called on the cut ($Y_i \dagger K$).(lines 12-15).

3.5 MARGIN-d: Finding disconnected maximal frequent subgraphs

Finding disconnected maximal frequent subgraphs is more complex than the problem of finding connected maximal frequent subgraphs. The search space of Apriori like algorithms would increase as compared to its connected maximal frequent subgraph finding counterparts since every combination of connected subgraphs forms a valid node in the graph lattice of a graph in the database. For academic interest, we discuss in this section the application of MARGIN to disconnected maximal finding followed by an evaluation of results. Given a subgraph g and a supergraph G_{sup}, a subgraph isomorphism

Input:
$\mathbb{LF}$: The maximal frequent subgraphs seen so far in G_i.
Cut:$C \dagger P$

Output: The updated set of maximal frequent subgraphs $\mathbb{LF}$.

Algorithm:

```
1.  Let Y_1, Y_2, ..., Y_c be the parents of C.
2.  for each Y_i, i = 1, ..., c do
3.      if Y_i is frequent
4.          LF = LF ∪ Y_i
5.          Let child(Y_i) be the set of children of Y_i
6.          for each element K in child(Y_i) do
7.              if K is infrequent do
8.                  ExpandCut(LF,K † Y_i)
9.              if K is frequent do
10.                 Find common child M of C and K
11.                 ExpandCut(LF,M † K)
12.     if Y_i is infrequent
13.         Let parent(Y_i) be the set of parents of Y_i
14.         if one frequent parent K in parent(Y_i) exists
15.                 ExpandCut(LF,Y_i † K )
```

Table 3.2 Algorithm ExpandCut($\mathbb{LF}$,$C \dagger P$)

searches for an occurrence of g in G_{sup} which is determined by mapping each node in g to a node in G_{sup} such that the corresponding edges between the nodes in g map to the edges incident on the mapped nodes of G_{sup}. In the worst case, the set of nodes in g are attempted to be mapped to every subset of nodes in G_{sup} looking for a possible match. It is sufficient to find a single successful match to determine subgraph isomorphism.

This problem becomes more complex when the subgraph g is disconnected. Each component of g needs to be mapped to a corresponding subgraph in G_{sup} such that the mapped subgraphs of G_{sup} are non-overlapping. Each component of g can be mapped to multiple matchings in G_{sup}. Here it is not sufficient to find a successful match for each component of g but to ensure that this set of successful matches are non-overlapping. This might require exploring all possible combinations of the matching across the components.

Further, for connected subgraphs g the number of children or immediate supergraphs of $g \subset G_{sup}$ is the total number of edges incident on nodes of g that are not already contained in g. Hence g is guaranteed at least one child. For disconnected subgraphs, the number of children of g is the total number of edges in the graph G_{sup} that are not contained in g. This number is the maximum number of children in the case of connected maximal frequent subgraphs mining(in case of cliques). Similarly the number of parents of g in the connected subgraph version is the number of edges that can be dropped by keeping the resultant graph connected which is a minimum of two for graphs with greater than two edges. For the disconnected version the number of parents is the number of edges in g which is again

the maximum for connected versions. This change in number of parents and children generated is significant in case of an Apriori approach where the children of each explored node in the bottom-up traversal needs to be found.

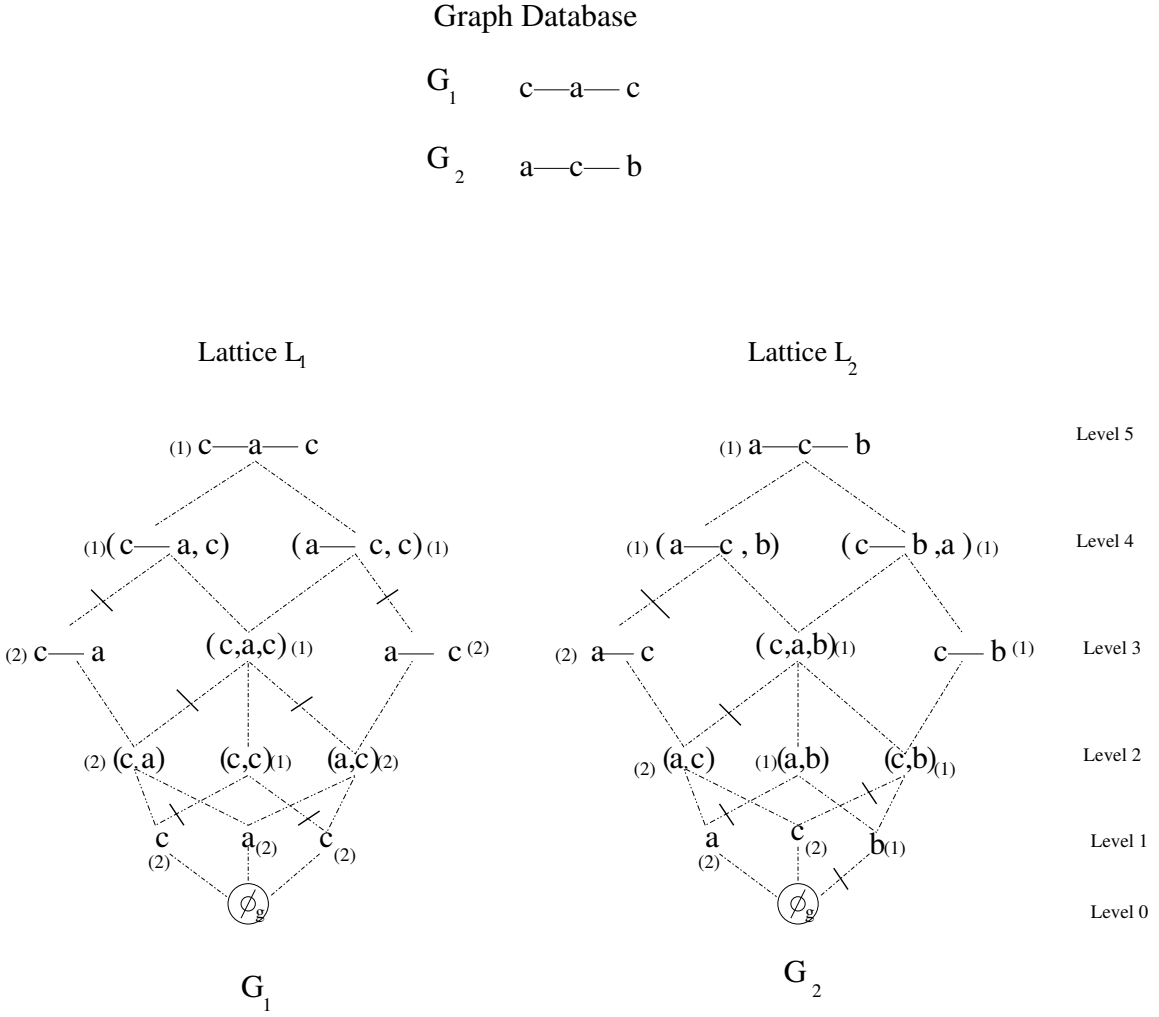

Figure 3.6 Graph Lattice for Disconnected Graphs

The algorithm for mining maximal frequent disconnected subgraphs is same as that for connected subgraphs except for parent-child definitions. Margin dealt with only connected parent and child subgraphs while the disconnected version of Margin which we refer to as Margin-d permits parent and children to be disconnected. It is trivial to fit the algorithm to directed graphs and unlabeled graphs. The lattice is modified in case of disconnected graphs as represented in Figure 3.6. The lattice is drawn for the graph database shown in the figure and was seen earlier for the connected subgraph lattice in Figure 3.2. Each level i represents subgraphs whose sum of edges and nodes is i. The numbers given in brackets show the frequency of each subgraph in the database. The $Upper$-$\diamond$ property holds as for the case of the connected graph lattice. For a support value of two, the $ExpandCut$ function as described

earlier will find the cuts as shown in the Figure 3.6. In section 4.5 we validate experimentally why it should do better than the disconnected subgraph extensions of Apriori-based algorithms.

3.6 Proof of Correctness

In this section, we prove the correctness of MARGIN algorithm. We first prove the essential claim that finding one cut in $G_i \in \mathbb{D}$ leads to all cuts in $G_i \in \mathbb{D}$. The cuts in $\mathbb{D}$ are finally pruned to compute the maximal frequent subgraphs. Note that ϕ_g is always considered frequent as ϕ_g is a subgraph of every $G_i \in \mathbb{D}$.

Claim 1 *Given two cuts $(C_1 \dagger P_1)$ and $(C_2 \dagger P_2)$ in $G_i \in D$, invoking $ExpandCut$ on one cut finds the other cut.*

Proof: We shall use the term *"node"* for the nodes of the lattice and *"lattice edge"* for an edge in the lattice. Similarly, the term *"vertex"* is used to represent the nodes of the graph G_i and *"graph edge"* to represent an edge in the graph G_i. The information available to us for the proof are:

- The given two cuts for which reachability has to be proved are $(C_1 \dagger P_1)$ and $(C_2 \dagger P_2)$.
- $C_j = (V_j, E_j, \Lambda, \lambda_j)$ for $j = 1, 2$ such that $C_j \subset_{graph} G_i$.
- If the subgraphs C_1 and C_2 have no graph edge in common, then we find the minimal number of graph edges in G_i joining a vertex of subgraph C_1 with a vertex of the subgraph C_2. Hence, such a sequence of graph edges forms a *connected subgraph of G_i referred as $Path_P = (V_p, E_p, \Lambda, \lambda_p)$*.

We consider two cases namely :

1. Case 1: $P_1 \cap P_2 \neq \emptyset$ which implies that there exists a common subgraph of P_1 and P_2.
2. Case 2: $P_1 \cap P_2 = \emptyset$ which implies that there does not exist a common subgraph of P_1 and P_2. Hence the common parent node in the lattice for P_1 and P_2 is the bottom-most node ϕ of the lattice.

Starting at one cut, the sequence of cuts that finds the other cut can be shown in a sub-lattice L_i^{sub} of L_i. We next describe the sub-lattice L_i^{sub} which is contained in lattice L_i of G_i for each of two cases above.

Case1: If $P_1 \cap P_2 \neq \emptyset$, consider the largest common subgraph t of P_1 and P_2 in lattice L_i (Figure 3.7). The node t is the lower most node of the lattice L_i^{sub} and hence is at level 0 of L_i^{sub}. Note that the level 0 of L_i^{sub} occurs at the level $|t| + 1$ of lattice L_i since, as mentioned earlier, in lattice L_i, the level of a node is one added to the size of the node. For two graphs G_i and G_j, the set of graph edges present in G_i and not present in G_j are represented by $G_i \setminus G_j$. Extend t to g_1 by adding a graph

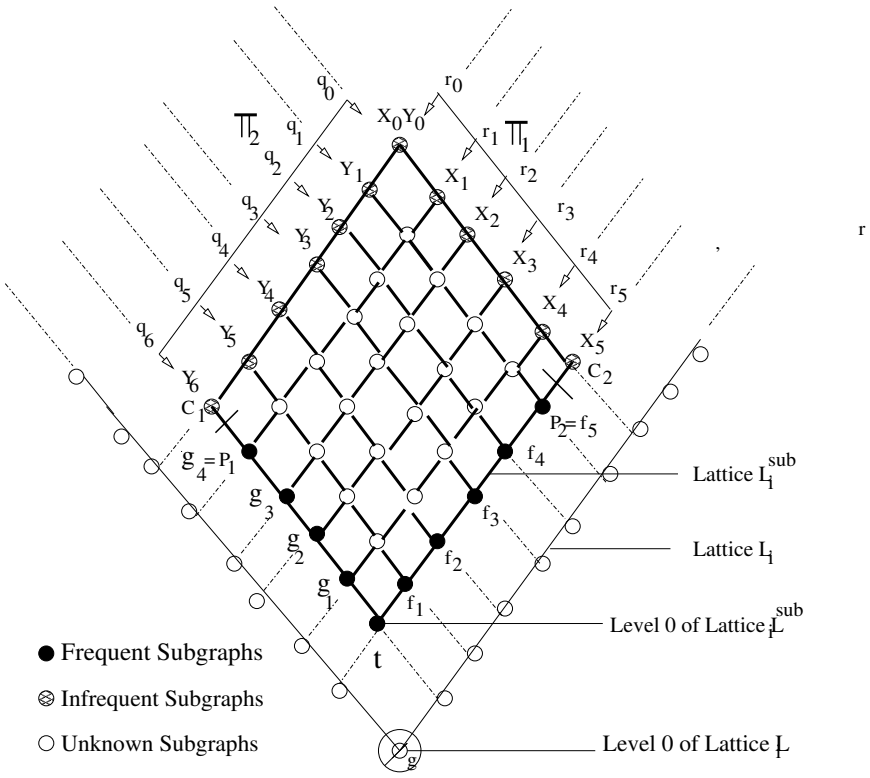

Figure 3.7 Case1: Lattice $\mathbf{L_i^{sub}}$: $\mathbf{P_1 \cap P_2 \neq \emptyset}$

edge $e_{g_1} \in (P_1 \setminus t)$ incident on t. Extend g_i for $i > 0$ in lattice L_i^{sub} to g_{i+1}, by adding a graph edge $e_{g_{i+1}} \in (P_1 \setminus g_i)$ incident on g_i until $g_{i+1} = P_1$. Extend P_1 in the lattice L_i^{sub} to C_1. Similarly, extend t to f_1 by adding a graph edge $e_{f_1} \in (P_2 \setminus t)$ incident on t. Extend f_i to f_{i+1} for $i > 0$ by adding a graph edge $e_{f_{i+1}} \in (P_2 \setminus f_i)$ incident on f_i until $f_{i+1} = P_2$. Extend P_2 to C_2. For every two nodes in level $i > 0$ of L_i^{sub} which has a common parent in L_i^{sub}, construct the common child in L_i^{sub}. Such a common child always exists by the $Upper$-$\diamond$ property.

Case2: If $P_1 \cap P_2 = \emptyset$, consider the shortest path $Path_P = (V_p, E_p, \Lambda, \lambda_p)$ that connects P_1 and P_2 in G_i. Let $\{m_1, \ldots, m_{|E_p|}\}$ be the sequence of graph edges that constitute the shortest path where graph edge m_i is adjacent to graph edge m_{i+1} for $i = 1, \ldots, |E_p| - 1$. m_1 is incident on a vertex in P_1 and $m_{|E_p|}$ is incident on a vertex in P_2 (Figure 3.9). Let subgraph $g_1 = m_1 \cup e_{g_1}$ where e_{g_1} is the graph edge of P_1 adjacent to m_1. Similarly let $f_1 = m_{|E_p|} \cup e_{f_1}$ where e_{f_1} is the graph edge of P_2 incident on $m_{|E_p|}$. The lower most node of the lattice L_i^{sub} is ϕ_g at level 0 (Figure 3.8). Level 1 of the lattice(L_i^{sub}) consists of the set of vertices V_p of the path $Path_p$, which are the children of the node ϕ_g. The set of graph edges $E_p \cup e_{g_1} \cup e_{f_1}$ forms the nodes of the level 2 of the lattice L_i^{sub}. If $C_1 \not\subseteq_{graph} g_1$, extend g_i to g_{i+1} by taking a graph edge $e_{g_{i+1}} \in P_1 \setminus g_i$ incident on g_i for $i > 0$ until $g_{i+1} = P_1$. Then, extend P_1 to C_1. Similarly, if $C_2 \not\subseteq_{graph} f_1$, extend f_i to f_{i+1} by taking a graph edge $e_{f_{i+1}} \in P_2 \setminus f_i$ that is incident on f_i to form a subgraph f_{i+1} in the lattice L_i^{sub} and then extend P_2 to C_2. For every two

nodes in level $i > 1$ which have a common parent in the lattice L_i^{sub}, construct the common child. If the common child is one of the g_i's or f_i's then draw the required connecting lattice edges to make it the common child in the lattice.

Hence above we have shown the construction of L_i^{sub} for both cases which exists as a sub-lattice of the lattice L_i of G_i. We next mention the terms that we shall use to denote specific nodes of the sub lattice L_i^{sub} as depicted in Figure 3.7.

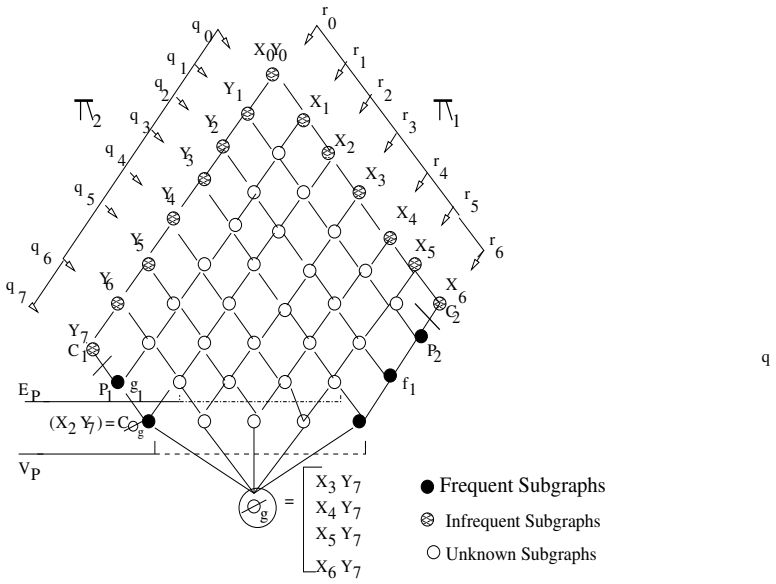

Figure 3.8 Case2: Lattice $\mathbf{L_i^{sub}}$: $\mathbf{P_1 \cap P_2 = \emptyset}$

1. Let $\mathbb{Y} = Y_0, \ldots, Y_n$ be the ordered sequence of super graphs of C_1 including C_1 in lattice L_i^{sub} such that $|Y_0| > |Y_1| > \ldots > |Y_n|$.

2. Let $\mathbb{X} = X_0, \ldots, X_m$ be the ordered sequence of super graphs of C_2 including C_2 in lattice L_i^{sub} such that $|X_0| > |X_1| > \ldots > |X_m|$ as illustrated in Figure 3.7.

3. We define the set of paths $\Pi_1 = r_0, \ldots, r_m$ (shown in Figure 3.7) where each $r_i \in \Pi_1$ is the union of the lattice edges of form $((X_i, Y_j), (X_i, Y_{j+1}))$ for $j = 0, \ldots, n-1$.

4. Similarly $\Pi_2 = q_0, \ldots, q_n$ where each $q_j \in \Pi_2$ is the union of lattice edges of form $((X_i, Y_j),(X_{i+1}, Y_j))$ for $i = 0, \ldots, m-1$.

The sub-lattice L_i^{sub} has the following properties:

- All super graphs of C_1 and C_2 in lattice L_i^{sub} are infrequent.
- All subgraphs of P_1 and P_2 are frequent.

