Chapter 12

MINING GRAPH PATTERNS

Hong Cheng
Department of Systems Engineering and Engineering Management
Chinese University of Hong Kong
hcheng@se.cuhk.edu.hk

Xifeng Yan
Department of Computer Science
University of California at Santa Barbara
xyan@cs.ucsb.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
hanj@cs.uiuc.edu

Abstract Graph pattern mining becomes increasingly crucial to applications in a variety of domains including bioinformatics, cheminformatics, social network analysis, computer vision and multimedia. In this chapter, we first examine the existing frequent subgraph mining algorithms and discuss their computational bottleneck. Then we introduce recent studies on mining significant and representative subgraph patterns. These new mining algorithms represent the state-of-the-art graph mining techniques: they not only avoid the exponential size of mining result, but also improve the applicability of graph patterns significantly.

Keywords: Apriori, frequent subgraph, graph pattern, significant pattern, representative pattern

C.C. Aggarwal and H. Wang (eds.), *Managing and Mining Graph Data*,
Advances in Database Systems 40, DOI 10.1007/978-1-4419-6045-0_12,

1. Introduction

Frequent pattern mining has been a focused theme in data mining research for over a decade. Abundant literature has been dedicated to this research area and tremendous progress has been made, including efficient and scalable algorithms for frequent itemset mining, frequent sequential pattern mining, frequent subgraph mining, as well as their broad applications.

Frequent graph patterns are subgraphs that are found from a collection of graphs or a single massive graph with a frequency no less than a user-specified support threshold. Frequent subgraphs are useful at characterizing graph sets, discriminating different groups of graphs, classifying and clustering graphs, and building graph indices. Borgelt and Berthold [2] illustrated the discovery of active chemical structures in an HIV-screening dataset by contrasting the support of frequent graphs between different classes. Deshpande et al. [7] used frequent structures as features to classify chemical compounds. Huan et al. [13] successfully applied the frequent graph mining technique to study protein structural families. Frequent graph patterns were also used as indexing features by Yan et al. [35] to perform fast graph search. Their method outperforms the traditional path-based indexing approach significantly. Koyuturk et al. [18] proposed a method to detect frequent subgraphs in biological networks, where considerably large frequent sub-pathways in metabolic networks are observed.

In this chapter, we will first review the existing graph pattern mining methods and identify the combinatorial explosion problem in these methods – the graph pattern search space grows exponentially with the pattern size. It causes two serious problems: (1) the computational bottleneck, *i.e.*, it takes very long, or even forever, for the algorithms to complete the mining process, and (2) patterns' applicability, *i.e.*, the huge mining result set hinders the potential usage of graph patterns in many real-life applications. We will then introduce scalable graph pattern mining paradigms which mine *significant* subgraphs [19, 11, 27, 25, 31, 24] and *representative* subgraphs [10].

2. Frequent Subgraph Mining

2.1 Problem Definition

The vertex set of a graph g is denoted by $V(g)$ and the edge set by $E(g)$. A label function, l, maps a vertex or an edge to a label. A graph g is a subgraph of another graph g' if there exists a subgraph isomorphism from g to g', denoted by $g \subseteq g'$. g' is called a supergraph of g.

Definition 12.1 (Subgraph Isomorphism). *For two labeled graphs g and g', a subgraph isomorphism is an injective function $f : V(g) \rightarrow V(g')$, s.t., (1), $\forall v \in V(g), l(v) = l'(f(v))$; and (2), $\forall (u, v) \in E(g), (f(u),$*

$f(v)) \in E(g')$ and $l(u,v) = l'(f(u), f(v))$, where l and l' are the labeling functions of g and g', respectively. f is called an embedding of g in g'.