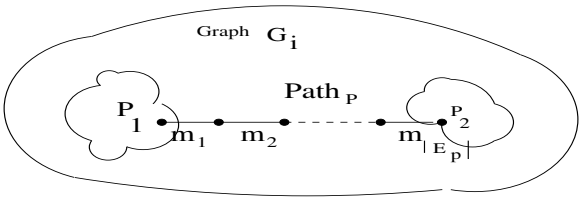

Figure 3.9 Path $Path_P = (V_p, E_p, \Lambda, \lambda_p)$

- Any node $N \neq \phi_g$ in the lattice L_i^{sub} can be now referred to as a unique tuple (X_i, Y_j) such that X_i is the smallest super graph of N in $\mathbb{X}$ and Y_j is the smallest super graph of N in $\mathbb{Y}$.
- (X_i, Y_0) is infrequent while (X_i, Y_n) is frequent for $\forall i > 0$. Hence, for each $r_i \in \Pi_1$, $\forall\ i > 0$, there exists exactly one cut.
- Similarly, (X_0, Y_i) is infrequent while (X_m, Y_i) is frequent for $\forall i > 0$. Hence, there lies one cut on each $q_j \in \Pi_2$, $\forall j > 0$.
- Thus, there are $m + n$ cuts in the lattice L_i^{sub}.
- Cut $(C_1 \dagger P_1)$=cut $((X_0, Y_n) \dagger (X_1, Y_n))$ by the construction of the lattice L_i^{sub} and lies on the path q_n.
- The final cut $(C_2 \dagger P_2)$= cut $((X_m, Y_0) \dagger (X_m, Y_1))$ to be reached lies on the path r_m.

Next, given the initial cut $(C_1 \dagger P_1)$, we are to show that cut $(C_2 \dagger P_2)$ is reachable. We give the proof for case 1 which uses Figure 3.7. The proof also holds for case 2 and is illustrated through the running example in subsection 3.6.1. To conclude the proof we state and prove the reachability sequence property below.

The Reachability Sequence:
The order of paths $k_1, \ldots, k_{m+n}$ where $k_i \in \Pi_1$ or Π_2 on which the cuts from path q_n to r_m are found, satisfy the following:

1. for $k_j = r_s \in \Pi_1, r_{s+1} = k_l, l > j$
2. for $k_j = q_s \in \Pi_2, q_{s-1} = k_l, l > j$

Proof: Since we traverse the paths from q_n to r_m (Figure 3.10), the initial path $k_1 = q_n$ and the final path $k_{m+n} = r_m$. We prove the above claim by induction.
Base Case: To prove that the claim holds for k_1. The cut on the path $k_1 = q_n$ corresponds to the initial cut $(C_1 \dagger P_1)$. The child(X_0, Y_{n-1}) of C_1 is infrequent. Consider the node $N = (X_1, Y_{n-1})$. If node N is frequent, cut $((X_0, Y_{n-1}) \dagger N)$ on the path q_{n-1} is found by lines(8-10) of the $ExpandCut$ algorithm. If node N is infrequent, cut $(N \dagger P_1)$ on the path r_1 is found by lines(6-7) of the $ExpandCut$ algorithm. Hence the next cut is found either on path q_{n-1} or path r_1. Hence, conditions 1 and 2 of the claim are

satisfied.

Induction Step: Assuming that the claim holds for $k_1, \ldots, k_s$, $s < m + n$, we prove that the claim holds for k_{s+1}. Let $r_1, r_2, \ldots, r_a$ and $q_n, q_{n-1}, \ldots, q_b$ be the set of paths whose cuts have been found. Hence, the shaded area in Figure 3.10 is thus eliminated while finding further cuts.

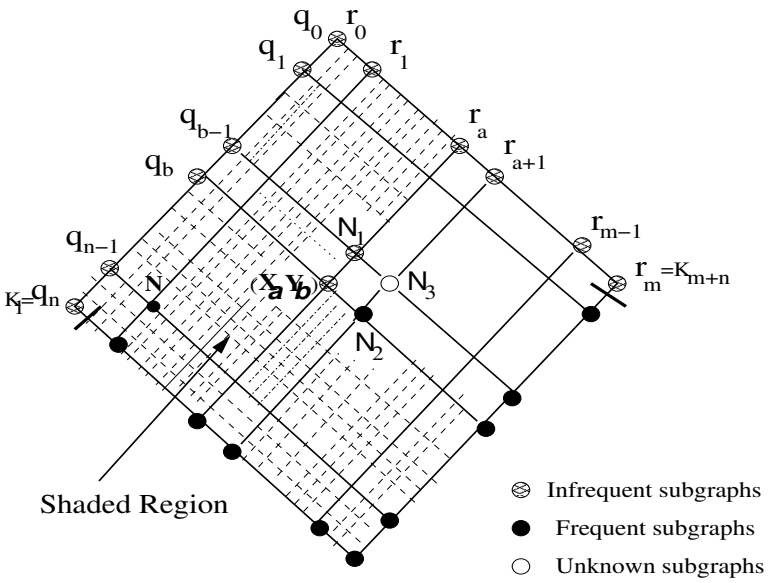

Figure 3.10 The Reachability Sequence between cuts.

Suppose $k_s = q_b$. For a cut on the path q_i, if the cuts on the paths $r_0, \ldots, r_a$ have already been seen, then the cut on the path q_i is of form cut $((X_a, Y_i) \dagger (X_{a+1}, Y_i))$. Hence, the cut on the path q_b is cut $((X_a, Y_b) \dagger (X_{a+1}, Y_b))$. Hence the node $N_1 = (X_a, Y_{b-1})$ is infrequent while the node $N_2 = (X_{a+1}, Y_b)$ is frequent as shown in the Figure 3.10. Consider the node $N_3 = (X_{a+1}, Y_{b-1})$ which is the parent of N_1 and the child of N_2. If N_3 is frequent, cut $(N_1 \dagger N_3)$ on the path q_{b-1} is found by lines(3-4) of the $ExpandCut$ algorithm. On the other hand, if N_3 is infrequent, cut $(N_3 \dagger N_2)$ on the path r_{a+1} is found by lines(6-7).

Suppose $k_s = r_a$. For a cut on the path r_i, if the cuts on the paths $q_n, q_{n-1}, \ldots, q_b$ have been seen, then the cut on r_i is of form cut $((X_i, Y_{b-1}) \dagger (X_i, Y_b))$. Hence, the cut seen on the path r_a is cut $((X_a, Y_{b-1}) \dagger (X_a, Y_b))$. Arguing as in the case of $k_t = q_b$, where $N_1 = (X_a, Y_{b-1})$ is infrequent while the node $N_2 = (X_{a+1}, Y_b)$ is frequent, it can be shown that if N_3 is frequent, cut $(N_1 \dagger N_3)$ on the path q_{b-1} is found by steps(3-4) of the $ExpandCut$ algorithm. On the other hand, if N_3 is infrequent, cut $(N_3 \dagger N_2)$ on the path r_{a+1} is found by lines(11-13).

The next cut is found on either the path q_{b-1} or the path r_{a+1}. Thus, the conditions 1 and 2 of the claim holds for k_{s+1}. Since the claim is true for k_1 and we showed that the claim holds for any k_{s+1} when k_s is true, by induction the claim holds for all $s = 1, \ldots, m + n$. □

By the reachability sequence, starting at the initial cut ($C_1 \dagger P_1$) on path q_n, a cut ($C_2 \dagger P_2$) on path r_m has been reached. Hence, starting at an initial cut any cut in a graph $G_i \in \mathbb{D}$ can be reached. Hence, all cuts in G_i can be reached from an initial cut. □

Corollary 1 *MARGIN algorithm finds all maximal frequent subgraphs in* $\mathbb{D}$.

MARGIN algorithm initially finds one cut in each $G_i \in \mathbb{D}$. By claim 1, all cuts in G_i are found. Hence all cuts in lattice $L_i \; \forall i = 1, \ldots, n$ have been found. The merge operation returns the maximal frequent subgraphs among the $f(\dagger)$ nodes reported.

,

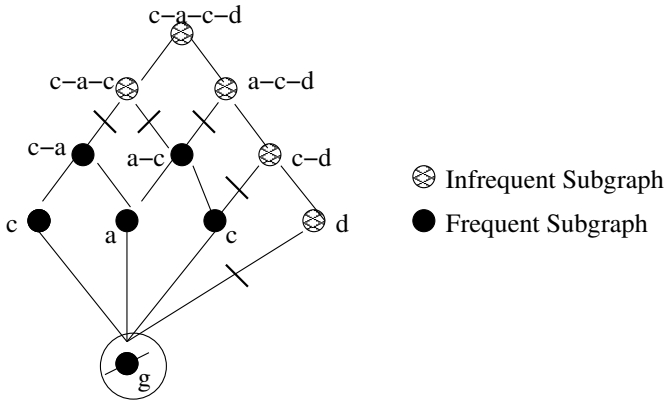

Figure 3.11 Example Lattice.

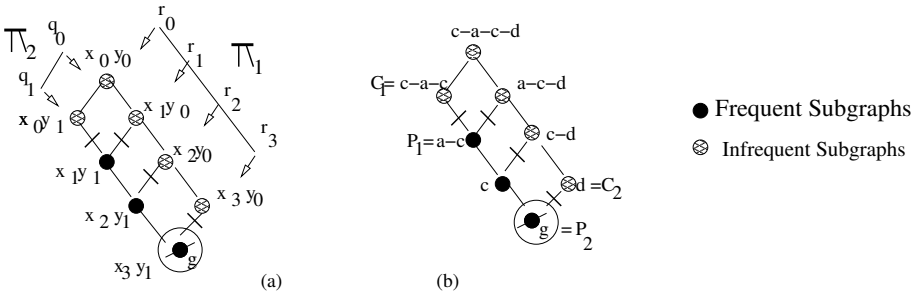

Figure 3.12 Constructed Lattice.

3.6.1 Running Example for the proof

Let the graph $c - a - c - d$ be a $G_i \in \mathbb{D}$. Consider the graph lattice of $c - a - c - d$ given in Figure 3.11. Assume that the infrequent and frequent nodes in the lattice are as marked in Figure 3.11. The resulting cuts are indicated in the figure. Say, we would like to reach the cut $(d \dagger \phi)$ from the cut $(c - a - c \dagger c - a)$. Hence $C_1 = c - a - c$, $C_2 - d$, $P_1 = c - a$ and $P_2 = \phi$. As $P_1 \cap P_2 = \emptyset$, we shall construct the lattice as in Case2 above. In the graph $G_i = c - a - c - d$, the shortest path connecting C_1 and C_2 is $c - d$. Hence m_1=m_{E_p}=$c - d$ which is incident on C_1 and C_2. The lower most level of the lattice being constructed is ϕ_g at level 0. Level 1 of the lattice(L_i^{sub}) consists of the set of vertices $V_p = c, d$ as shown in Fig 3.12(b). The set of graph edges $E_p \cup e_{g_1} \cup e_{f_1}$ =$c - d \cup a - c \cup$ where e_{g_1} is the graph edge of P_1 adjacent to m_1 in G_i and forms the nodes of the level 2 of the lattice L_i^{sub}. We next construct g_1 and f_1 as per Case 2. Subgraph $g_1 = m_1 \cup a - c = a - c - d$. As $a - c - d \subseteq_{graph} a - c - d$, we do not extend g_1 further. Similarly, as $d \subseteq_{graph} c - d$, we do not extend f_1 any further. For every two nodes in level $i > 1$ which have a common parent in the lattice, construct the common child. If the common child already exists in the next level then draw the connecting lattice edges to the common child. This results in the lattice shown in Figure 3.12(b).

All super graphs of $C_1 = c - a - c$ and $C_2 = d$ in lattice L_i^{sub} will be infrequent while all subgraphs of $P_1 = a - c$ and $P_2 = \phi$ will be frequent. Let $\mathbb{Y} = Y_0, \ldots, Y_n$ be the ordered sequence of super graphs of $c - a - c$ including $c - a - c$ in lattice L_i^{sub} such that $|Y_0| > |Y_1| > \ldots > |Y_n|$. Hence in our example $n = 1$. Let $\mathbb{X} = X_0, \ldots, X_m$ be the ordered sequence of super graphs of d including d in lattice L_i^{sub} such that $|X_0| > |X_1| > \ldots > |X_m|$. Hence $m = 3$. Every node $N \neq \phi_g$ in the lattice L_i^{sub} is now referred to as a unique tuple (X_i, Y_j). The nodes of the example we picked, are represented as in Figure 3.12(a) where vertex c of Figure 3.12(b) corresponds to X_2Y_1 of 3.12(b), vertex d of Figure 3.12(b) corresponds to X_3Y_0 of 3.12(a) and so on. C_2=(X_m, Y_0). Further, the set of paths Π_1 and Π_2 are represented in the figure.

Notice that for any path $r_i \in \Pi_1$, (X_i, Y_0) is infrequent while (X_i, Y_n) is frequent for $\forall i > 0$. Hence, there has to be exactly one lattice edge that is incident on one frequent and one infrequent node. Hence, for each $r_j \in \Pi_1$, for $j > 0$, there exists exactly one cut. Arguing similarly, (X_0, Y_i) is infrequent while (X_m, Y_i) is frequent for $\forall i > 0$. Hence, there lies one cut on each $q_j \in \Pi_2$, $\forall j > 0$. Thus, there are $m + n = 3 + 1$ cuts in the lattice L_i^{sub}. Given the initial cut, we are to reach the final cut. Cut $(C_1 \dagger P_1)$=cut $((X_0, Y_1) \dagger (X_1, Y_1))$ by the construction of the lattice L_i^{sub} and lies on the path $q_n = q_1$. The final cut $(C_2 \dagger P_2)$= cut $((X_3, Y_0) \dagger (X_3Y_1))$ to be reached lies on the path r_3. Hence we show the reachability sequence between path q_1 to r_3.

The Reachability Sequence in the proof above stated that the order of paths $k_1, \ldots, k_{m+n}$ on which the cuts from path q_n to r_m are found where $k_i \in \Pi_1$ or $k_i \in \Pi_2$ satisfy the following:

1. for $k_j = r_s \in \Pi_1$, $r_{s+1} = k_l, l > j$

2. for $k_j = q_s \in \Pi_2$, $q_{s-1} = k_l, l > j$

This can be illustrated in the example as follows: The paths in π_1 are r_0, r_1, r_2, r_3 while the paths in π_2 are q_0, q_1. Since we traverse the paths from q_1 to r_3, the initial path $k_1 = q_1$ and the final path $k_{m+n} = r_3$. The cut on the path $k_1 = q_1$ corresponds to the initial cut ($c - a - c \dagger a - c$). $C_1 = X_0Y_1$. The child(X_0, Y_0) of $c - a - c$ is infrequent. Consider the node $N = (X_1, Y_0)$. As the node N is infrequent, cut ($N \dagger P_1$) on the path r_1 is found by lines(6-7) of the $ExpandCut$ algorithm. Hence the next cut is found either on path r_1.

Hence $k_1 = q_1$ and $k_2 = r_1$. We next consider the case of $k_s = r_a$ in the proof above. The cut seen on the path r_1 is cut ($(X_1, Y_0) \dagger (X_1, Y_1)$). Let $N_1 = X_1Y_0, N_2 = X_2Y_1$ and $N_3 = X_2Y_0$ then, as N_3 is infrequent, cut ($N_3 \dagger N_2$) on the path r_2 is found by lines(11-13).

Now, $k_1 = q_1$, $k_2 = r_1$ and $k_3 = r_2$. Again considering the case of $k_s = r_a$, the cut seen on the path r_2 is the cut ($(X_2, Y_0) \dagger (X_2, Y_1)$). Again, let $N_1 = X_2Y_0, N_2 = \phi$ and $N_3 = X_3Y_0$ then, as N_3 is infrequent, cut ($N_3 \dagger N_2$) on the path r_3 is found by lines(11-13).

Hence, we have reached the cut on r_3 from the cut q_1 as stated in the reachability sequence in the order of q_1, r_1, r_2, r_3.

3.7 Optimizing MARGIN

In this chapter we discuss the pruning that MARGIN does by the approach it takes followed by further optimizations that can be applied to the naive MARGIN algorithm. The optimizations introduced are very effective in reducing the run-time of the MARGIN algorithm. Results show that about 50% reduction of running time was achieved by them.

3.7.1 Discussion

Consider a cut ($C \dagger P$) in the lattice L_1 of the graph G_1. $P \subset_{graph} G_1$ being frequent (see Fig 3.13), all the parents of P are frequent due to which no $I(\dagger)$ node (hence no cut) can lie among the subgraphs of P in the lattice L_i. Similarly all the supergraphs of C being infrequent, no $f(\dagger)$ node (hence no cut) would lie among the supergraphs of the C. Hence the subgraph and supergraph space of P can be pruned out of the candidate set for maximal frequent subgraphs. Hence, each cut found prunes our search space substantially. MARGIN traverses all the nodes on the cuts in the lattice $L_i \; \forall i$, the infrequent parents of the $I(\dagger)$ nodes and their infrequent parents. The remaining nodes on the lattice need not be explored.

Frequent subgraphs wherein the count value of each frequent subgraph need not be reported can be obtained by reporting all subgraphs of $\mathbb{MF}$ or the $f(\dagger)$ nodes. Furthermore, techniques in [53] can be adapted to provide an approximate frequency count value for the frequent subgraphs.

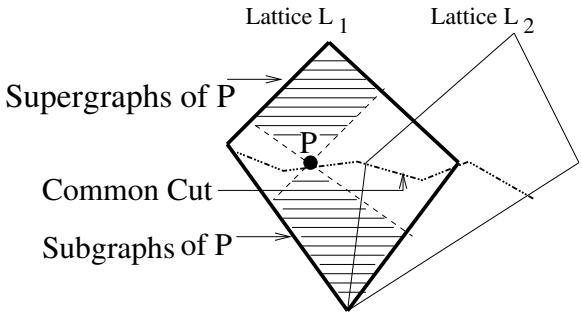

Figure 3.13 Exploiting the commonality of the $f(\dagger)$-$nodes$

3.7.2 Optimizations

Optimizations to reduce the number of revisited cuts are essential in order to reduce the computation time spent on verifying whether the cut has been visited previously. We discuss these optimizations separately in order to keep the main algorithm simple. Few of the optimizations that help speed the results are given below.

- **(A):** Let the cut ($C \dagger P$) be the cut on which $ExpandCut$ has been invoked. Line 1 of the $ExpandCut$ algorithm computes the parents of C. Consider a frequent parent Y_i of C. If there exists at least one frequent child CY_i of Y_i, then by $Upper$-$\diamond$-property, there exists a node M which is the common child of CY_i and C. C being infrequent, M is infrequent. It can be shown that lines 5-10 of the $ExpandCut$ algorithm that iterate over all the children of Y_i can be replaced by calling $ExpandCut$ on just one cut ($M \dagger CY_i$). This leads to reducing the number of revisited cuts. Also, a single frequent child CY_i needs to be generated instead of all the children of Y_i in line 5. Also line 4 is executed only if there does not exist any frequent child CY_i of Y_i.

- **(B):** Consider an invocation of $ExpandCut$ on cut ($C \dagger P$). The parents $Y_1, \ldots, Y_c$ of C are computed in line 1. For some Y_i, $P = Y_i$ since P is a parent of C. Therefore, in line 5 all the children $C_{c_1}, \ldots, C_{c_k}$ of P are computed and explored. If any child $C_{c_i} \neq C$ is infrequent, then $ExpandCut$ is invoked on cut ($C_{c_i} \dagger P$). In the invocation of $ExpandCut$ on the cut ($C_{c_i} \dagger P$), the children of P are recomputed and revisited. Since the children of P are already explored in the invocation of $ExpandCut$ on the cut ($C \dagger P$), their re-exploration can be avoided in the invocation of $ExpandCut$ on cut ($C_{c_i} \dagger P$) by passing the appropriate information.

- **(C):** Consider an invocation of $ExpandCut$ on cut ($C \dagger P$). Lines 11-13 of the algorithm check for infrequent parents Y_i of C. If Y_i is found among the infrequent subgraphs already visited, then $ExpandCut$ invoked on cut ($C \dagger P$) skips executing lines 12-13 on Y_i.