Definition 12.2 (Frequent Graph). *Given a labeled graph dataset $D = \{G_1, G_2, \ldots, G_n\}$ and a subgraph g, the supporting graph set of g is $D_g = \{G_i | g \subseteq G_i, G_i \in D\}$. The support of g is $support(g) = \frac{|D_g|}{|D|}$. A frequent graph is a graph whose support is no less than a minimum support threshold, min_sup.*

An important property, called *anti-monotonicity*, is crucial to confine the search space of frequent subgraph mining.

Definition 12.3 (Anti-Monotonicity). *Anti-monotonicity means that a size-k subgraph is frequent only if all of its subgraphs are frequent.*

Many frequent graph pattern mining algorithms [12, 6, 16, 20, 28, 32, 2, 14, 15, 22, 21, 8, 3] have been proposed. Holder et al. [12] developed SUBDUE to do approximate graph pattern discovery based on minimum description length and background knowledge. Dehaspe et al. [6] applied inductive logic programming to predict chemical carcinogenicity by mining frequent subgraphs. Besides these studies, there are two basic approaches to the frequent subgraph mining problem: the Apriori-based approach and the pattern-growth approach.

2.2 Apriori-based Approach

Apriori-based frequent subgraph mining algorithms share similar characteristics with Apriori-based frequent itemset mining algorithms. The search for frequent subgraphs starts with small-size subgraphs, and proceeds in a bottom-up manner. At each iteration, the size of newly discovered frequent subgraphs is increased by one. These new subgraphs are generated by joining two similar but slightly different frequent subgraphs that were discovered already. The frequency of the newly formed graphs is then checked. The framework of Apriori-based methods is outlined in Algorithm 14.

Typical Apriori-based frequent subgraph mining algorithms include AGM by Inokuchi et al. [16], FSG by Kuramochi and Karypis [20], and an edge-disjoint path-join algorithm by Vanetik et al. [28].

The AGM algorithm uses a *vertex-based candidate generation* method that increases the subgraph size by one vertex in each iteration. Two size-$(k+1)$ frequent subgraphs are joined only when the two graphs have the same size-k subgraph. Here, *graph size* means the number of vertices in a graph. The newly formed candidate includes the common size-k subgraph and the additional two vertices from the two size-$(k+1)$ patterns. Figure 12.1 depicts the two subgraphs joined by two chains.

Algorithm 14 Apriori(D, *min_sup*, S_k)

Input: Graph dataset D, minimum support threshold *min_sup*,
size-k frequent subgraphs S_k
Output: The set of size-$(k+1)$ frequent subgraphs S_{k+1}

1: $S_{k+1} \leftarrow \varnothing$;
2: **for** each frequent subgraph $g_i \in S_k$ **do**
3: **for** each frequent subgraph $g_j \in S_k$ **do**
4: **for** each size-$(k+1)$ graph g formed by joining g_i and g_j **do**
5: **if** g is frequent in D and $g \notin S_{k+1}$ **then**
6: insert g to S_{k+1};
7: **if** $S_{k+1} \neq \varnothing$ **then**
8: call Apriori(D, *min_sup*, S_{k+1});
9: **return**;

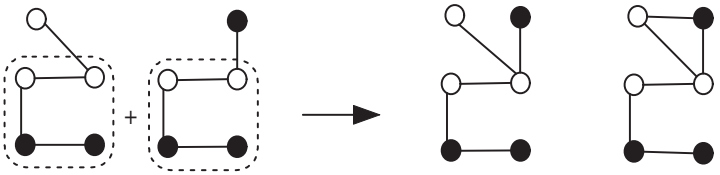

Figure 12.1. AGM: Two candidate patterns formed by two chains

The FSG algorithm adopts an *edge-based candidate generation* strategy that increases the subgraph size by one edge in each iteration. Two size-$(k+1)$ patterns are merged if and only if they share the same subgraph having k edges. In the *edge-disjoint path* method [28], graphs are classified by the number of disjoint paths they have, and two paths are edge-disjoint if they do not share any common edge. A subgraph pattern with k+1 disjoint paths is generated by joining subgraphs with k disjoint paths.

The Apriori-based algorithms mentioned above have considerable overhead when two size-k frequent subgraphs are joined to generate size-$(k+1)$ candidate patterns. In order to avoid this kind of overhead, non-Apriori-based algorithms were developed, most of which adopt the pattern-growth methodology, as discussed below.

2.3 Pattern-Growth Approach

Pattern-growth graph mining algorithms include gSpan by Yan and Han [32], MoFa by Borgelt and Berthold [2], FFSM by Huan et al. [14], SPIN by Huan et al. [15], and Gaston by Nijssen and Kok [22]. These algorithms are

inspired by PrefixSpan [23], TreeMinerV [37], and FREQT [1] in mining sequences and trees, respectively.

The pattern-growth algorithm extends a frequent graph directly by adding a new edge, in every possible position. It does not perform expensive join operations. A potential problem with the edge extension is that the same graph can be discovered multiple times. The gSpan algorithm helps avoiding the discovery of duplicates by introducing a *right-most extension* technique, where the only extensions take place on the *right-most path* [32]. A right-most path for a given graph is the straight path from the starting vertex v_0 to the last vertex v_n, according to a depth-first search on the graph.

Besides the frequent subgraph mining algorithms, constraint-based subgraph mining algorithms have also been proposed. Mining closed graph patterns was studied by Yan and Han [33]. Mining coherent subgraphs was studied by Huan et al. [13]. Chi et al. proposed CMTreeMiner to mine closed and maximal frequent subtrees [5]. For relational graph mining, Yan et al. [36] developed two algorithms, CloseCut and Splat, to discover exact dense frequent subgraphs in a set of relational graphs. For large-scale graph database mining, a disk-based frequent graph mining method was introduced by Wang et al. [29]. Jin et al. [17] proposed an algorithm, TSMiner, for mining frequent large-scale structures (defined as topological structures) from graph datasets.

For a comprehensive introduction on basic graph pattern mining algorithms including Apriori-based and pattern-growth approaches, readers are referred to the survey written by Washio and Motoda [30] and Yan and Han [34].

2.4 Closed and Maximal Subgraphs

A major challenge in mining frequent subgraphs is that the mining process often generates a huge number of patterns. This is because if a subgraph is frequent, all of its subgraphs are frequent as well. A frequent graph pattern with n edges can potentially have 2^n frequent subgraphs, which is an exponential number. To overcome this problem, *closed subgraph mining* and *maximal subgraph mining* algorithms were proposed.

Definition 12.4 (Closed Subgraph). *A subgraph g is a closed subgraph in a graph set D if g is frequent in D and there exists no proper supergraph g' such that $g \subset g'$ and g' has the same support as g in D.*

Definition 12.5 (Maximal Subgraph). *A subgraph g is a maximal subgraph in a graph set D if g is frequent, and there exists no supergraph g' such that $g \subset g'$ and g' is frequent in D.*

The set of closed frequent subgraphs contains the complete information of frequent patterns; whereas the set of maximal subgraphs, though more compact, usually does not contain the complete support information regarding to

its corresponding frequent sub-patterns. Close subgraph mining methods include CloseGraph [33]. Maximal subgraph mining methods include SPIN [15] and MARGIN [26].

2.5 Mining Subgraphs in a Single Graph

While most frequent subgraph mining algorithms assume the input graph data is a set of graphs $D = \{G_1, ..., G_n\}$, there are some studies [21, 8, 3] on mining graph patterns from a single large graph. Defining the support of a subgraph in a set of graphs is straightforward, which is the number of graphs in the database that contain the subgraph. However, it is much more difficult to find an appropriate support definition in a single large graph since multiple embeddings of a subgraph may have overlaps. If arbitrary overlaps between non-identical embeddings are allowed, the resulting support does not satisfy the anti-monotonicity property, which is essential for most frequent pattern mining algorithms. Therefore, [21, 8, 3] investigated appropriate support measures in a single graph.