- **(D):** The line 13 of the $ExpandCut$ algorithm finds the parents of the infrequent subgraph Y_i. For the new $ExpandCut$ invoked in line 15, this parent information can be passed as argument so that the recomputation of parents is avoided in line 1 of the new $ExpandCut$.

3.7.3 The Replication and Embedded Model

As mentioned earlier, either of the embedded or replication model can be used by the MARGIN algorithm. Let C_r, $P_r \in G_i$ be a pair of child-parent nodes in the replication model. Then, the graphs C_e and P_e in the embedded model that are isomorphic to subgraphs C_r and P_r would now form a child-parent pair in the embedded model. Hence, every child-parent edge that exists in the lattice of the replication model also exists in the embedded model. Thus, the sequence of cuts in the embedded model would follow the same sequence of child-parent cut edges as in the replication model.

Let $E_i \in G_i$ and $E_j \in G_j$ be two subgraphs isomorphic to g. Let both E_i,$E_j \in f(\dagger)\text{-}nodes$. In the replication model, the counts for E_i and E_j are accumulated in the $ExpandCut$ calls of G_i and G_j respectively. Though the counts of E_i and E_j would be the same, it has been computed twice due to the individual lattices for G_i and G_j. Since in the embedded model there is only one node that corresponds to g, it will be visited only once in the $ExpandCut$ function and hence the frequency will be computed only once. A subgraph in the embedded model is reported as maximal if all its children over the entire graph database are found to be infrequent which is same as globally maximal (maximal with respect to the entire graph database). In the replication model however, we gather locally maximal subgraphs and do an additional step to identify the globally maximal set.

With respect to the two models, there is a trade-off between the amount of information that is to be stored and the number of times the frequency of the subgraphs that lie on the cut are computed. In our implementation, using the replication model, we traverse each graph G_i one at a time and the information like cuts that are already visited and frequency count of nodes on the cut found in each iteration of G_i are flushed out. Hence, the amount of information stored reduces. However, if the subgraphs belong to cuts of multiple graphs, its frequency needs to be recomputed in the iteration of each G_i whose cut it belongs to. On the other hand, in the embedded model, the frequency counts of subgraphs that lie in the overlapping region of the cuts of multiple graphs are computed only once. The common cut region shown in Figure 3.13 belongs to the lattices L_1 and L_2 which will be explored only once in case of the embedded lattice. However, this is at the cost of increased storage space. If the intersection of the $f(\dagger)$ nodes of the graphs in the graph database is expected to be low, the replication model is preferred. In the embedded model, all instances of a subgraph that does not occur as candidate node in a given G_i but occurs as candidate node in some $G_{j\neq i}$ will have to be found. All children of such subgraphs also have to be accounted for and so on. In the replicated model, instead you would have visited not more than one instance in G_i for frequency count. Thus, with minor modifications, the $ExpandCut$ algorithm can be applied to the embedded model.

Chapter 4

Experimental Evaluation

We implemented the MARGIN algorithm and tested it on both synthetic and real-life datasets. We ran our experiments on a 1.8GHz Intel Pentium IV PC with 1 GB of RAM, running Fedora Core 4. The code is implemented in C++ using STL, Graph Template Library [1] and the igraph library [3]. We conducted experiments for comparative results with the gSpan executable [65], *our implementation* of the SPIN algorithm and the CloseGraph [66] executable. We compare with gSpan in order to show the saving MARGIN makes against the time of an algorithm that explores the lattice space below the "border". We compare with CloseGraph as it also prunes the search space to report a subset of the frequent subgraphs and a superset of the maximal frequent subgraphs. Since SPIN generates maximal frequent subgraphs, we compare with it. We give time comparative results with gSpan and both time and generic operation comparisons with SPIN.

Our experimental results show that MARGIN runs more efficiently than SPIN and gSpan on synthetic datasets and denser real-life datasets. The number of lattice node computations with low support values show that about one-fifth of the search subspace of SPIN is visited by MARGIN. Also, the cost of the operations involved in SPIN and MARGIN are comparable while the difference in the number of operations is huge. We generated all maximal frequent subgraphs from the frequent subgraphs obtained by gSpan and cross validated the results with that of MARGIN and SPIN.

We generated the synthetic datasets using the graph generator software provided in [47]. The graph generator generates the datasets based on the six parameters: D (the number of graphs), E,V (the number of distinct edge and vertex labels respectively), T (the average size of each graph), I (the average size of frequent graphs) and L (the number of frequent patterns as frequent graphs).

4.1 Analysis of MARGIN

For varying values of the database size and support, we show in Table 4.1, the total number of nodes in the lattice, the number of frequent nodes in the lattice, the frequent nodes visited by MARGIN which is the set of candidate nodes in MARGIN, the extra set of infrequent nodes that MARGIN visits in order to find the representative and the saving MARGIN makes in terms of the number of nodes visited as

$L = 5, E = 50, V = 50, I = 6, T = 9$						
D	**Sup%**	**Total No. Of Lattice Nodes**	**Frequent Nodes**	**Candidate Frequent Nodes**	**Nodes to find Repre--sentatives**	**Gain in Nodes Explored**
200	8	293359	5908	1946	20	3942
300	12	342463	5908	1946	24	3938
400	16	352371	5908	1946	27	3935
500	20	462611	5908	1948	30	3930
600	24	800855	5908	1948	34	3926
700	28	800855	5908	1948	34	3926
800	32	872536	5908	2034	36	3838
900	36	894800	6414	4278	38	2098
1000	40	1648819	3028	2806	107	115
1100	44	1708471	2867	2668	115	84
1200	48	2137059	3028	2806	120	102
1300	52	2165766	2062	1888	123	51
1400	56	2173221	2818	2627	124	67
1500	60	2274580	2902	2753	131	18

Table 4.1 MARGIN: Lattice Space Explored

against the total number of frequent nodes in the lattice. We ignore the set of infrequent nodes visited by MARGIN while running $ExpandCut$ since the infrequent nodes visited are immediate super graphs of the frequent nodes in the lattice. Such a set of infrequent nodes have to be visited by any Apriori based algorithm in order to stop further extending the subgraph during its depth first search. Table 4.1 shows that the number of infrequent nodes visited to find the representative is much smaller than the total number infrequent nodes (the difference between the total number of nodes in the lattice and the frequent nodes in the lattice). The values of the parameters used to generate the dataset for this experiment are L =5,E =50, V =50, I =6 and T =9. The number of frequent nodes is calculated by running the gSpan algorithm while the total number of lattice nodes is calculated by running gSpan for the graph database with $minSup$=1. The column "Gain in Nodes Explored" thus shows for various experimental values the number of nodes that MARGIN avoided exploring in the lattice space by avoiding a bottom-up approach.

Table 4.1 clearly indicates that the extra set of infrequent nodes visited by MARGIN to find the $Representative$ (Figure 1.1) is very small as compared to the infrequent or frequent lattice space.

Table 4.2 shows the effect of optimizations on the MARGIN algorithm. The optimizations listed in the work are labeled as A, B, C and D. The optimization by B was not significantly affecting the computation time. Table 4.2 shows the time taken algorithm when all the optimizations are used in column "All Opt" of Table 4.2, when no optimizations are used in column "No Opt", when only optimization A is used, when only C is used, when only D is used and only when any two of A,C and D are used. We can observe that the time taken by MARGIN when all the optimizations are used is almost 50% less

$L = 5, E = 50, V = 50, I = 8, T = 10$									
D	**Sup %**	**All Opt (sec)**	**No Opt (sec)**	**Only A (sec)**	**Only C (sec)**	**Only D (sec)**	**CD (sec)**	**AD (sec)**	**AC (sec)**
100	20	42.56	80.86	70.53	63.61	67.9	56.14	54.9	50.08
	15	44.59	84.13	74.29	65.96	71.39	58.16	57.02	53.02
	10	56.38	116.6	100.63	87.37	105.67	78.33	74.61	66.56
	5	82.66	186.87	158.62	134.6	167.68	123.86	116.02	97.75
200	40	107.65	198.74	176.42	147.13	165.93	133.59	139.12	126.02
	30	128.62	252.23	227.06	188.58	210.95	166.83	170.97	149.7
	20	130.35	265.24	240.33	194.79	220.33	171.51	175.96	153.57
	10	135.62	303.74	262.48	212.68	262.67	188.38	188.35	160.8

Table 4.2 Effect of Optimizations

than the time it takes when no optimization is used. The values of the parameters used in this dataset are $L = 5$, $E = 50$, $V = 50$, $I = 8$ and $T = 10$. The sizes of the dataset are 100 and 200, the support values are varied between 5-20%.

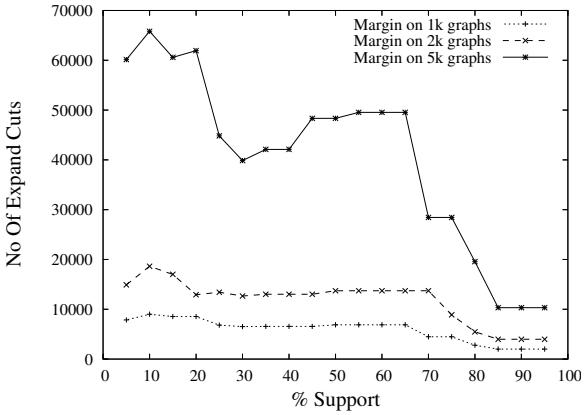

Figure 4.1 Number of Expand Cuts

Figure 4.1 shows the number of times the $ExpandCut$ algorithm is invoked against varying support values. The experiments are conducted on three datasets of sizes 1000, 2000 and 5000. The values of the parameters used to generate the dataset are L=5, E=50, V=50, I=12 and T=15. The Figure shows that the number of invocations of the $ExpandCut$ algorithm does not vary linearly with the support. Such a non-linear variation is attributed to the non-dependence on level wise traversal but instead to the number of $f(\dagger)$-$nodes$ in the graph lattice. The number of $ExpandCut$ invocations approaches zero for large support values since most subgraphs may not find the required support and hence the $ExpandCut$ algorithm might be moving between ϕ_g and single node graphs which terminates quickly. The lattice structure can create an anomaly behavior in MARGIN because of the structure of different graphs.

Figure 4.2 shows the time taken by the MARGIN algorithm against varying the support values for the same datasets as that used in Figure 4.1. However, the exact support of the subgraph is not computed.

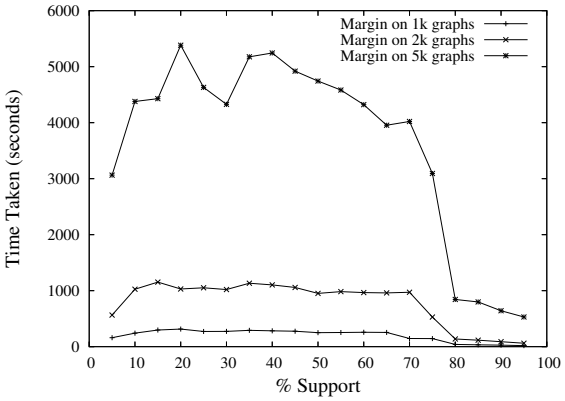

Figure 4.2 Runtime of MARGIN

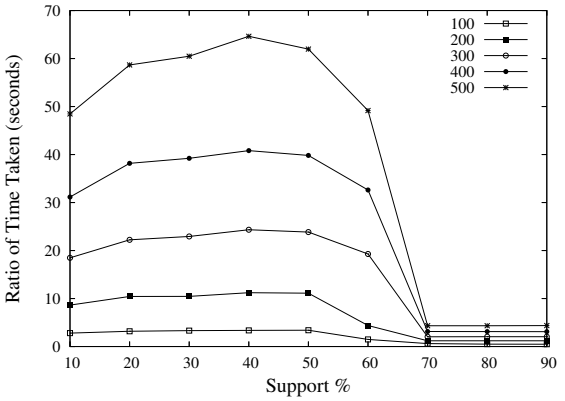

Figure 4.3 Runtime of MARGIN for exact support count

The support count function stops when the support exceeds the $minSup$ value or when the number of graphs in which it does not occur exceeds $|D| - minSup$. Figure 4.2 thus does not show the time variation it would incur if the exact support of a subgraph were to be computed. Figure 4.3 shows the variation of time with support when the exact support is computed. An increase in time followed by a decrease in time thereafter is seen. For low support values, the border between the infrequent and frequent nodes of the lattice, lies in the higher levels of the lattice. As the support value increases, the border moves towards the central levels in the lattice and further down towards the lower levels. Thus, with increasing support values, the number of nodes visited increases and then decreases due to which the computational time increases and then decreases. The parameters used in Figure 4.3 for D=100 to 500 are L=5, E=50, V=50, I=5 and T=8. The support has been varied from 10% to 90%. With an increase in size of the database, the time MARGIN takes to find the maximals increases proportionally as more number of graphs need to be checked for the support.

As in Figure 4.1, the variation of the time taken shows similar behavior as that of the number of $ExpandCut$ invocations which forms the core operation in the MARGIN algorithm.

4.2 Comparing MARGIN with gSpan

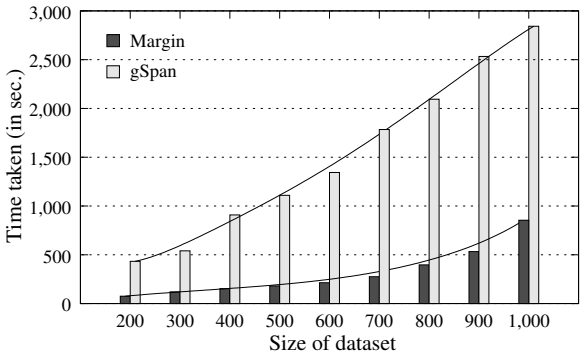

Figure 4.4 Running time with 2% Support

$L = 5, E = 50, V = 50, I = 12, T = 15$			
D	**MARGIN (sec)**	**gSpan (sec)**	**Ratio**: $\frac{\text{gSpan}}{\text{MARGIN}}$
200	32.73	616.56	18.84
300	44.51	932.43	20.95
400	45.97	1222.2	26.59
500	64.23	1409.55	21.94
600	90.43	1532.23	16.94
700	132.1	1654.32	12.52
800	156.07	1839.38	11.79
900	183.78	2432.43	13.24
1000	265.43	2842.42	10.71

Table 4.3 Running time with support 20

Figure 4.4 shows the result where D is varied between 200 and 1000 graphs. The other values of the parameters used for this experiment are: L=5, E=50, V=50, I=12 and T=15. The minimum support used for each case is 2% of D. Since the average size of each graph is 15 and the average size of each frequent subgraph is 12, the maximal frequent subgraphs tend to lie in the higher levels of the lattice. MARGIN avoids level-wise traversal of the lattice which speeds up the results making MARGIN perform better.

Table 4.3 shows the running times of MARGIN and gSpan on datasets with varying number of graphs from 200 to 1000. The parameters used are: L=5, E=50, V=50, I=12 and T=15. The last column shows the ratio of the running times of gSpan to that of MARGIN. It can be observed that MARGIN performs better than gSpan.

Figure 4.5 compares the effect of varying average size of the frequent graphs. In this experiment, we vary the parameter I from 5 to 12 while generating the dataset. The number of graphs generated are 1000 and the minimum support is taken to be 2%. The other parameters are set to as follows: E=50,

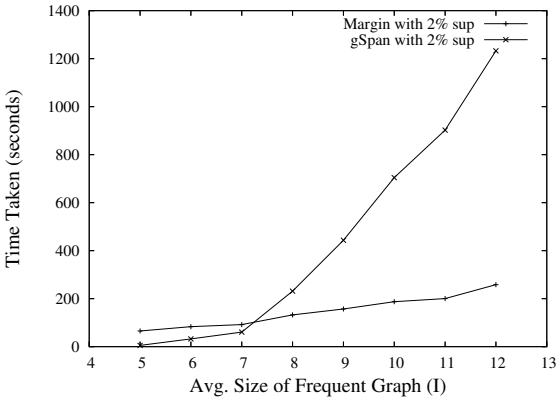

Figure 4.5 Comparison with gSpan: Effect of size of frequent graphs

V=50, L=5 and T=20. It was observed that gSpan performs very efficiently with lower values of I (5-7). With increasing values of I, MARGIN performs better. This should be expected as for higher values of I, the lattice space explored for Apriori-based algorithms would increase since larger graphs are expected to be frequent.

We next compare the running times of gSpan and MARGIN with varying ratios of the number of edges to the number of vertices. This measure gives us an estimate of the density of the graphs. We represent the edge to vertex ratio by the symbol "*EVR*".

For higher edge-to-vertex ratio around *EVR*=2, the parameters used are $D = 100$ to 500, $L = 5$, $E = 20$, $V = 20$, $I = 15$, $T = 18$ and the support values varying from 10 to 50. Tables 4.4 and 4.5 shows the results and it can be observed that the running time of MARGIN is significantly less than that of gSpan.

$L = 5$, $E = 20$, $V = 20$, $I = 15$, $T = 18$						
	$D = 100$, *EVR*$= 1.98$		$D = 200$, *EVR*$= 2.12$		$D = 300$, *EVR*$= 2.16$	
Support	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**
10	3.38	0.28	49.97	0.59	106.42	0.95
20	3.41	0.29	49.98	0.62	110.48	1.01
30	3.38	0.28	50.47	0.64	106.56	1.07
40	3.32	0.52	48.41	1.62	110	3.42
50	3.34	0.72	48.45	1.43	106.32	3.29

Table 4.4 Comparison of gSpan and MARGIN for high edge-to-vertex(*EVR*) ratio:D=100-300

We repeated the experiment for low edge-to-vertex ratio where the ratio is less than 1. The parameters used for the generation of the dataset are: $L = 5$, $E = 20$, $V = 20$, $I = 6$ and $T = 8$. Tables 4.6 and 4.7 show the result and the running times of gSpan to be small for this case.

We also present results for varying edge-vertex ratio of *EVR*=1.25 to *EVR*=2 in Table 4.8

$L = 5$, $E = 20$, $V = 20$, $I = 15$, $T = 18$				
	$D = 400$, *EVR*= 2.17		$D = 500$, *EVR*= 2.18	
Support	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**
10	163.57	1.3	220.17	1.65
20	164.41	1.41	218.86	1.81
30	164.5	1.51	220.41	1.94
40	162.87	4.22	218.91	5.02
50	163.09	4.12	220.66	4.97

Table 4.5 Comparison of gSpan and MARGIN for high edge-to-vertex(*EVR*) ratio: D=400,500

$L = 5$, $E = 20$, $V = 20$, $I = 6$, $T = 8$						
	$D = 100$, *EVR*= 0.91		$D = 200$, *EVR*= 0.97		$D = 300$, *EVR*= 0.97	
Support	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**
10	0.03	2.42	0.02	9.03	0.03	13.35
20	0.02	2.68	0.01	12.16	0.03	22.17
30	0.02	2.56	0.01	14.61	0.02	25.04
40	0.01	2.16	0.01	7.34	0.02	25.74
50	0.01	1.24	0.01	7.61	0.02	29.43

Table 4.6 Comparison of gSpan and MARGIN for low edge-to-vertex(*EVR*) ratio: D=100-300, *EVR*=0.9

We can observe that gSpan takes less time for lower values of edge-to-vertex ratios while MARGIN performs better for higher values. In sparse datasets, the Apriori-based approaches have the advantage of having to extend to fewer children. As each extension gives rise to a subtree rooted on it, each reduced extension saves on computation time considerably. Hence, a decrease in the number of children cascades the improvement in the performance of the Apriori-based approaches due to which they would perform considerably better than MARGIN.