Kuramochi and Karypis [21] proposed two efficient algorithms that can find frequent subgraphs within a large sparse graph. The first algorithm, called HSIGRAM, follows a horizontal approach and finds frequent subgraphs in a breadth-first fashion. The second algorithm, called VSIGRAM, follows a vertical approach and finds the frequent subgraphs in a depth-first fashion. For the support measure defined in [21], all possible occurrences φ of a pattern p in a graph g are calculated. An *overlap-graph* is constructed where each occurrence φ corresponds to a node and there is an edge between the nodes of φ and φ' if they overlap. This is called *simple overlap* as defined below.

Definition 12.6 (Simple Overlap). *Given a pattern $p = (V(p), E(p))$, a simple overlap of occurrences φ and φ' of pattern p exists if $\varphi(E(p)) \cap \varphi'(E(p)) \neq \varnothing$.*

The support of p is defined as the size of the maximum independent set (MIS) of the overlap-graph. A later study [8] proved that the MIS-support is anti-monotone.

Fiedler and Borgelt [8] suggested a definition that relies on the non-existence of equivalent ancestor embeddings in order to guarantee that the resulting support is anti-monotone. The support is called *harmful overlap support*. The basic idea of this measure is that some of the simple overlaps (in [21]) can be disregarded without harming the anti-monotonicity of the support measure. As in [21], an overlap graph is constructed and the support is defined as the size of the MIS. The major difference is the definition of the overlap.

Definition 12.7 (Harmful Overlap). *Given a pattern $p = (V(p), E(p))$, a harmful overlap of occurrences φ and φ' of pattern p exists if $\exists v \in V(p) : \varphi(v), \varphi'(v) \in \varphi(V(p)) \cap \varphi'(V(p))$.*

Bringmann and Nijssen [3] examined the existing studies [21, 8] and identified the expensive operation of solving the MIS problem. They defined a new support measure.

Definition 12.8 (Minimum Image based Support). *Given a pattern $p = (V(p), E(p))$, the minimum image based support of p in g is defined as*

$$\sigma_{\wedge}(p, g) = \min_{v \in V(p)} |\{\varphi_i(v) : \varphi_i \; is \; an \; occurrence \; of \; p \; in \; g\}|.$$

It is based on the number of unique nodes in the graph g to which a node of the pattern p is mapped. This measure avoids the MIS computation. Therefore it is computationally less expensive and often closer to intuition than measures proposed in [21, 8].

By taking the node in p which is mapped to the least number of unique nodes in g, the anti-monotonicity of $\sigma_{\wedge}$ can be guaranteed. For the definition of support, several computational benefits could be identified: (1) instead of $O(n^2)$ potential overlaps, where n is the possibly exponential number of occurrences, the method only needs to maintain a set of vertices for every node in the pattern, which can be done in $O(n)$; (2) the method does not need to solve an NP complete MIS problem; and (3) it is not necessary to compute all occurrences: it is sufficient to determine for every pair of $v \in V(p)$ and $v' \in V(g)$ if there is one occurrence in which $\varphi(v) = v'$.

2.6 The Computational Bottleneck

Most graph mining methods follow the combinatorial pattern enumeration paradigm. In real world applications including bioinformatics and social network analysis, the complete enumeration of patterns is practically infeasible. It often turns out that the mining results, even those for closed graphs [33] or maximal graphs [15], are explosive in size.