4.3 Comparing MARGIN with SPIN

Table 4.9 shows the number of nodes in the lattice that are explored by the MARGIN algorithm as compared to SPIN when executed on synthetic datasets of various sizes and support. As can be seen, MARGIN explores one-fifth of the nodes explored by SPIN .

Figure 4.6 shows a time comparison of SPIN and MARGIN. Time with varying support values has been shown for D=500 and D=1000, with other parameters set to E=10, V=10, L=10, I=5 and T=6

$L = 5$, $E = 20$, $V = 20$, $I = 6$, $T = 8$				
	$D = 400$, *EVR*$= 0.92$		$D = 500$, *EVR*$= 0.98$	
Support	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**
10	0.03	24.88	0.06	37.92
20	0.02	36.16	0.05	43.53
30	0.02	42.32	0.04	59.59
40	0.02	35.09	0.04	61.6
50	0.02	35.32	0.03	76.16

Table 4.7 Comparison of gSpan and MARGIN for low edge-to-vertex(*EVR*) ratio: D=400-500, *EVR*=0.9

$L = 5$, $E = 20$, $V = 20$, $I = 15$, $T = 18$, $D = 100$								
	EVR$= 1.25$		*EVR*$= 1.5$		*EVR*$= 1.75$		*EVR*$= 2$	
Support	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**	**MARGIN (sec)**	**gSpan (sec)**
10	1062.3	12.17	2.88	21.2	0.6	274.87	0.28	3.38
20	657.53	4.52	1.9	20.93	0.56	261.72	0.29	3.41
30	1491.17	0.43	1.86	21.46	0.52	264.96	0.28	3.38
40	535.97	0.2	1.82	20.95	0.47	263.98	0.52	3.32
50	60.98	0.02	6.99	13.21	2.48	265.43	0.72	3.34

Table 4.8 Comparison of gSpan and MARGIN for various edge-to-vertex(*EVR*) ratio

and varying support from 1 to 5%. With an increase in the support values, the number of graphs that are frequent reduce and hence the lattice space below the "border" is smaller. It can be seen that with an increase in the support value, the time taken by MARGIN and SPIN reduce to comparable values. However, for smaller values of support which causes the "border" to be much higher up the lattice, MARGIN performs better than SPIN as expected.

We ran more experiments to compare MARGIN with SPIN. Figure 4.7 shows the result.

Figure 4.7 shows the ratio of time taken by SPIN to that of the MARGIN when the support is varied. The size of the dataset is varied from 400 to 700. The other values of the parameters used for generation of the data are: $L = 5$, $E = 50$, $V = 50$, $I = 10$ and $T = 11$. The x-axis shows the percentage of the support which is varied between 2.5% and 5% in the increments of 0.5%. The y-axis shows the ratio of the time taken by SPIN to the time taken by MARGIN. It can be seen that the ratio ranges from 11-18 for the parameters as listed above. We have conducted further experiments and found that the ratio depends on the parameters used.

Table 4.10 shows the time taken by MARGIN and SPIN along with their ratio which is plotted in Figure 4.7.

$E = 10, V = 10, L = 10, I = 5, T = 6$		
DataSet	Lattice Nodes Visited	
D(**Support%**)	**SPIN**	**MARGIN**
100 (2)	43,861	9,311
200 (2)	54,026	10,916
300 (2)	57,697	14,954
400 (2)	58,929	42,201
100 (5)	42,584	9,930
200 (5)	49,767	12,318
300 (5)	52,118	24,660
400 (5)	54,726	44,686
500 (2)	32,556	12,619
500 (3)	21,669	10,078
500 (4)	9,187	9,912
500 (5)	4,162	8,264

Table 4.9 Lattice Space Explored

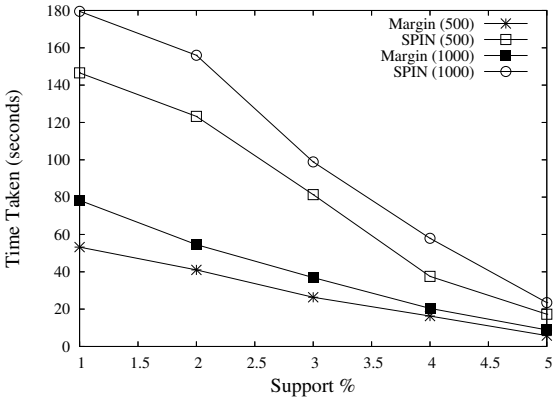

Figure 4.6 Comparison of MARGIN with SPIN

We also ran experiments to compare the memory used by gSpan, SPIN and MARGIN. We measured the memory used by the executables of these algorithms as the maximum value of "percentage memory" reported by the "top" command [6] (standard tool available on all the Linux implementations) during the entire course of execution of the binary. In order to determine this maximum percentage memory, we used the library Libgtop [4] in the implementation.

Table 4.11 shows the memory values for the two datasets which are used for the generation of the results reported in Figure 4.6 for which the dataset parameters are: $D = 500$ & 1000, $E = 10$, $V = 10$, $L = 10$, $I = 5$ and $T = 6$ and the support is varied between 1% to 5%.

As Table 4.11 shows, the memory taken by MARGIN is more than that of gSpan and SPIN. This is because the number of subgraph patterns that are to be stored in the main memory at any instance of time is very less for gSpan and SPIN compared to that of MARGIN. gSpan visits the subgraphs in a depth-first manner and extends them without having to store the intermediate nodes. Similarly,

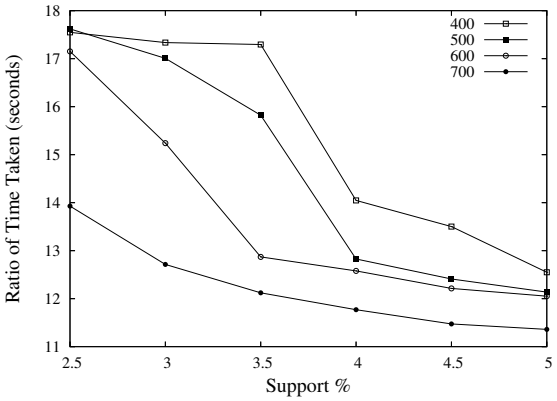

Figure 4.7 Ratio of SPIN to MARGIN

SPIN generates the maximal tree patterns which are then extended to maximal subgraphs with not many intermediate nodes being stored. On the other hand, the memory incurred by MARGIN might be large because it needs to store the nodes on the border of each graph lattice, in order to avoid running the $ExpandCut$ function on the cuts visited earlier. As we have seen, gSpan and SPIN take less memory compared to MARGIN.

An estimate of the number of nodes visited in the lattice has been provided in Table 4.9. The database parameters are similar to that of Table 4.12. Since we implement the replication model, we include the count of all subgraphs visited in MARGIN while in SPIN we accumulate only the number of non-isomorphic subgraphs generated. The distinct graphs visited by MARGIN can be only lesser than that on including all the isomorphic forms. It can be seen that MARGIN visits less than one-fifth of the space of SPIN.

Comparison of Generic Operations

Since time comparison is not a good measure to compare algorithms, we include a comparison of the complexity of the most frequent operations of both algorithms. We consider the subgraph and graph isomorphic operations of the MARGIN algorithm and the subtree isomorphism and maximal-CAM-tree operations of the SPIN algorithm.

In SPIN, in order to calculate the frequency of a subgraph, the information about the occurrence of the parents which is passed over the levels, is utilized. Instead, MARGIN requires a subgraph isomorphic check for the frequency computation of a graph g where g is being generated from its child. Although the problem of determining whether g is a subgraph of G is hard in theory, based on all experiments conducted by us we find it to be of O(n^4) where n is the number of edges in g. In our experiments, for each subgraph isomorphic operation we calculate the count c of the total number of edges of G that are visited. We then determine $log_n c{=}k$ which expresses in n the complexity of the subgraph isomorphic operation. Running the MARGIN algorithm on various datasets for varying parameters, we estimate the maximum value of k to be 3.4. Also, for almost all operations, k was found to be less than 2.1. This is much lower than the complexity of subtree isomorphism which is O($t.n.\sqrt{t}$) [19] where t is the size

$L = 5, E = 50, V = 50, I = 10, T = 11$				
D	Sup%	**MARGIN (sec)**	**SPIN (sec)**	$\frac{SPIN}{MARGIN}$
400	2.5	34.34	602.59	17.55
	3	31.44	545.01	17.33
	3.5	29.9	517.11	17.29
	4	35.43	497.66	14.04
	4.5	37.58	507.41	13.50
	5	37.99	476.8	12.55
500	2.5	41.34	728.39	17.62
	3	36.53	621.26	17.01
	3.5	39.81	629.83	15.82
	4	47.34	607.22	12.83
	4.5	49.83	618.36	12.41
	5	50.09	607.91	12.14
600	2.5	53.09	910.56	17.15
	3	57.34	873.87	15.24
	3.5	61.86	796.11	12.87
	4	61.48	773.28	12.58
	4.5	64.75	790.85	12.21
	5	64.07	772.16	12.05
700	2.5	65.08	906.51	13.93
	3	71.3	906.51	12.71
	3.5	76.78	930.76	12.12
	4	76.62	901.86	11.77
	4.5	80.56	924.3	11.47
	5	79.57	903.9	11.36

Table 4.10 Comparison of MARGIN and SPIN

of the larger tree in which the tree of size n is being searched. Similar analysis on SPIN gives us tree isomorphism to be of O(n^2) and maximum-CAM-tree to be of O(n^3).

Table 4.12 shows the number of generic operations in SPIN and MARGIN for various support and database sizes. The first eight rows correspond to results on dataset with parameters $D = 100$-400, $T = 15$, $L = 5$, $I = 7$, $V = 8$, $E = 8$ and support values of 2% and 5%. In SPIN, the generic operations include tree isomorphism and finding the maximum CAM tree of a graph. The remaining operations like candidate generation and associative edge pruning have been ignored. The generic operations of MARGIN include subgraph isomorphic checks for count computation. The only operation of MARGIN that is ignored is the generation of its parents and children. The operations of order O(n^2) and greater than O(n^2) are listed separately as shown in Table 4.12. We can observe that with an increase in the database size for a constant support, the number of operations of MARGIN performs two to three times better than SPIN.

$L = 10, E = 10, V = 10, I = 5, T = 6$				
D	**Support %**	**gSpan Memory**	**SPIN Memory**	**Margin Memory**
500	1	0.14	5.79	15.03
	2	0.14	2.69	14.52
	3	0.08	2.23	11.04
	4	0.08	1.58	11.56
	5	0.07	1.48	11.40
1000	1	0.17	13.58	24.56
	2	0.16	5.65	27.33
	3	0.16	4.47	23.12
	4	0.14	2.98	23.94
	5	0.14	2.82	10.35

Table 4.11 Memory Comparison of gSpan, SPIN and MARGIN

For datasets with small graphs, as the support increases, the lattice space below the "border" decreases. For the next four rows of the Table 4.12 we use $D = 500$, $T = 8$, $L = 5$, $I = 7$, $V = 8$ and $E = 8$ with varying support of 2%-5%. As can be observed, the performance of MARGIN to that of SPIN degrades with increase in support for small graphs leading to better performance of SPIN in some cases. Similar explanation holds for the last row of Table 4.9.

The last four rows of Table 4.12 correspond to results with parameters set to $D = 100$, $L = 5$, $V = 8$, $E = 8$ and a support of 2% for varying values of T and I. As T increases from 5 to 20 with I set to 75% of T, it is noticed that the ratio of number of operations of SPIN to that of MARGIN goes up to 20. This is because as T increases, the lattice space below the "border" increases and thus SPIN explores a bigger space as compared to MARGIN.

4.4 Comparing MARGIN with CloseGraph

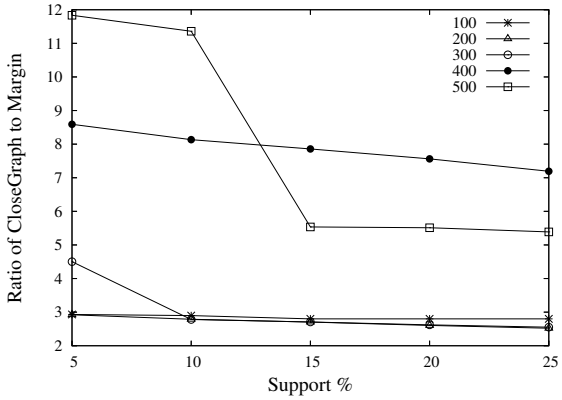

Figure 4.8 Time comparison with CloseGraph. $I = 20$, $T = 25$

DataSet	O(n^2)		greater than O(n^2)	
D (**Support**)	**SPIN**	**MARGIN**	**SPIN**	**MARGIN**
$L = 5, E = 8, V = 8, T = 15, I = 7$				
100(2)	1,76,781	39,669	7,956	466
200(2)	2,35,457	53,738	14,181	7,289
300(2)	2,86,788	72,270	17,515	8,931
400(2)	3,28,273	1,33,533	19,432	11,722
$L = 5, E = 8, V = 8, T = 15, I = 7$				
100(5)	1,79,081	58,229	6,156	2,349
200(5)	1,93,598	75,270	10,523	7,962
300(5)	2,47,826	1,00,225	15,568	10,368
400(5)	3,24,216	1,28,363	17,164	13,954
$L = 5, E = 8, V = 8, T = 8, I = 7$				
500(2)	2,57,626	1,84,301	33,669	15,937
500(3)	1,85,442	1,43,021	16,764	11,898
500(4)	87,883	96,703	8,677	10,145
500(5)	24,770	77,637	5747	9454
T (I)	**SPIN**	**MARGIN**	**SPIN**	**MARGIN**
$D = 100, L = 5, E = 8, V = 8$, Supp=2%				
5 (4)	63,194	20,569	5,328	304
10 (8)	1,45,772	36,999	12,220	668
15 (11)	3,11,147	53,937	17,220	783
20 (15)	10,42,997	1,64,992	1,28,910	3,378

Table 4.12 Generic Operations

We compare MARGIN with CloseGraph [66] since CloseGraph finds closed frequent subgraphs and hence is a superset of the maximal frequent subgraphs. Figure 4.8 and Figure 4.9 show the ratio of the time taken by CloseGraph to that of MARGIN. The dataset parameters used are: $D = 100 - 500$, $E = 20$, $V = 20$ and $L = 10$ with support varying from values 5 to 25. Figure 4.8 takes parameters $I = 20$ and $T = 25$ while Figure 4.9 takes parameters $I = 25$ and $T = 30$.

It can be seen that by increasing the values of I, T, the ratio of the time taken by CloseGraph and MARGIN increases. This is because, the lattice space to be explored increases for Apriori-based algorithms.

4.5 Analysis of MARGIN-d

We provide an analysis of the disconnected maximal frequent subgraph behaviour and show results against the connected maximal frequent subgraph version.

It was observed that with increasing values of I the time taken increases and again reduces as in Table 4.13. Note that the graph lattice has less nodes at the lower and higher layers of the lattice as against

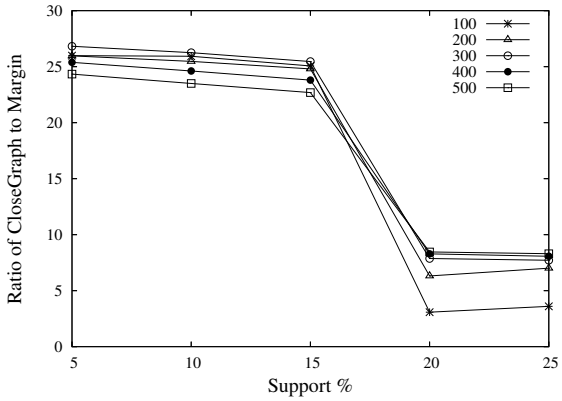

Figure 4.9 Time comparison with CloseGraph. $I = 25$, $T = 30$

the middle layers of the lattice. Hence, with large values of I close to the T, the maximal subgraphs lie higher up the lattice which reduces the number of nodes visited. Similarly for very low values of I, the nodes visited lie close to the empty graph and hence the number of nodes visited reduces and hence the time required for its running.

D=100, L=20, E=20, V=20, T=12, Support=80%		
DataSet	Margin	Margin-d
Size(I)	**Time(sec)**	**Time(sec)**
100(5)	4.23	11.8
100(7)	5.49	15.31
100(9)	5.91	22.96
100(11)	4.92	12.98

Table 4.13 Effect of varying I

With an decrease in E, V, similarity in the graphs generated increases. This leads to larger frequent subgraphs. Also, since more labels tend to match due to higher similarity, detecting the correct component mapping becomes complex and hence increase the running time. Hence, in MARGIN-d for increasing values of E, V the time decreases as seen in Table 4.14.

D=100, L=20, I=5, T=7, Support=80%		
DataSet	Margin	Margin-d
Size(E,V)	**Time(sec)**	**Time(sec)**
100(20)	2.34	6.1
100(15)	2.32	7.3
100(12)	2.36	8.59
100(8)	3.06	19.8
100(5)	3.53	20.44

Table 4.14 Effect of varying E, V

The running time of MARGIN to MARGIN-d was found to be on an average five times lesser as in Table 4.15. The subgraph isomorphic operation of disconnected maximal frequent subgraphs is more expensive as that of connected maximal frequent subgraphs. Also, with an increase in the size of the database, the time increases as the subgraph isomorphism operation needs to iterate through more graphs of the database to find the frequency of a subgraph. All maximal connected frequent subgraphs are a subset of some maximal disconnected frequent subgraphs for the same support which might lead to a reduction in the number of maximal frequent disconnected subgraphs. However, a set of non-maximal subgraphs together might form a maximal frequent disconnected subgraph which might lead to an increase in the subgraphs explored. The increase in running time can be attributed to the cost of the subgraph isomorphism operation. The search space of Apriori techniques where all connected as

D=200-700, L=20, I=5, T=7, E=20, V=20, Support=90%		
DataSet	Margin	Margin-d
Size	**Time(sec)**	**Time(sec)**
200	4.22	15.4
300	7.92	37.89
400	10.67	60.88
500	16.27	90.76
600	20.34	140.27
700	21.13	140.14

Table 4.15 Effect of Varying Database Size D

well as disconnected children of a frequent node have to be computed, is expected to be much higher than that of its connected counterpart.

In Apriori techniques, the property that the connected frequent maximal subgraph is subgraph of a disconnected maximal subgraph does not help in a reduction of the number of nodes. In order to find the disconnected maximal frequent subgraph, an Apriori technique would have to explore all subgraphs of it, both connected and disconnected. Hence the number of nodes visited by Apriori techniques has to increase.

Earlier, we had estimated MARGIN to do three-four times better than SPIN. Here, we have estimated MARGIN to do five times better than MARGIN-d. Hence, MARGIN-d takes approximately twice the time taken by SPIN for connected graphs. SPIN for disconnected subgraphs would hence do better than MARGIN-d only if the increase in the time taken by SPIN for disconnected graphs is double that of SPIN for connected graphs. Let us consider a high approximation of an average of $(e-k)/2$ generated children for each subgraph in the connected subgraph scenario as against a fixed $e-k$ generated children in the disconnected scenario where e is the total number of edges and k the number of edges in the subgraph. Also it can be seen that the cost of canonical minimal tree finding for disconnected subgraphs is higher than for connected subgraphs. Taking into consideration these factors the increase in running time of Apriori based techniques can be estimated to be much more than double the present running time.

Chapter 5

Applications of Margin

In this chapter we present the various applications of the Margin algorithm. Section 5.1 generalises the $ExpandCut$ technique used in the Margin algorithm such that it can be applied to areas outside graph mining. Section 5.2 presents the application of Margin to the RNA database. Section 5.3 presents other real life experiments conducted comparing Margin with other existing algorithms.