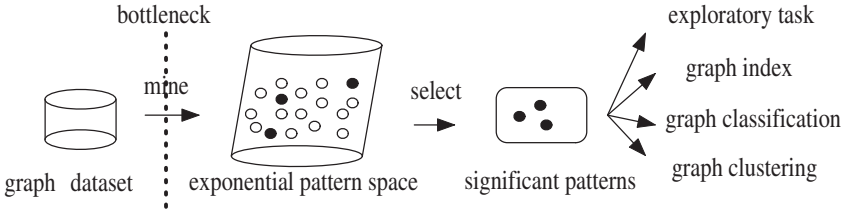

Figure 12.2. Graph Pattern Application Pipeline

Figure 12.2 depicts the pipeline of graph applications built on frequent subgraphs. In this pipeline, frequent subgraphs are mined first; then significant patterns are selected based on user-defined objective functions for different applications. Unfortunately, the potential of graph patterns is hindered by the limitation of this pipeline, due to a scalability issue. For instance, in order to find subgraphs with the highest statistical significance, one has to enumerate all the frequent subgraphs first, and then calculate their p-value one by one. Obviously, this two-step process is not scalable due to the following two reasons: (1) for many objective functions, the minimum frequency threshold has to be set very low so that none of significant patterns will be missed—a low-frequency threshold often means an exponential pattern set and an extremely slow mining process; and (2) there is a lot of redundancy in frequent subgraphs; most of them are not worth computing at all. When the complete mining results are prohibitively large, yet only the significant or representative ones are of real interest. It is inefficient to wait forever for the mining algorithm to finish and then apply post-processing to the huge mining result. In order to complete mining in a limited period of time, a user usually has to sacrifice patterns' quality. In short, the frequent subgraph mining step becomes the bottleneck of the whole pipeline in Figure 12.2.

In the following discussion, we will introduce recent graph pattern mining methods that overcome the scalability bottleneck. The first series of studies [19, 11, 27, 31, 25, 24] focus on mining the optimal or significant subgraphs according to user-specified objective functions in a timely fashion by accessing only a small subset of promising subgraphs. The second study [10] by Hasan et al. generates an orthogonal set of graph patterns that are representative. All these studies avoid generating the complete set of frequent subgraphs while presenting only a compact set of interesting subgraph patterns, thus solving the scalability and applicability issues.

3. Mining Significant Graph Patterns

3.1 Problem Definition

Given a graph database $D = \{G_1, ..., G_n\}$ and an objective function F, a general problem definition for mining significant graph patterns can be formulated in two different ways: (1) find all subgraphs g such that $F(g) \geq \delta$ where δ is a significance threshold; or (2) find a subgraph g^* such that $g^* = \text{argmax}_g F(g)$. No matter which formulation or which objective function is used, an efficient mining algorithm shall find significant patterns directly without exhaustively generating the whole set of graph patterns. There are several algorithms [19, 11, 27, 31, 25, 24] proposed with different objective functions and pruning techniques. We are going to discuss four recent studies: gboost [19], gPLS [25], LEAP [31] and GraphSig [24].

3.2 gboost: A Branch-and-Bound Approach

Kudo et al. [19] presented an application of boosting for classifying labeled graphs, such as chemical compounds, natural language texts, *etc.* A weak classifier called decision stump uses a subgraph as a classification feature. Then a boosting algorithm repeatedly constructs multiple weak classifiers on weighted training instances. A gain function is designed to evaluate the quality of a decision stump, *i.e.*, how many weighted training instances can be correctly classified. Then the problem of finding the optimal decision stump in each iteration is formulated as mining an "optimal" subgraph pattern. gboost designs a branch-and-bound mining approach based on the gain function and integrates it into gSpan to search for the "optimal" subgraph pattern.

A Boosting Framework. gboost uses a simple classifier, *decision stump*, for prediction according to a single feature. The subgraph-based decision stump is defined as follows.