5.1 Applications of the $ExpandCut$ technique

The procedure adopted in the MARGIN algorithm is not limited to finding maximal frequent subgraphs alone. The $ExpandCut$ technique can be used to solve a wider range of problems. In this chapter we discuss below a generic platform to which the $ExpandCut$ technique can be applied. We first state the properties that need to hold for applying this framework. We further give few scenarios where the $ExpancCut$ technique can be applied. In section 5.1.2 we illustrate how a specific class of OLAP queries can be answered using the $ExpandCut$ technique.

5.1.1 Framework for applying $ExpandCut$

We list below a set of properties that should be present in a problem statement in order to apply the $ExpandCut$ technique to solve it.

1. The search space is a subset of elements on a lattice.
2. The $Upper$-$\diamond$-property (Chapter 3.3) holds.
3. A constraint C is anti-monotone if and only if for any pattern S not satisfying C, none of the super patterns of S can satisfy C. In the case of maximal subgraph mining, the pattern S is a subgraph and the constraint C refers to the subgraph being frequent. Similarly, a constraint C is monotone if and only if for any pattern S satisfying C, every super pattern of S satisfies C. The elements of the lattice need to satisfy either the monotone or the anti-monotone property in order to apply the $ExpandCut$ technique.

4. A candidate set can be defined which is a "boundary" set such that every element in the set satisfies a given user-constraint and there exists an immediate child in the lattice that does not satisfy the constraint in a anti-monotone lattice. Also for every element in the set, there exists an immediate parent that does not satisfy the constraint in a monotone lattice.

5. The solution set can be generated from the candidate set.

Note that the candidate set could be a set of sets such that elements of each set act as a boundary set.

Examples: The $ExpandCut$ function can be used in solving the following problem statements that satisfy the properties listed above.

1. The closed frequent itemsets and subgraphs define layers of "boundary" where each layer acts as a boundary for a particular value of support. The uppermost layer acts as the boundary region between elements of frequency above and below the user defined threshold. Hence the candidate set here is a set of sets.

2. $ExpandCut$ can be adopted for mining minimal itemsets such that the sum of the cost of items in the itemset exceeds a given threshold (for positive costs). This qualifies as an example for a lattice wherein the monotone property is satisfied.

3. Given a data cube lattice, $ExpandCut$ can be used to find all data cubes which satisfy a given user constraint that follows a monotonic property. For example, we could report all the subset of attributes for which the sum of sales in their data cube is greater than a given user threshold.

4. The problem of finding maximal frequent cliques satisfies the listed conditions.

5. $ExpandCut$ can be used to find the frequent itemsets satisfying a given user support range x_1 to x_2 where $x_1 > x_2$. While the lattice elements satisfy the anti-monotone property; the resultant set lies between two boundary regions. The lower boundary separates the elements of the solution from the elements of support greater than x_1. Similarly, the upper boundary separates the elements of the solution from the elements of support smaller than x_2. Here the candidate set is a subset of the solution set which can be generated based on the candidate set.

5.1.2 Using $ExpandCut$ to process OLAP queries

Data cube is a data abstraction that allows one to view aggregated data from a number of perspectives. Conceptually, the cube consists of a core or base cuboid, surrounded by a collection of sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. The dimension to be aggregated is called the measure attribute, while the remaining dimensions are known as the feature attributes.

In total, a d-dimensional base cube is associated with 2d cuboids. Each cuboid represents a unique view of the data at a given level of granularity. Not all these cuboids need actually be present since any

cuboid can be computed by aggregating across one or more dimensions in the base cuboid. Nevertheless, for anything but the smallest data warehouses, some or all of these cuboids may be computed so that users may have rapid query responses at run time.

The data cube lattice depicts the relationships between all 2d cuboids in a given d-dimensional space. (See 5.1) Starting with the base cuboid which contains the full set of dimensions, the lattice branches out by connecting every node with the set of child nodes/views that can be derived from its dimension list. In general, a parent containing *d* dimensions can be connected to *d* views at the next level in the lattice.

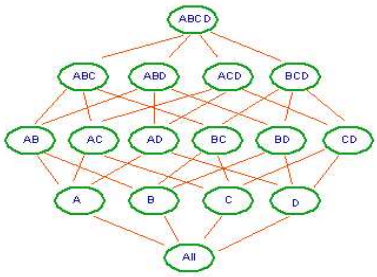

Figure 5.1 Data Cube Lattice

The commonly used operations on a data cube for OLAP analysis are roll-up, drill-down and slice-dice operations. These operations allow an analyst to view the data at different levels of granularity for deriving useful business logic. However, certain applications might require finding the subsets of dimensions that contribute to the aggregation of measure value being more than or less than a given threshold. For example, consider a company data warehouse with product, time, place, manufacturer as the feature attributes and with sales as the measure attribute. One of the overlap queries involves finding the subsets of the feature attributes such that their total sales is more than a given user threshold t. The number of rows that satisfy a query with a condition say year=X will be more than or equal to the number of rows that with the conditions year=X and place=Y. Hence, more rows qualify as the number of features reduce and hence larger will be the sum of the rows that qualify. Thus, if for a subset of features s, the total sales is greater than the required threshold t, then, for every subset of s, the total sales will also be greater than t. Since the number of feature attributes of a typical data warehouse is large, finding all such subsets will be prohibitively expensive. Also, the subsets of our interest can be computed from the set of maximal attribute subsets. The *threshold* t is different from the notion of *support* used in frequent mining algorithms. The term *support* is referred to as a user desired frequency value. That is, the user is interested in those graphs or itemsets whose frequency is above the support value. On the other hand, in the data cube scenario, the user is interested in only those data cubes whose aggregate attribute value for any row is above *threshold* value t.

The data cube lattice satisfies the properties required by the $ExpandCut$ technique as given in section 5.1.1. Hence, the $ExpandCut$ can be used to find the maximal attribute subsets whose aggregated measure value is more than or less than a given user threshold. Given two subsets A, B of attributes, the

cut(A†B) exists if the aggregated measure value of A exceeds the user threshold while that of B does not and B is the immediate superset of A.

We modified the MARGIN algorithm to run on a data cube lattice. $ExpandCut$ would now generate subsets and supersets of a dimensional subset as parents and children respectively. A dimensional set qualifies as frequent if the aggregated minimum measure value is greater than t. We ran our experiments on a 1.8GHz Intel Pentium IV PC with 1 GB of RAM, running Fedora Core 4. The code is implemented in C++ using STL. We generated fact tables with varying number of feature attributes and varying number of rows. By changing the user threshold, we find the maximal attribute subsets whose aggregated minimum measure value exceeds the user threshold. We store the fact table as a relation in MySQL database and we issue the corresponding group by queries in order to compute the aggregated measure value for each data cuboid. The number of attributes is varied from 5 to 20 feature attributes. The number of rows is varied from 5000 to 50,000.

Supp %	No.Of Rows: 1000		No.Of Rows: 25000		No.Of Rows: 50000	
	Time (sec)	**No.Of Maximals**	**Time (sec)**	**No.Of Maximals**	**Time (sec)**	**No.Of Maximals**
1	11.01	78	18.2	105	63.77	195
2	0.43	13	18.2	105	17.69	99
3	0.5	15	18.25	105	14.96	74
5	0.5	15	0.51	15	0.52	15
10	0.5	15	0.51	15	0.5	15
15	0.51	15	0.5	15	0.5	15

Table 5.1 Finding Maximal Attribute sets for 15 feature attributes

Supp %	No.Of Rows: 1000		No.Of Rows: 25000		No.Of Rows: 50000	
	Time (sec)	**No.Of Maximals**	**Time (sec)**	**No.Of Maximals**	**Time (sec)**	**No.Of Maximals**
1	5.94	34	155.3	190	861.67	441
2	1.79	13	154.29	190	146.67	180
3	1.86	17	154.55	190	121.67	117
5	2.16	20	2.16	20	2.16	20
10	2.15	20	2.18	20	2.14	20
15	2.16	20	2.19	20	2.16	20

Table 5.2 Finding Maximal Attribute sets for 20 feature attributes

Table 5.1 and Table 5.2 show the time taken for $ExpandCut$ to run on the data cube lattice and the number of maximal attribute subsets when the user threshold is varied as 1-15% of the sum of the measure value. The number of attributes considered is 15 in Table 5.1 and 20 in Table 5.2. The number of attributes in the attribute combinations of the data cuboids which qualify are more for lower support values. Hence, the number of sub-cuboids found during the ExpandCut operation are more. This could

be responsible for the sudden rise in time taken from 3% to 5% seen. However, the number of expand cuts called is largely dependent on the dataset.

The SPIN and gSpan algorithm cannot be generalised to be applied to other domains and have been specifically developed for the purpose of pattern mining in graphs. We present MARGIN not only as a solution to finding maximal frequent subgraphs but as a technique that can be applied to multiple domains once the properties mentioned in section 5.1.1 are satisfied.

5.2 RNA Margin

The Ribo Nucleic Acid(RNA) can both encode genetic information and catalyze chemical reactions. As the only biological macromolecule capable of such diverse activities, it has been proposed that RNA preceded DNA and protein in early evolution. RNA structure is important in understanding many biological processes, including translation regulation in messenger RNA, replication of single-stranded RNA viruses, and the function of structural RNAs and RNA/protein complexes. Therefore, developing methods for recognizing conformational states of RNA and for discovering statistical rules governing conformation may help answer basic biological and biochemical questions including those related to the origins of life.

A detailed understanding of the functions and interactions of biological macromolecules requires knowledge of their molecular structure. Structural genomics, the systematic determination of all macromolecular structures represented in a genome, is focused at present exclusively on proteins. However, it is being increasingly clear that apart from the earlier known RNA molecules such as mRNAs, introns, tRNAs and rRNAs, a large number of non (protein) coding RNA or ncRNA molecules arising out of non coding regions of the genome play a variety of significant functional roles in cells. A comprehensive understanding of these roles, which include protein synthesis and targeting, several forms of RNA processing and splicing, RNA editing and modification and chromosome end maintenance, is crucial to our understanding of life processes at the molecular level. With the introduction of high throughput experimental methods for identification, the databases of ncRNA molecules have already started swelling up at a rapid pace. After identification, the next task is to understand their structures in their functional context. The structural bioinformatics of RNA molecules is thus an important and emerging research area. The realization that conserved amino acid motifs in proteins can often be related to function has greatly aided the evaluation of unidentified open reading frames in sequence databases. As sequences have accumulated, so has the number of recognizable motifs, thereby guaranteeing that an ever-increasing role will be played by functional inference or in silico analysis of sequence motifs. RNA molecules due to their composition being based on only four major nucleotides renders the primary sequence of RNA insufficient, by itself, in defining motifs. Secondary and tertiary structural aspects must therefore be made part of RNA motif definitions. In spite of these complications, evidence is accumulating that RNA motifs will provide the ultimate basis for an understanding of RNA structure and function.

Given a RNA database, it is interesting to identify different characteristics that are common across RNA molecules and local environment and species dependent variations thereof. We also try to identify motifs from the RNA structures. Shown in the Figure 5.2 is a small RNA graph as fed to the MARGIN algorithm. The nodes contain labels of the form Cx_y where x stands for the node number and y the node label. The double and single bonds indicate the covalent and hydrogen bonds respectively. Another figure of a RNA converted into the MARGIN format which is of the typically seen size is given in Figure 5.3.

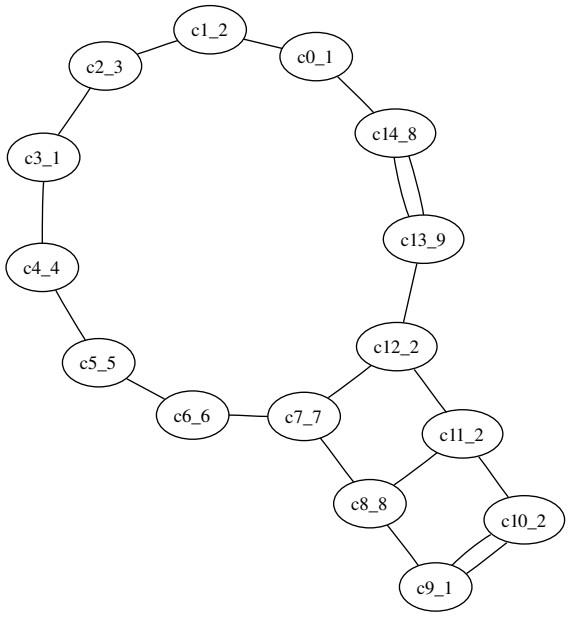

Figure 5.2 RNA Graph 1

The input RNA structures are taken from the Protein data bank(www.rcsb.org). The RNAs are pre-processed in order to find nodal parameters, the hydrogen and covalent bonds and the base pairs.

The vertex information contains the chain identifier, the vertex number, the first nodal parameters which are the torsion angles: eta and theta, the conformational code, the bin ascii code and the base identity respectively. The edge information contains the chain id, the vertex numbers between which the edges are formed and the i,j value for covalent bond and hydrogen bond orientation for non covalent bonds. The given input graph is transformed to the margin format. In order to assign a vertex label, the eta, theta values are binned which along with the other attributes forms one node labels. Vertices that have the same binned eta, theta values along with the same values for other attributes hence get assigned the same vertex label. In the case of edges, the hydrogen or covalent bonds along with their values pertain to the edge labels.

Experiments are run on a RNA database of 50 graphs. The average number of nodes in each graph in the database is 1500 and the average number of edges is 2750. The degree of each node is around two to three only. High support values varying from 70% to 95% are used as maximal frequent subgraphs

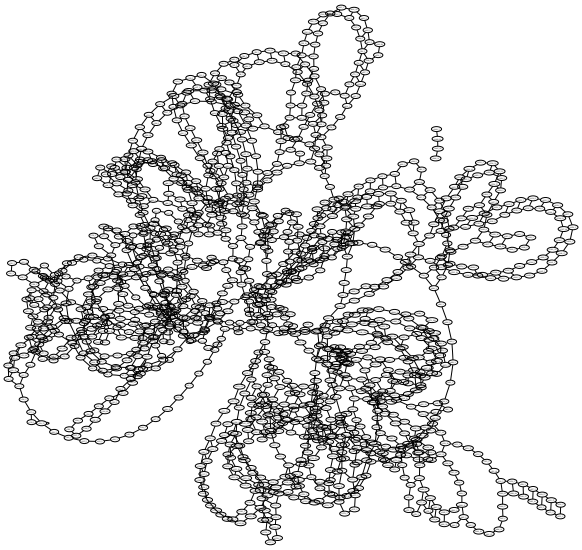

Figure 5.3 RNA Graph 2

form meaningful motifs only when they occur in a high number of RNA structures. Table 5.3 shows the running time for various support values. The maximal frequent subgraphs found are converted back into the original format. The scor database[5] contains information about the specific instances of motifs in the RNA structures. The maximal frequent subgraphs found are cross checked with the motifs in the scor database to validate whether the RNA motifs listed have been found. A hairpin loop is an unpaired loop of RNA motif that is created when an RNA strand folds and forms base pairs with another section of the same strand. The resulting structure looks like a loop or a U-shape. Most hairpin loops were identified some of which are:
Residue number 380-383 in structure 1hnx[5]
Residue number 459-463 in structure 1hnx[5]
Residue number 523-526 in structure 1hnx[5]

Support %	Number of Maximal	Time taken(mins)
95	75	156.2
90	191	209.22
85	149	206.47
80	157	215.3
75	168	222.00
70	173	232.60

Table 5.3 Running MARGIN on the RNA database

5.3 Real-life dataset

In this section we present experiments conducted with three real life datasets.

5.3.1 Page Views from msnbc.com

We tested the performance of MARGIN algorithm on a real-life data obtained from the UCI KDD archive [7]. The dataset corresponds to the records of page views by the users of msnbc.com on a particular day. Each record contains the sequence of categories of the pages viewed by a particular user. The dataset contains 9,89,818 records and the average number of page views per record is 5.7.

We processed the data by converting each record in the dataset into a graph as follows. The categories that belong to the record correspond to the vertices of the graph. Edges are created between every pair of categories of the same record. Thus each record is converted into a clique. We considered the graphs which had more than 70 edges. A total of 306 graphs with the following statistics were generated: the 13 minimum number of vertices, 17 maximum number of vertices, 13.9379 average number of vertices, 78 minimum number of edges, 136 maximum number of edges and 90.7549 average number of edges. Table 6.4 shows the results with different values of support. It can be observed that MARGIN algorithm reports results efficiently.

One of the maximal frequent subgraph that was reported for support value of 70 is the set of web page categories {"frontpage", "news", "weather", "sports"}. Thus, this set is one of the largest subset of web page categories that are visited together most frequently. The subset of categories can also help characterize the web users visiting them. This information can be exploited for providing better navigation mechanisms and personalizing the web pages that suit the user according to his/her interests.

Minimum Support	MARGIN	gSpan
50	693.66 sec	4864.23 sec
60	617.13 sec	3758.4 sec
70	545.61 sec	722.68 sec

Table 5.4 Running time on web data

5.3.2 Stock Market Data

We further present our results of stock market data of 20 companies collected from the source [8]. We use the correlation function below [62] to calculate the correlation between any pair of companies, A and B, C_B^A.

$$C_B^A = \frac{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(A^i \times B^i - \bar{A} \times \bar{B})}{\sigma_A \times \sigma_B}$$

Here, $|T|$ denotes the number of days in the period T, A^i and B^i denote the average price of the stocks on day i of the companies A and B respectively, and $\bar{A}$, $\bar{B}$, σ_A, and σ_B are defined as follows:

$$\bar{A} = \frac{1}{|T|}\Sigma_{i=1}^{|T|}A^i$$

$$\bar{B} = \frac{1}{|T|}\Sigma_{i=1}^{|T|}B^i$$

$$\sigma_A = \sqrt{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(A^i)^2 - \bar{A}^2}$$

$$\sigma_B = \sqrt{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(B^i)^2 - \bar{B}^2}$$

We construct a graph database where each graph in the database corresponds to seven successive working days (referred to as a week). For every week, we find the correlation values between every pair of companies. We also find the top ten companies with highest average stock price for the week. The nodes in the graph correspond to these top ten companies. An edge exists between any pair of nodes if their correlation value is positive. We construct 133 such graphs corresponding to the stock data of four years. Table 5.5 reports the time taken by Margin and gSpan. A maximally frequent subgraph refers to a set of cohesive companies that perform consistently through the time line. We found the average size of the maximal subgraphs to be 32 edges and hence lie in the higher region of the lattice. Therefore the time taken for Margin is seen to be lower than gSpan.

Support %	**MARGIN (sec)**	**gSpan (sec)**
2	0.48	6.7
4	0.56	6.64
6	0.62	6.61
8	0.68	6.56
10	0.74	6.56
12	0.81	6.58

Table 5.5 Comparison of Margin and gSpan on stock data

Support %	**MARGIN (sec)**	**gSpan (sec)**	**SPIN (sec)**
95	171.93	0.05	0.11
85	134.29	0.05	0.12
75	133.73	0.05	0.12
65	114.58	0.05	0.15
55	581.03	0.07	1.94
45	659.34	0.11	2.38

Table 5.6 Comparison using the Chem340 dataset

Support %	MARGIN (sec)	gSpan (sec)	SPIN (sec)
95	464.93	0.07	0.16
85	433.82	0.08	4.23
75	554.98	0.09	5.06
65	833.63	0.09	7.65
55	889.02	0.12	17.97
45	908.68	0.13	24.28

Table 5.7 Comparison using the Chem422 dataset

5.3.3 Chemical Compound Datasets

We further test the performance of MARGIN on the two chemical compound datasets used widely in gSpan and SPIN, referred to in [65] (available at [2]). We refer to them as Chem340 and Chem422 in this work. The datasets contain 340 and 422 graphs respectively. We eliminated disconnected graphs from these datasets and work with the remaining 327 and 328 graphs respectively. The running time of gSpan and SPIN are considerably better than MARGIN as seen in table 5.6 and 5.7. We list the time taken for support ranging from 95% to 15%. We also found the size of the maximal subgraphs generated for these varying supports. For the Chem340 (Chem422) dataset, the maximal subgraphs found were all single node (single edge) graphs for support values from 95% to 75%. With decreasing support from 65% to 15%, the size of the maximal subgraphs change from three-edge (two-edge) graphs to six-edge (eight-edge) graphs. The average size of graphs in Chem340 (Chem422) is 25 (36) edges. Hence, the maximals for most of the support values lie in the lower region of the lattice which is disadvantageous to MARGIN.