Definition 12.9 (Decision Stumps for Graphs). *Let t and $\boldsymbol{x}$ be labeled graphs and $y \in \{\pm 1\}$ be a class label. A decision stump classifier for graphs is given by*

$$h_{\langle t,y\rangle}(\boldsymbol{x}) = \begin{cases} y, & t \subseteq \boldsymbol{x} \\ -y, & otherwise \end{cases}.$$

The decision stumps are trained to find a rule $\langle \hat{t}, \hat{y}\rangle$ that minimizes the error rate for the given training data $T = \{\langle \mathbf{x}_i, y_i\rangle\}_{i=1}^{L}$,

$$\begin{aligned} \langle \hat{t}, \hat{y}\rangle &= \arg\min_{t\in\mathcal{F}, y\in\{\pm 1\}} \frac{1}{L}\sum_{i=1}^{L} I(y_i \neq h_{\langle t,y\rangle}(\mathbf{x}_i)) \\ &= \arg\min_{t\in\mathcal{F}, y\in\{\pm 1\}} \frac{1}{2L}\sum_{i=1}^{L}(1 - y_i h_{\langle t,y\rangle}(\mathbf{x}_i)), \end{aligned} \tag{3.1}$$

where $\mathcal{F}$ is a set of candidate graphs or a feature set (*i.e.*, $\mathcal{F} = \bigcup_{i=1}^{L}\{t|t \subseteq \mathbf{x}_i\}$) and $I(\cdot)$ is the indicator function. The gain function for a rule $\langle t, y\rangle$ is defined as

$$gain(\langle t, y\rangle) = \sum_{i=1}^{L} y_i h_{\langle t,y\rangle}(\mathbf{x}_i). \tag{3.2}$$

Using the gain, the search problem in Eq.(3.1) becomes equivalent to the problem: $\langle \hat{t}, \hat{y}\rangle = \arg\max_{t\in\mathcal{F}, y\in\{\pm 1\}} gain(\langle t, y\rangle)$. Then the gain function is used instead of error rate.

gboost applies AdaBoost [9] by repeatedly calling the decision stumps and finally produces a hypothesis f, which is a linear combination of K hypotheses

produced by the decision stumps $f(\mathbf{x}) = sgn(\sum_{k=1}^{K} \alpha_k h_{\langle t_k, y_k \rangle}(\mathbf{x}))$. In the kth iteration, a decision stump is built with weights $\mathbf{d}^{(k)} = (d_1^{(k)}, ..., d_L^{(k)})$ on the training data, where $\sum_{i=1}^{L} d_i^{(k)} = 1$, $d_i^{(k)} \geq 0$. The weights are calculated to concentrate more on hard examples than easy ones. In the boosting framework, the gain function is redefined as

$$gain(\langle t, y \rangle) = \sum_{i=1}^{L} y_i d_i h_{\langle t,y \rangle}(\mathbf{x}_i). \tag{3.3}$$

A Branch-and-Bound Search Approach. According to the gain function in Eq.(3.3), the problem of finding the optimal rule $\langle \hat{t}, \hat{y} \rangle$ from the training dataset is defined as follows.

Problem 1 [Find Optimal Rule] Let $T = \{\langle \mathbf{x}_1, y_1, d_1 \rangle, ..., \langle \mathbf{x}_L, y_L, d_L \rangle\}$ be a training data set where $\mathbf{x}_i$ is a labeled graph, $y_i \in \{\pm 1\}$ is a class label associated with $\mathbf{x}_i$ and d_i ($\sum_{i=1}^{L} d_i = 1$, $d_i \geq 0$) is a normalized weight assigned to $\mathbf{x}_i$. Given T, find the optimal rule $\langle \hat{t}, \hat{y} \rangle$ that maximizes the gain, *i.e.*, $\langle \hat{t}, \hat{y} \rangle = \arg\max_{t \in \mathcal{F}, y \in \{\pm 1\}} y_i d_i h_{\langle t,y \rangle}$, where $\mathcal{F} = \bigcup_{i=1}^{L} \{t | t \subseteq \mathbf{x}_i\}$.

A naive method is to enumerate all subgraphs $\mathcal{F}$ and then calculate the gains for all subgraphs. However, this method is impractical since the number of subgraphs is exponential to their size. To avoid such exhaustive enumeration, the method to find the optimal rule is modeled as a branch-and-bound algorithm based on the upper bound of the gain function which is defined as follows.