Further, the chemical compound datasets are very sparse where the edge-to-vertex ratio was found to be around 1.05. The discovered patterns are mostly tree-like structures. As per the discussion about the effect of edge-to-vertex ratio in section 4.2, it is expected that Apriori-based algorithms perform better for sparse datasets.

Though the average size of the graphs in Chem340 and Chem422 is 25 and 36 respectively, the largest graph is of size 214 and 196 respectively. Since MARGIN explores and stores the candidate subgraphs of such large graphs during the $ExpandCut$ operation, the memory utilization shoots up. Our present implementation does not support huge graphs and hence we provide results only till support value 45%.

Chapter 6

ISG:Mining maximal frequent graphs using itemsets

It is common to model complex data with the help of graphs consisting of nodes and edges that are often labeled to store additional information. The problem of maximal frequent subgraph mining can often be complex or simple based on the kind of input dataset. If the graph database is known to satisfy some constraint, like constraints on the size of graphs, the node or edge labels of graphs, the degree of nodes etc, it might be possible to solve the problem of subgraph mining without using any graph techniques and exploiting the properties of the constraint. Though the class of such graphs might be small, this problem is worth investigating since it could lead to immense reduction in the run time.

Given a set of graphs $\mathbb{D} = \{G_1, G_2, \ldots, G_n\}$, the support of a graph g is defined as the fraction of graphs in $\mathbb{D}$ in which g occurs. As defined earlier, g is frequent if its support is at least a user specified threshold. *In this chapter, we find maximal frequent subgraphs for a database of graphs having unique edge labels using itemset mining techniques*. Though itemset mining itself is exponential in the number of items, the complexity of graph mining algorithms rises steeply as against its itemset counterpart.

In itemset mining algorithms, an itemset can be generated exactly once by using a lexicographical order on the items. However, it is difficult to guarantee that a subgraph is generated only once during the process of frequent subgraph mining because due to the repetition of node and edge labels (i) a subgraph can be grown in several different ways by adding nodes or edges in different orders. Hence avoiding multiple explorations of the same graph adds to the overhead of graph mining algorithms which requires the use of canonical forms, (ii) the frequency of an itemset can be determined based on tid-list intersection of certain subitemsets. On the other hand, even the presence of all subgraphs of a graph g_1 in another graph g_2 does not guarantee the presence of the graph g_1 in g_2. Hence, detecting the presence of a subgraph in a graph requires additional operations, and (iii) subgraph isomorphism is NP-Hard. This makes it critical to minimize the number of subgraphs that need to be considered for finding the frequency counts. Other strategic methods such as information of the occurrences of the subgraph in the database that could avoid subgraph isomorphism have been developed. Finding whether an itemset is a subitemset of another is trivial as compared to determining whether a graph is a subgraph of another. Hence, our algorithm, ISG, finds the maximal frequent subgraphs by invoking a maximal itemset mining algorithm.

In the current literature on maximal frequent subgraph mining, standard Apriori techniques and their extensions were used to find the maximal frequent subgraphs. Such algorithms adopt a bottom up approach to find the final set of maximal frequent subgraphs. While SPIN [36] extended subtrees to finally construct maximal frequent subgraphs, MARGIN[58] found the n-size candidate subgraphs that have $n + 1$-size infrequent supergraphs. All variations of algorithms either need to perform subgraph isomorphism or do subgraph extensions.

6.1 Our Approach

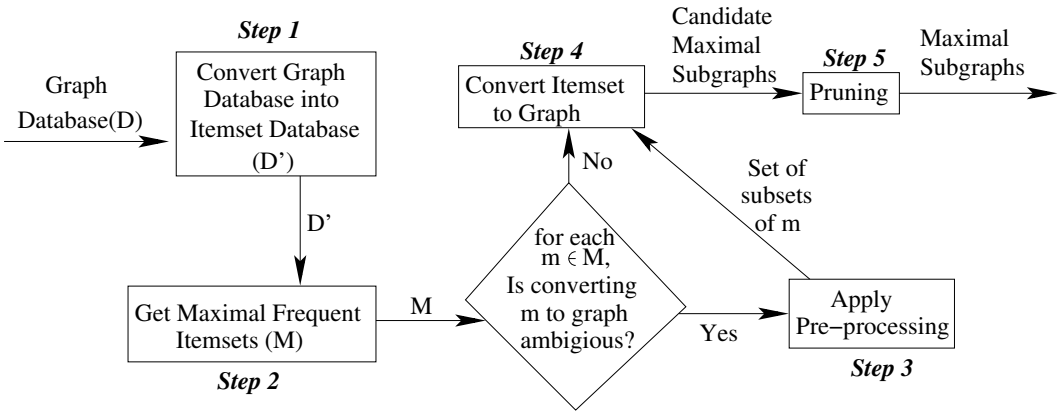

Figure 6.1 Overview of ISG Algorithm

Figure 6.1 shows the overview of the ISG algorithm. The entire algorithm can be broadly divided into five steps. Step 1 converts each graph in the database D into a transaction. This conversion step involves mapping parts of the graph to items and assigning a unique item id to each such item. If a graph g is converted into a transaction t which is a set of items then we should be able to reconstruct g using t. In order to ensure this the edges and "converse edges" (pair of adjacent edge labels) of the graph are mapped to items. Section 6.2.1, we show that there is a need for mapping of additional substructures of the graph into items in order to ensure unambiguous conversion of the itemsets into graphs. *These additional substructures are called secondary structures and are explained in detail in Section 6.2.1.* Thus, the transaction database, D' constructed contains the item ids of the edges, converse edges and the secondary structures.

Example: Consider a database of three graphs as in Figure 6.2 where G_1 is a triangle, G_2 is a spike and G_3 is a linear chain. Step 1 converts these graphs into the transaction database in D' shown in Figure 6.3.

In step 2, the transaction database D' that is constructed in step 1 is given as input to any maximal itemset mining algorithm and the set of maximal itemsets of D', denoted as M, is generated. In Figure 6.4, the transaction T_1 corresponds to graph G_1 of Figure 6.2, T_2 to G_2 and T_3 to G_3. For the transaction database of the example in Figure 6.4, the only maximal frequent itemset generated for support value

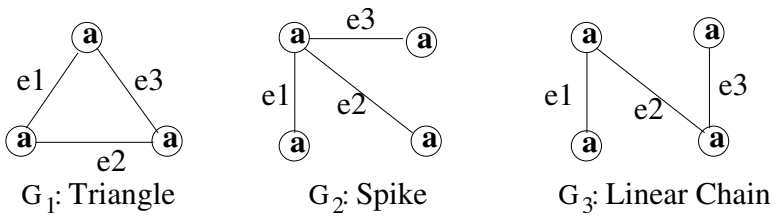

Figure 6.2 Example Graph Database

two is the set {1,2,3,51,52,53,100}. Each itemset $m \in M$ should be converted into a graph which is done in steps 3 and 4. If m can be unambiguously converted into a graph then step 3 is omitted and step 4 is invoked which generates the graph corresponding to the itemset m. The information corresponding to the mapping of the edges, converse edges and secondary structures to items is used in the process of converting itemsets to graphs (explained in section 6.2.2). On the other hand, if conversion of m into a graph is ambiguous, additional processing of step 3 is needed to resolve the ambiguity (more details in Section 6.2.1). In our example the maximal itemset m={1,2,3,51,52,53,100} is ambiguous and is hence preprocessed in step 3 and converted into a set of sub-itemsets m_{set}={{1,2,51}, {2,3,52}, {1,3,53}}. m_{set} will now be passed to step 4 which converts each subitemset into a graph as shown in Figure 6.3.

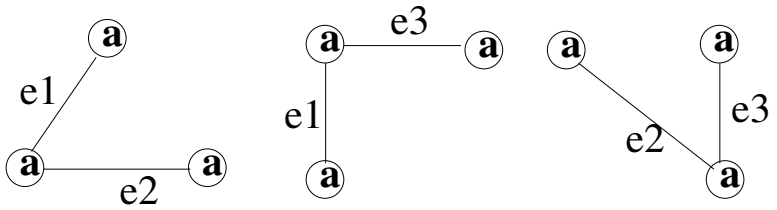

Figure 6.3 Maximal Frequent Subgraphs for the Example (Support=2)

T1	1, 2, 3, 51, 52, 53, 100, 150
T2	1, 2, 3, 51, 52, 53, 100, 200
T3	1, 2, 3, 51, 52, 100, 250

Figure 6.4 The Transaction Database

The output of Step 4 are candidate maximal frequent subgraphs which might require additional pruning in order to generate the maximal frequent subgraphs of D. Step 5 compares the graphs generated by the end of step 4 and prunes any graph which is a subgraph of any other candidate maximal frequent subgraph. Since no graph in our example in Figure 6.3 is a subgraph of another, the pruning in step 5 does not eliminate any graphs and the three graphs are given as output of the ISG algorithm.

6.2 The ISG Algorithm

Unique edge labeled graphs refer to the class of graphs where each edge label occurs at most once in each graph. Formally, we find maximal frequent subgraphs from a database of graphs $\mathbb{D} = \{G_1, G_2, \ldots, G_n\}$ where each graph G_i contains no two edges with the same label. Due to multiple occurrences of a node label, it is not sufficient to check the common node label to determine whether two edges are neighbors. In this section, we discuss the steps given in the Figure 6.1. The key steps in the algorithms are the conversion of graphs to itemsets in step 1 and conversion of maximal frequent itemsets back to graphs in steps 4 and 5.

6.2.1 Conversion of Graphs to Itemsets

This subsection explains step1 of the framework. Each edge is mapped to a unique triplet of the form (n_i,e,n_j) where e is the edge label of the edge incident on the nodes with labels n_i and n_j. Similarly, we define a *converse edge triplet* as a triplet of the form (e_i, n, e_j) where e_i and e_j are edges labels of two edges incident on a node with node label n. Converse edge triplets are essentially a triplet representation of every pair of adjacent edges. Each edge triplet and converse edge triplet, being unique is mapped to a unique item id. A graph G_i in the graph database is converted into an itemset transaction where the items are the item ids of the edge triplets and the item ids of the converse edge triplets of G_i.

The set of transactions so formed by the graphs in the database is now termed as the transaction database D'. The conversion of a graph to an itemset should be such that after finding the maximal frequent itemsets, they can be mapped back to graphs unambiguously. Considering only edge triplets does not guarantee that each maximal frequent itemset can be uniquely converted back to a graph. Even the edge triplets and converse edge triplets together do not guarantee the the given maximal frequent itemset be converted into a graph unambiguously. Consider Figure 6.2.

The edge triplets that correspond to the triangle and the spike structures are $\{(a, e1, a)$, $(a, e2, a)$ and $(a, e3, a)$ while the converse edge triplets are $(e1, a, e2)$, $(e1, a, e3)$ and $(e2, a, e3)\}$. The edge triplets that correspond to the linear chain are $\{(a, e1, a)$, $(a, e2, a)$ and $(a, e3, a)$ while the converse edge triplets are $(e1, a, e2)$ and $(e2, a, e3)\}$.

Case 1: The set of triplets corresponding to triangle and spike are the same. Hence, they cannot be uniquely converted back to a graph causing ambiguity.

Case 2: The maximal frequent itemset found for the transaction database of Figure 6.4 for support value three would be the set $\{(a, e1, a)$, $(a, e2, a)$, $(a, e3, a)$, $(e1, a, e2)$, $(e2, a, e3)\}$. This set of triplets is the same as the set of triplets for the linear chain as we saw above. However, we can observe that the linear chain is not frequent.

Thus, it can be concluded that the itemset database D' using only edge and converse edge triplets lead to ambiguity. Also, it can be trivially seen that there are only three structures exist with three edges as in Figure 6.2 of which the triangle and spike have the same set of edge and converse edge triplets. Also it is trivial to note that any two non-isomorphic graphs of n edges that have the same edge and

converse edge triplets should each have a $n-1$ edge substructure both of whose edge and converse edge triplets are the same. Let the edge with label e_i be the edge removed from both the n-edge graphs to create two $n-1$ edge graphs. The single edge triplet containing the edge label e_i and all converse edge triplets containing the edge label e_i will thus be removed. Since the edge and converse edge triplets of the n edge graphs was the same, the edge and converse edge triplets removed will also be the same set. Hence, the edge and converse edge triplets of the $n-1$ edge subgraph of each that remain would again be the same. Thus, it can be concluded further that both non-isomorphic n edge graphs with the same edge and converse edge triplets will have three edge substructures whose edge and converse edges are the same. The attempt is hence to remove the ambiguity at the level of three edge substructures so that the graphs of sizes greater than three can be represented and transformed non-ambiguously.

In order to handle the two cases of ambiguity mentioned earlier, we introduce three secondary structures called *triangle*, *spike* and *linear chain*. Figure 6.2 which we used as an example database shows each secondary structure. Note that the node labels of all the nodes in each secondary structure are the same. Hence, for a given graph, all substructures which are either a triangle, spike or linear chain, each having the same node labels form its secondary structure. The edge triplets and converse edge triplet form the primary structures.

Each secondary structure is assigned a unique item id in addition to the unique ids of the edge and the converse edge triplets. The concept of a *common item id* is also introduced which is a representation of a set of triplets irrespective of the structure it denotes. That is, if a spike and triangle have the same set of edge and converse edge triplets as in our example, then both are assigned the same common item id though their triangle and spike ids are different. Two triangles with different edge labels or different node label will have two different triangle ids. In case of the linear chain we make an exception. For a linear chain that contains the three edge triplets same as that of a triangle or spike, the linear chain is assigned the same common item id as that of the triangle or spike. Hence, the three structures in the Figure 6.2 would get the same common item id though their spike, triangle and linear chain id would be different. Since an edge label occurs at most once in a graph, each common item id present in an itemset can only correspond to exactly one of the three: triangle or spike or linear chain.

Definition 4 ***Conflicting Maximal Frequent itemset****: A maximal frequent itemset that contains a common item id without the presence of its secondary item id is called a conflicting maximal frequent itemset.*

Table 6.1 shows the id assigned to each edge, converse edge, common id and secondary structures for the graphs in Figure 6.2. The three edge triplets are given item ids 1, 2 and 3 while the converse edge triplets are given item ids 51, 52 and 53. The item id 150 represents the triangle, 200 the spike and 250 the linear chain. The item id 100 is the common item id which occurs for all the three structures of Figure 6.2. Using these item ids, the transaction database in Figure 6.4 is generated.

The graph G_1 representing a triangle will form the itemset {1,2,3,51,52,53,150,100} while the spike in G_2 of Figure 6.2 will form the itemset {1,2,3,51,52,53,200,100} and G_3 forms {1,2,3,51,52,250,100}

Type	ID	Set of Triplets
Edge Triplet	1	$(a, e1, a)$
Edge Triplet	2	$(a, e2, a)$
Edge Triplet	3	$(a, e3, a)$
Converse Edge Triplet	51	$(e1, a, e2)$
Converse Edge Triplet	52	$(e2, a, e3)$
Converse Edge Triplet	53	$(e1, a, e3)$
Common Id	100	$(a, e1, a)$, $(a, e2, a)$, $(a, e3, a)$, $(e1, a, e2)$, $(e1, a, e3)$, $(e2, a, e3)$
Triangle Id	150	$(a, e1, a)$, $(a, e2, a)$, $(a, e3, a)$, $(e1, a, e2)$, $(e1, a, e3)$, $(e2, a, e3)$
Spike Id	200	$(a, e1, a)$, $(a, e2, a)$, $(a, e3, a)$, $(e1, a, e2)$, $(e1, a, e3)$, $(e2, a, e3)$
Linear Chain Id	250	$(a, e1, a)$, $(a, e2, a)$, $(a, e3, a)$, $(e1, a, e2)$, $(e2, a, e3)$

Table 6.1 Mapping of edges of graphs in Figure 6.2 to unique item id

as seen in Figure 6.4. Consider finding the maximal frequent subgraphs for support value two. The six items $\{$1,2,3,51,52,53,100$\}$ that represent the edge and converse triplets of both the triangle and spike will show up as the maximal frequent itemset. However, neither the spike structure nor the triangle structure is maximally frequent. Such a maximal frequent itemset whose secondary item id is not present but the common id is present is defined above to be a conflicting maximal frequent itemset. There should be a way of indicating that neither the triangle nor the spike should be constructed as the three edge structures are infrequent but its two edge substructures are all frequent. Common item id is used for this purpose where the conflicting maximal frequent itemset indicates the need of preprocessing to avoid constructing either of the triangle or spike but include only the two edge substructures of both which are maximal frequent. Hence, if the maximal frequent itemset is a conflicting maximal frequent itemset then converting the itemset into graphs needs preprocessing in order to attach the three frequent two-edge substructures (if attachable) to the remaining constructed graph. This would form more than one possible graph out of the itemset. Step 3 of Figure 6.1, which preprocesses the conflicting itemset forms a set of itemsets such that each itemset can unambiguously create a graph. The itemset thus creates multiple graphs where each graph contains only one of the three possible two edge structures.

6.2.2 Conversion of Maximal Frequent Itemsets to Graphs

Once the maximal frequent itemsets are found in the previous phase, the corresponding candidate maximal frequent subgraphs are to be built. As mentioned earlier, the conflicting maximal frequent itemsets need to undergo a preprocessing phase in order to resolve the conflict that none of the three edge structures represented by the common item id is frequent while the common item id itself is frequent. This indicates that all three edges are frequent and occurring as neighbours to each other but they cannot

co-occur in a graph to form any three edge structure as each possible three edge structure itself is infrequent.

Preprocessing Phase: The preprocessing phase breaks the conflicting maximal frequent itemset to form non-conflicting subsets. It makes sure that the three edges that caused conflict are not present together in any subitemset created. We refer to the common item id whose secondary item ids are not present as the conflicting common item id of the conflicting maximal frequent itemset. The three edge labels present in the triplet set of each conflicting common item are marked as *Invalid Edge Combinations* since all the three cannot be present in a single graph. For example, as the common item with id 100 in the example database is conflicting, the edges $\{e1, e2, e3\}$ will form one set in the set of *Invalid Edge Combinations*. Since 100 corresponds to the the only conflicting common item id for the conflicting maximal frequent itemset {1,2,3,51,52,53,100}, being considered , *Invalid Edge Combinations* contains only one entry.

The preprocessing of the maximal frequent itemset that is conflicting begins by forming an initial set of components and recursively extending these components until they cannot be extended. The pair of edges that belong to each converse edge triplet of the maximal frequent itemset constitute the initial set of components. Thus, for the maximal frequent itemset {1,2,3,51,52,53,100} of our example database, 51, 52 and 53 form the three converse edge triplets and hence the initial set of components contains three edge pairs: $\{C_1 = (e1, e2)$, $C_2 = (e2, e3)$, $C_3 = (e1, e3)\}$.

These components are extended recursively. At each stage of the recursion, every pair of components are merged if the following two conditions are valid: (i) they both differ by a single element, (ii) on merging, they do not contain any edge combination that belongs to *Invalid Edge Combinations*. For the example maximal frequent itemset that is being considered, it can be seen that no two components can be merged as merging would produce a new component $C = (e1, e2, e3)$ which belongs to the *Invalid Edge Combinations*. All the edge label sets that can be merged are collected into $NewComponents_i$ where i denotes the i-th iteration of the merging step. Any components that could not be merged with any other components become a part of the final output as they cannot be extended further. The edge label sets in $NewComponent_i$ that can be merged are merged to form $NewComponent_{i+1}$ and those that cannot be merged with any set in $NewComponent_i$ are added to the final output. The merging process is repeated until no further merging is possible. In our running example, the $NewComponents_1$ set remains empty since each edge label set forming the components C_1, C_2 and C_3 cannot merge. Hence C_1, C_2 and C_3 form the final components. The set of edge label sets are returned as the output of the preprocessing phase.

After each conflicting maximal frequent itemset undergoes the preprocessing stage, each edge label set in the set of edge label sets reported is subjected to the graph construction algorithm. Maximal frequent itemsets that are not conflicting can be directly subjected to the graph construction phase. In the running example, the preprocessing phase returns three components thus forming three subitemsets. Each itemset corresponds to the edge triplets and converse edge triplets that constitute the components.

Thus, C_1 forms the subitemset $I_1 = \{1, 2, 51\}$ which corresponds to the triplet set $\{(a, e_1, a), (a, e_2, a), (e_1, a, e_2)\}$. Similarly $I_2 = \{2, 3, 52\}$ and $I_3 = \{1, 3, 53\}$ are formed.

Converting itemsets to graphs: We next describe how an maximal itemset can be converted into one or more unique set of graphs. If the maximal frequent itemset is a conflicting itemset then the pre-processing phase first converts it into a set of subitemsets. Each subitemset thus formed contains the maximum number of edge and converse edge triplets that can be included without causing conflict. Each such subitemset can be converted into a single unique graph. On the other hand, if the maximal itemset is non-conflicting then it might correspond to one or more connected graphs which can be seen as the components of a disconnected graph that can be constructed from it.

The basic idea of converting a maximal frequent itemset or subitemset into a graph is:
Step 1: Start with any edge triplet that is not visited so far and mark as visited. Construct a graph with the single edge picked in this step. In Figure 6.7(1), the edge triplet (a,e_1,b) is initially picked.
Step 2: Among the converse edge triplets that are not visited so far, find and mark as visited a converse edge triplet, if present, that contains the edge label of the edge picked in Step 1. Hence, in Figure 6.7, the converse edge triplet (e_1,b,e_2) forms a valid converse edge triplet that can be picked. Extend the graph formed in Step 1 by adding the new edge label seen in the converse edge triplet picked in this step as can be seen in Figure 6.7(2).
Step 3: Among the remaining unvisited edge triplets find and mark as visited the edge triplet that contains the edge label inserted in Step 2. Hence (b,e_2,c) is located among the edge triplets shown in the table in Figure 6.7. Label c is thus added to the unlabeled node in Figure 6.7(2) which forms Figure 6.7(3).
Step 4: Repeat steps 2 and 3 recursively until no more extension is possible.
Step 5: The graph so generated might contain unlabeled nodes. All edges incident on a node that is not labeled is dropped For example, if the maximal frequent itemset corresponds to only two triplets (n,e,n) and (e,n,e_1), then, out of the two nodes incident on e_1, we know that one node label is n while we do not have information about the node label of the other node. The edge e_1 is thus dropped and the graph formed would be a single edge graph with edge (n,e,n).
Step 6: Include the graph that is formed by the end of step 4 to the set of candidate maximal frequent subgraphs.
Step 7: Repeat from step 1 if there are any more unvisited edge triplets.

For a subitemset or for a maximal frequent itemset that corresponds to a single connected graph, all the edge triplets will be visited in one iteration of the algorithm. Conversely, if a maximal frequent itemset corresponds to a disconnected graph, then, in each iteration of the algorithm from step 1 to 6, one component of the disconnected graph will be generated.

The key to the above procedure are two steps: the extension of an edge triplet using converse edge triplet in step 2 and the extension of a converse edge triplet using an edge triplet in step 3. We explain Step 2 and Step 3 in detail next.

Extension of an edge triplet: Let (n_1, e, n_2) be an edge triplet in the graph constructed so far where the label n_1 is not same as n_2. If a converse edge triplet (e, n_1, e_1) is present in the itemset then (n_1, e, n_2) can be extended by adding the converse edge triplet on the n_1 node. As yet, we only have the information that an edge with label e_1 can be incident on the node with label n_1. The information about the opposite node label of the edge with label e_1 is not yet known. Extension of (n_1, e, n_2) using (e, n_1, e_1) will introduce an edge whose opposite node details are not known and hence is called a *dangling edge*. The unknown opposite node id is called a *dangling node*. The node label of the dangling node is assigned in step 3 in the procedure called "Extension of an converse edge triplet"(described earlier as Step 3). Step 3 is called on a pair say (e, n_1) for which it finds matching converse triplets of the form (e,n_1, *) and does the necessary extension in the graph. However, extending an edge triplet of the form (n, e, n) where the node labels of the both nodes are the same is complex. If a converse edge triplet (e, n, e_1) is present then the edge triplet (n, e, n) can be extended on any one of the two nodes incident on the edge, where each extension will lead to a different graph being constructed.

We discuss the details of the various cases when the node labels of the edge to be extended are the same. That is, when the edge to be extended is of the form (n,e,n). The below cases cover all possible states that the graph could be in while wanting to extend the edge of form (n,e,n). Let the nodes on which the edge with label e is incident be nodes A and B. Let the edge that we want to construct adjacent to the edge with edge label e be an edge with label e_1 due to the presence of the converse edge triplet (e, n, e_1). We use Figure 6.5 to illustrate the cases explained below.

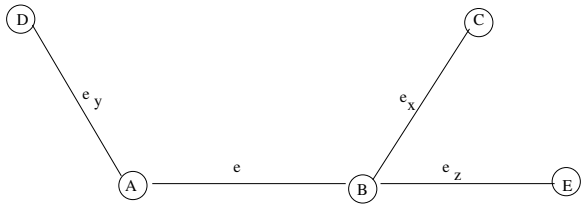

Figure 6.5 Cases in Edge Extension

Case 1: degree(A)=1 and degree(B)=1: This is the case when the graph contains only one edge so far which is the edge (A,e,B) itself. In such a case one can extend the edge with label e_1 on either of the sides.

Case 2: degree(A)=1 and degree(B)=2: Consider the Figure 6.5. Since in this case the degree(A)=1 and degree(B)=2 assume that the edge with label e_y and e_z do not exist in the figure. The algorithm first checks if an converse edge triplet of the form (e_1,n,e_x) exists. Suppose such a triplet does not exist in the maximal frequent itemset being considered, it is straightforward that we create a dangling edge on the node A. However, suppose such a converse edge triplet exists, then, we could make two constructions both of which could be correct. That is, we could add an edge with label e_1 connecting the nodes A and C forming a triangle if the node label of node C is also n or we could construct a dangling edge on the node B. In such a ambiguity, where node C also has label n, we check the maximal frequent itemset for whether a triangle or spike is present using edge labels e,e_1 and e_x. Only one of the two can be

present in any maximal frequent itemset, thus, resolving the ambiguity. In cases where the node labels are different, it is straightforward to construct a dangling edge on the node B and proceed to the step of "Extension of the Converse Edge Triplet"(step (3)).

Case 3: degree(A)=1 and degree(B)>2: The presence of any node with degree greater than two is trivial to resolve. Consider the Figure such that the edge e_y does not exist. The three edges incident on the node B are the edges with labels e, e_x and e_z. If the edge with label e_1 is to be incident on the node B, then all the three converse triplets (e_1,n,e), (e_1,n,e_x) and (e_1,n,e_Z) should be present in the set of converse edge triplets of the maximal frequent itemset being processed for construction. If all the three converse triplets are not present then the dangling edge with label e_1 is constructed on the node A.

Case 4: degree(A)=2 and degree(B)=1: This case is symmetric to the Case 1 and thus can be handled like Case 1.

Case 5: degree(A)=2 and degree(B)=2: Consider the figure such that the edge with label e_z does not exist as both nodes A and B are to have degree two. Consider the case when the converse edge triplets (e_1,n,e_x) and (e_1,n,e_y) are both present in the list of converse edge triplets. If the node labels of the nodes C and D are also n, it would imply that any one of the triangles (A,B,C) and (A,B,D) exist. This is resolved by referring to the triangle ids checking for the presence of either of edge labels (e,e_1,e_x) or (e,e_1,e_y). The other cases when the degree(A)=2 and degree(B)=1 are trivial to resolve.

Case 6: degree(A)=2 and degree(B)>2: As the degree of node B is greater than two, the case is trivial to resolve as in Case 3.

Case 7: degree(A)>2: As the degree of node A is greater than two, the case is trivial to resolve as in Case 3.

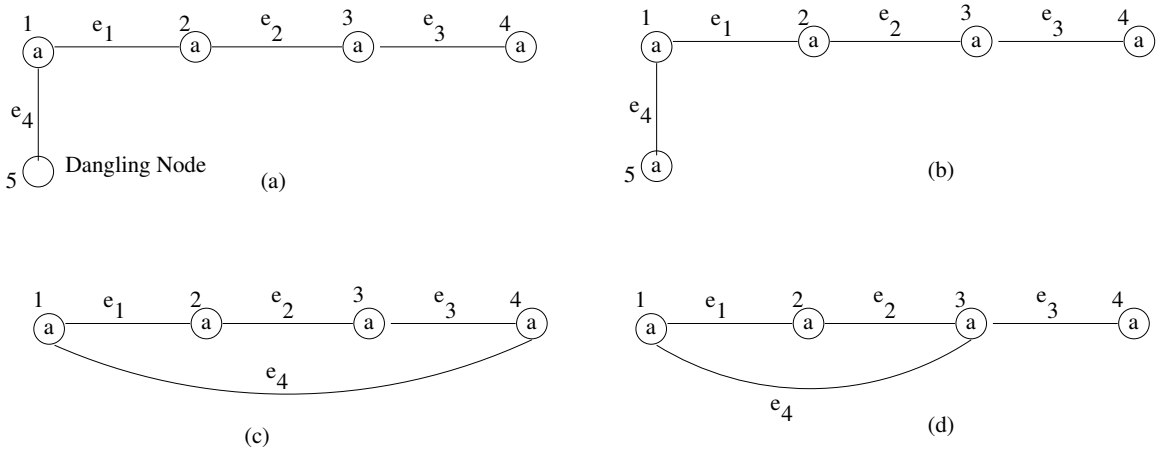

Figure 6.6 Extending the converse edge triplet

Extension of a converse edge triplet: In this step, the dangling edges that are introduced in Step 2 ("extension of a edge triplet" described above) are converted into regular edge triplets. Consider Figure 6.6(a). Say the edge with label e_4 was introduced in Step 2 last. In order to extend the converse edge triplet, step 3 looks for the unvisited edge triplet with edge label e_4. Say the edge triplet found was of

the form (a,e_4,a). This indicates that both nodes on which the edge label e_4 is to be incident should have edge label a one of which is already marked as a. The converse edge triplet extension function then needs to determine whether the dangling node is a node already existing in the graph constructed so far or whether a new node needs to be created. As in Figure 6.6(b), the edge e_4 could be connected to a new node which is node 5 which is labeled a or could be connected to an existing node as in Figure 6.6(c) or 6.6(d). In Figure 6.6(c) or 6.6(d) the edge e_4 on the dangling node is thus removed.

Generalising the above, let the converse edge triplet included last be (e,n,e_1) where the edge label e was introduced before the converse edge triplet extension. Edge label e_1 is thus constructed incident on the node with label n and a dangling node. In order to complete the construction of the edge label e_1 incident on the node with label n, one needs to know the opposite node id as well. The function hence determines the node label of the dangling node by looking at the edge triplet for the edge e_1. It is trivial to find the node label of the opposite node id from the set of edge triplets in the maximal itemset. Consider that the edge triplet (n,e_1,l_k) is present in the maximal itemset. We hence know that the node label of the opposite node is l_k. However, due to the possible multiple occurrences of nodes with the same label the function needs to determine whether a new node is to be created which is to be assigned the label l_k or which existing node with label l_k the edge with label e_1 is to be incident on. The function checks each existing node n_a with label l_k by picking any one edge incident on node n_a (say with label e_k) to check whether a converse edge triplet (e_1, l_k, e_k) exists. If such a triplet exists, an edge is inserted between node n_a and the node with label n. The dangling edge incident on the node with label n with edge label e_1 built in the edge triplet extension phase is dropped. If an edge cannot be formed with any of the existing nodes then the dangling node is marked with label l_k. Every edge triplet and converse edge triplet once visited is marked as visited and is not used for further construction.

Consider a node n_{p_1} in the graph constructed so far with node label l_k. Let the edge e_k incident on n_{p_1} have its opposite node as n_{p_2}. If the node label of n_{p_2} is also l_k then there is ambiguity about whether the edge with label e_1 is to be connected to n_{p_1} or n_{p_2}. This is because if the converse triplet (e_1, l_k, e_k) exists then it could be due to the triplets (e_1,label(n_{p_1}),e_k) or (e_1,label(n_{p_2}),e_k). Since a graph so far is connected, either of the nodes n_{p_1} or n_{p_2} has to have degree greater than one. Let n_{p_1} have degree greater than one. This would mean that there exists another edge with label say e_p incident on n_{p_1} apart from the edge with label e_k. If the edge e_1 is to be incident on n_{p_1} and not n_{p_2} then the converse edge triplet (e_p, label(n_{p_1}),e_1) should also exist apart from the converse edge triplet (e_k, label(n_{p_1}),e_1). If such a triplet does not exist then the edge e_1 should be incident on the node n_{p_2}. Also, the simple check to avoid self loops is applied during $EdgeExtension$.

6.2.3 Pruning Phase

Given two maximal frequent itemsets M_i and M_j, both M_i and M_j might yield a connected or a disconnected graph. One component of the graph generated by M_i might be a subgraph of the graph generated by M_j or one of the components of the graph generated by M_j. Hence not all subgraphs generated in the previous phase are maximal frequent subgraphs. Hence an additional simple pruning phase

is needed. Using subgraph isomorphism is one way of determining whether a graph g is a subgraph of another graph g'. However, in ISG, we do away with the subgraph isomorphic operation as it is possible to determine whether a graph is a subgraph of another using a subset operation. Consider two graphs g and g'. Let I be the set of item ids of the edge and converse edge triplets in g. Let I' be the set of item ids of the edge and converse edge triplets in g'. If g is a subgraph of g', I has to be a subset of I'. However, I being a subset of I' does not guarantee that g is a subgraph of g'. This is because, edge and converse edge triplets alone cannot disambiguate if g has a spike using the triplets while g' has a triangle using the same triplets. Hence, here again, we retain for each candidate maximal frequent subgraph formed a corresponding itemset of its edge and converse edge triplets along with the triangle, spike and linear chain ids present in it. It can be trivially seen that for such an itemset constructed, I being subset of I' guarantees that g is a subgraph of g'. It is trivial to pass on the triangle, spike and linear chain ids already calculated and present in each maximal frequent itemsets that were used for graph construction.

6.2.4 Illustration of an Example

Figure 6.7 illustrates a simple run of the algorithm. Listed are the edge and converse edge triplets of a maximal frequent itemset and the stages in which the graph is extended for ease of understanding. The three edge triplets are (a, e_1, b),(a, e_3, c),(b, e_2, c) and the three converse edge triplets are (e_1, b, e_2),(e_1, a, e_3) and (e_2, c, e_3). In this example we assume that there was no common item id, spike id, triangle id or linear chain id for simplicity.

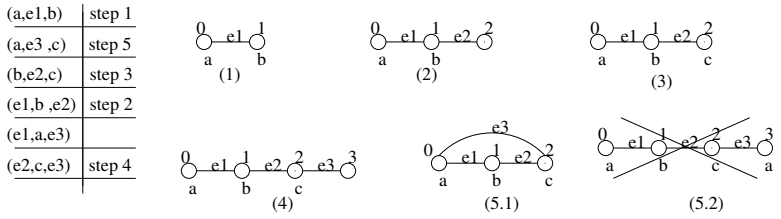

Figure 6.7 Running Example of the ISG algorithm

Figure 6.7(1) shows the initial edge picked. The graph is created with two nodes, node 0 with label a, node 1 with label b and edge label e_1. The edge triplet (a, e_1, b) is marked in the adjacent table as visited in step1. In Figure 6.7(2), $ConverseExtension$ looks for the extensions of the edge created in Figure 6.7(1). All converse edges of the form $(e_1, label(1), *)$ =$(e_1, b, *)$ are searched for among the converse edge triplets. The triplet (e_1, b, e_2) matches the search. Hence a new adjacent edge is created with edge label e_2. While the node label of incident node 1 is known, the label of the other node 2 is not known. We call node 2 a dangling node and edge with label e_2 as a dangling edge. $EdgeExtension$ now looks for edge triplet of the form $(label(1), e_2, *)$=$(b, e_2, *)$ in order to find the label of the dangling node. Label c which matches the search is updated in the graph as in Figure 6.7(3). Next, we look for converse edge triplet extensions of the form $(e_2, label(2), *)$= $(e_2, c, *)$ and extend the graph to contain edge with label e_3 as seen in Figure 6.7(4). On looking for edge triplets of the form $(label(2), e_3, *)$=$(c, e_3, *)$, the

triplet (a, e_3, c) qualifies. Since the node label a already exists in the graph, the edge with e_3 as label could create extensions of two forms. The first one is shown in 6.7(5.1) where it connects to an existing node with label a and the second one shown in 6.7(5.2) where it creates a new node with label a.

For Figure 6.7(5.1) to be valid, the converse triplet (e_1, a, e_3) should be present because edges with label e_1 and e_3 now become neighbours. For Figure 6.7(5.2) to be valid, the converse triplet (e_1, a, e_3) should not be present. Since converse edge triplet (e_1, a, e_3) exists, the dangling edge on node 2 is dropped and hence 6.7(5.1) qualifies as the valid graph. We next look for extensions among converse edge triplets of the form $(e_3, a, *)$. (e_1, a, e_3) is already introduced in the graph. Hence no match exists.

We had initially started with converse edge triplet extensions of the form $(e_1, label(1), *)$. We next look for converse edge triplet extensions of the form $(e_1, label(0), *)$. The only match being (e_1, a, e_3) is already introduced in the graph. As we cannot extend further, we terminate by producing the graph in 6.7(5.1). Since all edge triplets have been visited, we also note that the maximal frequent subgraph generated is connected. Else, we would need to restart the process with an unvisited edge triplet to construct other components of the disconnected graph being generated by the itemset. Note that alternating between searching for converse edge and edge triplets, we extend the graph until no more extension is possible or no unseen edge triplets exist.

6.2.5 Proof of Correctness

We next show that the algorithm gives all maximal frequent subgraphs. That is, every subgraph reported is maximal frequent and there is no maximal frequent that is unreported.

Claim 2 *All maximal frequent subgraphs are reported.*

Proof: Consider a maximal frequent subgraph m_i. The set of edge and converse edge triplets that form the maximal frequent subgraph is thus frequent and has to be a subset of some maximal frequent itemset MI_i reported in the itemset mining phase. Let the disconnected or connected graph corresponding to the itemset MI_i be GI_i. The connected graph GI_i or the subgraph of some component of GI_i has to correspond to the maximal frequent subgraph m_i. Since m_i is maximal, m_i can only be an improper subgraph of GI_i or some component of it which means the connected GI_i or the a component of GI_i itself corresponds to m_i. After the accumulation of all the possible candidate maximal frequent subgraphs, the pruning phase prunes this set by removing all graphs that are subgraphs of another graph in this set. A maximal frequent subgraph by definition will not get pruned. Hence, the set report contains all maximal frequent subgraphs. □

Claim 3 *There is no graph reported that is not a maximal frequent subgraph.*

Proof: A graph that is not maximal frequent gets pruned out in the pruning phase as there has to exist some subgraph from the set of maximal frequent subgraphs which is supergraph to it. □

From the above two claims it can be concluded that the set of graphs reported is the set of maximal frequent subgraphs of the original dataset $\mathbb{D}$.

6.3 Results

L=20, E=20, V=20, I=20, T=22					
D	**Supp**	**ISG(sec)**	**GS(sec)**	**Spin(sec)**	**MR(sec)**
D=100	30	0.04	43.55	121.23	2.37
	20	0.07	86.77	164.96	1.3
	10	0.07	88.09	176.24	0.38
	5	0.07	88	176.83	0.38
$D = 500$	30	0.14	14.83	277.66	2.76
	20	0.28	17.05	581.41	10.06
	10	0.16	31.44	488.15	2.4
	5	0.15	31.26	487.71	2.22
$D =$ 1000	30	0.29	17.34	581.16	10.69
	20	0.16	31.33	487.75	2.64
	10	0.3	20.08	650.77	6.44
	5	0.3	20.2	649.17	5.4

Table 6.2 Results with varying D

D=500, L=20, E=20, V=20					
I,T	**Supp**	**ISG(sec)**	**GS(sec)**	**Spin(sec)**	**MR(sec)**
I=10,T=15	30	0.08	0.15	3.22	9.83
	20	0.09	0.4	6.69	16.51
	10	0.1	0.47	8.5	13.75
	5	0.11	0.72	12.25	5.96
I=15,T=20	30	0.13	0.04	0.11	35.72
	20	0.14	5.18	60.13	39.67
	10	0.17	19.48	131.25	38.37
	5	0.2	29.39	247.59	12.89
I=20,T=25	30	0.18	114.68	1057.53	33.53
	20	0.2	138.69	1304.07	16.08
	10	0.23	352.03	2002.18	13.78
	5	0.24	350.52	2005.68	17.18

Table 6.3 Results with varying I and T

We implemented the ISG algorithm and tested it on both synthetic and real-life datasets. We ran our experiments on a 1.8GHz Intel Pentium IV PC with 1 GB of RAM, running Fedora Core 4. The code is implemented in C++ using STL, Graph Template Library[1] and the IGraph Library [2].

[1] http://www.infosun.fmi.uni-passau.de/GTL

[2] http://igraph.sourceforge.net

Supp	ISG(sec)	GSpan(sec)
3	0.19	0.33
2.5	0.2	0.45
2	0.19	0.75
1.5	0.17	1.01
1	0.13	2.29
0.5	0.1	4.43

Table 6.4 Results on Stocks Data

6.3.1 Results on Synthetic Datasets

Synthetic Data Generation: We generated the synthetic datasets using the graph generator software provided by [47]. The graph generator generates the datasets based on the six parameters: D (the number of graphs), E,V (the number of distinct edge and vertex labels respectively), T (the average size of each graph), I (the average size of frequent graphs) and L (the number of frequent patterns as frequent graphs). In a post-processing step, the edge labels of each graph are randomly modified such that the graph satisfies the unique edge label constraint.

We compared the time taken by ISG algorithm with two other maximal frequent subgraph mining algorithms Spin [36] and MARGIN [58] and one frequent subgraph mining algorithm gSpan [65]. We used our implementations of SPIN and MARGIN while we used the executable provided by the authors of gSpan. While SPIN and MARGIN generate the maximal subgraphs, gSpan outputs all the frequent subgraphs. In a post-processing step, we compare the frequent subgraphs generated by gSpan and find the maximal frequent subgraphs.

Table 6.2 shows results generated when the number of graphs are taken as 100, 500 and 1000. Other parameters used for the generation of the dataset are L=20, E=20, V=20, I=20 and T=22. We can see that the ISG algorithm performs orders of magnitude better than the rest of the approaches.

Table 6.3 shows the comparison of running time when D, L, E and V are kept constant at 500, 20, 20, 20 and 20 respectively while I and T are varied between 10 and 25. It can be observed that the performance of ISG improves drastically for higher values of I and T.

6.3.2 Results on Real-life Dataset

We further present our results on stock market data of 20 companies collected from the source [3]. We use the correlation function below [62] to calculate the correlation between any pair of companies, A and B.

$[C_B^A = \frac{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(A^i \times B^i - \bar{A} \times \bar{B})}{\sigma_A \times \sigma_B}]$ where $|T|$ denotes the number of days in the period T, A^i and B^i denote the average price of the stocks on day i of the companies A and B respectively, and $\bar{A} = \frac{1}{|T|}\Sigma_{i=1}^{|T|}A^i$, $\bar{B} = \frac{1}{|T|}\Sigma_{i=1}^{|T|}B^i$, $\sigma_A = \sqrt{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(A^i)^2 - \bar{A}^2}$ and $\sigma_B = \sqrt{\frac{1}{|T|}\Sigma_{i=1}^{|T|}(B^i)^2 - \bar{B}^2}$.

[3] Yahoo Finance: http://finance.yahoo.com

We construct a graph database where each graph in the database corresponds to seven successive working days (referred to as a week). For every week, we find the correlation values between every pair of companies. The companies are clustered into a specific number of groups based on their average stock value during that week. Each group is assigned a unique label which corresponds to the node label in the graph. The correlation values between every pair of companies are also ordered and top $K\%$ values are ranked and the value of the rank is used as edge label. Thus, the graph corresponding to each week will have unique edge labels.

We collected graphs corresponding to top 20 companies for 295 weeks. The top 20 companies are selected for each week based on their average share price during that week. Further, the companies are classified as high and low categories which are used as node labels. The correlation values are calculated for every pair of companies and the edge labels correspond to the rank of the value. Table 6.4 show the comparison of ISG algorithm with gSpan for this dataset. For low support values, ISG performs better than gSpan. The time values for other two algorithms, Spin and MARGIN could not be generated as they did not terminate even after running for very long time interval because of very low support value.

Chapter 7

Conclusions

Many applications require the computation of maximal frequent subgraphs such as mining contact maps [34], finding maximal frequent patterns in metabolic pathways [45], and finding set of large cohesive web pages. As an example, consider the web pages that are related. These can be determined based on the information about the set of web pages that are visited by users in one session. Each sequence of pages visited can be modeled as a graph. Given a set of such graphs, the cumulative behavior of the web users can be mined by finding the maximal frequent subgraphs. Each maximal frequent subgraph might denote a cohesive set of topics that are of interest to most of the visitors of the website. Further, given a database of chemical compounds, maximal frequent substructures may provide insight into the behavior of the molecule. For example, researchers exploring immunodeficiency viruses may gain insight into recombinants via commonalities within initial replicants. Recent interest in functional RNAs, spurred by a surge in the number of published X-ray structures, has highlighted the importance of characterization of recurrent structural motifs and their roles. Maximal subgraph mining methods can be applied on proteins to predict folding sequences and also maximal motifs in RNAs.

In this thesis, we propose the $MARGIN$ algorithm for maximal frequent subgraph mining for a graph database. Further, if the graph database has graphs with unique edge labels, itemset mining techniques can replace the expensive graph mining techniques. We propose the ISG algorithm for maximal frequent subgraph mining of graphs with unique edge labels by using itemset mining techniques.

Frequent subgraph mining faces several challenges. For a graph of e edges, the number of possible frequent subgraphs can grow exponentially in e. Further, it is critical to reduce the number of subgraphs for which frequency computation is required since doing so requires the NP-complete subgraph isomorphism operation to be applied. A typical approach to frequent subgraph mining problem has been to find the frequent subgraphs incrementally in an Apriori manner. The Apriori based approach has been further modified to suit maximal subgraph mining [36] with added pruning and hence faces challenges that are similar to those of frequent subgraph mining.

However, the size of the set of maximal frequent subgraphs is significantly smaller than that of the set of frequent subgraphs [36], thus, providing scope for ample pruning of the exponentially large search space. The set of candidate subgraphs which are likely to be maximally frequent are the set of n-edge

frequent subgraphs that have a $n+1$-edge infrequent super graph. We refer to such a set of nodes in the lattice as the set of $f(\dagger)$-$nodes$. The MARGIN algorithm computes such a candidate set efficiently. By a post-processing step MARGIN finds all maximally frequent subgraphs. The $ExpandCut$ step invoked within the MARGIN algorithm recursively finds the candidate subgraphs. The search space of Apriori based algorithms corresponds to the region below the $f(\dagger)$-$nodes$ all the way from the bottom of the lattice till the maximal frequent subgraphs. On the other hand, the search space of MARGIN is limited to the region in the lattice around the $f(\dagger)$-$nodes$.

We prove that the $MARGIN$ algorithm finds all the nodes that belong to the candidate set. We show that any two candidate nodes are reachable from one another using the $ExpandCut$ function used to explore the candidate nodes. Given that the graph lattice is dense and there exists several paths that connect two candidate nodes, one of the main challenges of the proof is the abstraction of the sublattice that contains the path taken by the $ExpandCut$ function. By the construction of such a sublattice we show that such a sublattice is guaranteed to exist. A reachability sequence is then shown that starting at any candidate node would reach another candidate node.

If the graph database has graphs with constraints such as, graphs with unique edge or node labels, itemset mining techniques can be used to efficiently find maximal frequent subgraphs. The set of edges of each graph can be uniquely mapped to an itemset thus mapping the graphs database into a transaction database. The maximal frequent itemsets of the transaction database are computed using any standard maximal frequent itemset mining algorithm. The conversion of maximal itemsets into graphs is non trivial as one maximal itemset might lead to several candidate maximal graphs. The ISG algorithm efficiently preprocesses each maximal itemset to generate a set of itemsets each of which can be unambiguously converted into candidate maximal graphs.

The main contributions of the work are: (i) A novel algorithm $MARGIN$ to find maximal frequent subgraphs is presented, (ii) The $ExpandCut$ technique used in $MARGIN$ need not be limited to maximal frequent subgraph mining. A generalisation of the $MARGIN$ algorithm that can be applied to a wider range of subgraph mining problems is shown. A set of properties are presented, having which, a problem can be solved using the $ExpandCut$ technique. (iii) The proof of the $MARGIN$ algorithm is given, (iv) Experimental results on both synthetic and real-life data sets are presented and the viability of the $MARGIN$ algorithm in efficiently finding maximal frequent subgraphs is shown and compared with known standard techniques like gSpan, SPIN and CloseGraph. Experimental analysis has shown that MARGIN performs well for databases where the frequent graph patterns are large and for lower support values. However, MARGIN suffers from the disadvantage of using more memory than existing algorithms for maximal frequent subgraph mining, (v) the ISG algorithm is proposed that efficiently mines maximal frequent subgraphs in a constrained database by using itemset mining techniques, (vi) experimental results are shown that compare using graph mining techniques as against the ISG algorithm for graphs with unique edge labels.

The future work consists of (i) Studying the effect of the initial cut selected at the runtime of the $MARGIN$ algorithm. The efficiency of the $MARGIN$ algorithm depends on the initial cut selected.

By selecting the appropriate initial cut efficiently, it might be possible to reduce the number of revisited cuts. Further, a number of initial cuts can be introduced and parallely processed in order to make the process completion faster. (ii) Developing a partition based modification to the MARGIN algorithm. Due to memory constraints, it is of great interest to partition the graphs onto several machines and then apply a modified $MARGIN$ approach to find maximal frequent subgraphs over a distributed graph database. (iii) Extending the $MARGIN$ and the ISG algorithm to incrementally compute the maximal frequent subgraphs in the presence of updates to the graph database. (iv) Finding graph databases with constraints other than unique edge labels, and their corresponding algorithms that benefit by methods other than those typically used in graph mining.

Bibliography

[1] The graph template library. http://www.infosun.fmi.uni-passau.de/GTL/.

[2] gspan software. http://www.xifengyan.net/software/gSpan.htm.

[3] The igraph library. http://igraph.sourceforge.net/.

[4] Libgtop. http://library.gnome.org/devel/libgtop/stable/libgtop-GlibTop.html.

[5] The scor database. http://scor.berkeley.edu/.

[6] Top. http://en.wikipedia.org/wiki/Top_(Unix).

[7] The uci kdd archive: msnbc.com anonymous web data. http://kdd.ics.uci.edu/databases/msnbc/msnbc.html.

[8] Yahoo finance. http://finance.yahoo.com/.

[9] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. pages 207–216. SIGMOD, 1993.

[10] R. Agrawal and J. C. Shafer. Parallel mining of association rules. volume 8, pages 962–969. IEEE Trans. Knowl. Data Eng, 1996.

[11] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. pages 487–499. VLDB, 1994.

[12] R. Agrawal and R. Srikant. Mining sequential patterns. pages 3–14. ICDE, 1995.

[13] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. pages 51–58. ICDM, 2002.

[14] G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive parallel graph mining for cmp architectures. pages 97–106. ICDM, 2006.

[15] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. pages 443–452. ICDE, 2001.

[16] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. volume 2, pages 742–753. PVLDB, 2009.

[17] D. W.-L. Cheung, J. Han, V. T. Y. Ng, A. W.-C. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. pages 31–42. PDIS, 1996.

[18] D. W.-L. Cheung, J. Han, V. T. Y. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. pages 106–114. ICDE, 1996.

[19] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok. Frequent subtree mining - an overview. pages 66(1–2): 161–198. Fundam. Inform., 2005.

[20] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. volume 17, pages 190–202. IEEE Trans. Knowl. Data Eng., 2005.

[21] Y. Chi, Y. Yang, and R. R. Muntz. Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. pages 11–20. SSDBM, 2004.

[22] M. Cohen and E. Gudes. Diagonally subgraphs pattern mining. pages 51–58. DMKD, 2004.

[23] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. volume 1, pages 231–255. J. Artif. Intell. Res., 1994.

[24] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. volume 3, pages 7–36. Data Min. Knowl. Discov., 1999.

[25] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. pages 30–36. KDD, 1998.

[26] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. pages 35–42. ICDM, 2003.

[27] W. Fan, K. Zhang, H. Cheng, J. Gao, X. Yan, J. Han, P. S. Yu, and O. Verscheure. Direct mining of discriminative and essential frequent patterns via model-based search tree. pages 230–238. KDD, 2008.

[28] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. pages 163–170. ICDM, 2001.

[29] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharm. Discovering all most specific sentences. pages 28(2): 140–174. ACM Trans. Database Syst, 2003.

[30] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. pages 1–12. SIGMOD, 2000.

[31] M. A. Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: Mining representative orthogonal graph patterns. pages 153–162. ICDM, 2007.

[32] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. page 38. ICDE, 2006.

[33] M. Holsheimer, M. L. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. pages 150–155. KDD, 1995.

[34] J. Hu, X. Shen, Y. Shao, C. Bystroff, and M. J. Zaki. Mining protein contact maps. pages 3–10. BIOKDD, 2002.

[35] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. pages 549–552. ICDM, 2003.

[36] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. pages 581–586. KDD, 2004.

[37] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. pages 13–23. PKDD, 2000.

[38] A. Inokuchi, T. Washio, H. Motoda, K. Kumasawa, and N. Arai. Basket analysis for graph structured data. pages 420–431. PAKDD, 1999.

[39] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. pages 566–575. ICDE, 2007.

[40] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering frequent topological structures from graph datasets. pages 606–611. KDD, 2005.

[41] R. J. B. Jr. Efficiently mining long patterns from databases. pages 85–93. SIGMOD, 1998.

[42] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. pages 129–140. ICDE, 2002.

[43] Y. Ke, J. Cheng, and W. Ng. Correlation search in graph databases. pages 390–399. KDD, 2007.

[44] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. pages 1–17. COCOON, 1999.

[45] M. Koyuturk, A. Grama, and W. Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. pages 200–207. ISMB, 2004.

[46] M. Koyuturk, A. Grama, and W. Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. pages 200–207. ICMB/ECCB(Suplement of Bioinformatics, 2004.

[47] M. Kuramochi and G. Karypis. Frequent subgraph discovery. pages 313–320. ICDM, 2001.

[48] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. pages 258–265. ICDM, 2002.

[49] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. volume 11, pages 243–271. Data Min. Knowl. Discov., 2005.

[50] S. Nijssen and J. Kok. The gaston tool for frequent subgraph mining. International Workshop on Graph-Based Tools, October 2, 2004.

[51] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. pages 175–186. SIGMOD, 1995.

[52] J. S. Park, M.-S. Chen, and P. S. Yu. Efficient parallel and data mining for association rules. pages 31–36. CIKM, 1995.

[53] J. Pei, G. Dong, W. Zou, and J. Han. On computing condensed frequent pattern bases. pages 378–385. ICDM, 2002.

[54] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. pages 21–30. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000.

[55] G. Ramesh, W. Maniatty, and M. J. Zaki. Feasible itemset distributions in data mining: theory and application. pages 284–295. PODS, 2003.

[56] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. pages 432–444. VLDB, 1995.

[57] S. Srinivasa and L. BalaSundaraRaman. A filtration based technique for mining maximal common subgraphs. Technical Report 0303, International Institute of Information Technology, Bangalore, 2003.

[58] L. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. pages 1097–1101. ICDM, 2006.

[59] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. pages 737–746. KDD, 2007.

[60] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. pages 458–465. ICDM, 2002.

[61] J. Wang, W. Hsu, M.-L. Lee, and C. Sheng. A partition-based approach to graph mining. page 74. ICDE, 2006.

[62] J. Wang, Z. Zeng, and L. Zhou. Clan: An algorithm for mining closed cliques from large dense graph databases. page 73. ICDE, 2006.

[63] T. Washio and H. Motoda. State of the art of graph-based data mining. volume 5, pages 59–68. SIGKDD Explorations, 2003.

[64] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. pages 976–985. ICDE, 2007.

[65] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. pages 721–724. ICDM, 2002.

[66] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. pages 286–295. KDD, 2003.

[67] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. pages 335–346. SIGMOD, 2004.

[68] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. pages 344–353. KDD, 2004.

[69] K. Yoshida, H. Motoda, and N. Indurkhya. Graph-based induction as a unified learning framework. volume 4, pages 297–328. J. of Applied Intel., 1994.

[70] M. J. Zaki. Scalable algorithms for association mining. volume 12, pages 372–390. IEEE Trans. Knowl. Data Eng, 2000.

[71] M. J. Zaki. Efficiently mining frequent trees in a forest. pages 71–80. KDD, 2002.

[72] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. SDM, 2002.

[73] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. volume 1, pages 343–373. Data Min. Knowl. Discov, 1997.

[74] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. page 2006. KDD, 797-802.

[75] S. Zhang, M. Hu, and J. Yang. Treepi: A new graph indexing method. pages 966–975. ICDE, 2007.

[76] Z. Zou, H. Gao, and J. Li.

[77] Z. Zou, J. Li, H. Gao, and S. Zhang. Mining frequent subgraph patterns from uncertain graph data. volume 22, pages 1203–1218. IEEE Trans. Knowl. Data Eng, 2010.