Lemma 12.10 (Upper bound of the gain). *For any $t' \supseteq t$ and $y \in \{\pm 1\}$, the gain of $\langle t', y \rangle$ is bounded by $\mu(t)$ (i.e., $gain(\langle t', y \rangle) \leq \mu(t)$), where $\mu(t)$ is given by*

$$\mu(t) = max(2 \sum_{\{i | y_i = +1, t \subseteq x_i\}} d_i - \sum_{i=1}^{L} y_i \cdot d_i, 2 \sum_{\{i | y_i = -1, t \subseteq x_i\}} d_i + \sum_{i=1}^{L} y_i \cdot d_i). \tag{3.4}$$

Figure 12.3 depicts a graph pattern search tree where each node represents a graph. A graph g' is a child of another graph g if g' is a supergraph of g with one more edge. g' is also written as $g' = g \diamond e$, where e is the extra edge. In order to find an optimal rule, the branch-and-bound search estimates the upper bound of the gain function for all descendants below a node g. If it is smaller than the value of the best subgraph seen so far, it cuts the search branch of that node. Under the branch-and-bound search, a tighter upper bound is always preferred since it means faster pruning.

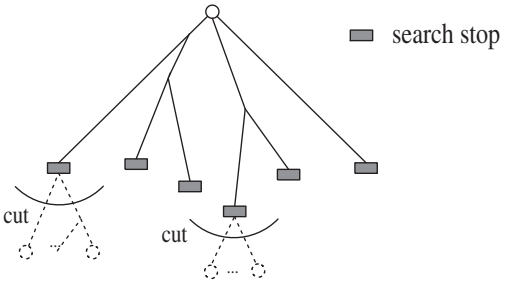

Figure 12.3. Branch-and-Bound Search

Algorithm 15 outlines the framework of branch-and-bound for searching the optimal graph pattern. In the initialization, all the subgraphs with one edge are enumerated first and these seed graphs are then iteratively extended to large subgraphs. Since the same graph could be grown in different ways, Line 5 checks whether it has been discovered before; if it has, then there is no need to grow it again. The optimal $gain(\langle \hat{t}, \hat{y} \rangle)$ discovered so far is maintained. If $\mu(t) \leq gain(\langle \hat{t}, \hat{y} \rangle)$, the branch of t can safely be pruned.

Algorithm 15 Branch-and-Bound

Input: Graph dataset D
Output: Optimal rule $\langle \hat{t}, \hat{y} \rangle$

1: $S = \{1\text{-edge graph}\}$;
2: $\langle \hat{t}, \hat{y} \rangle = \varnothing$; $gain(\langle \hat{t}, \hat{y} \rangle) = -\infty$;
3: **while** $S \neq \varnothing$ **do**
4: choose t from S, $S = S \setminus \{t\}$;
5: **if** t was examined **then**
6: **continue**;
7: **if** $gain(\langle t, y \rangle) > gain(\langle \hat{t}, \hat{y} \rangle)$ **then**
8: $\langle \hat{t}, \hat{y} \rangle = \langle t, y \rangle$;
9: **if** $\mu(t) \leq gain(\langle \hat{t}, \hat{y} \rangle)$ **then**
10: **continue**;
11: $S = S \cup \{t' | t' = t \diamond e\}$;
12: **return** $\langle \hat{t}, \hat{y} \rangle$;

3.3 gPLS: A Partial Least Squares Regression Approach

Saigo et al. [25] proposed gPLS, an iterative mining method based on partial least squares regression (PLS). To apply PLS to graph data, a sparse version

of PLS is developed first and then it is combined with a weighted pattern mining algorithm. The mining algorithm is iteratively called with different weight vectors, creating one latent component per one mining call. Branch-and-bound search is integrated into graph mining with a designed gain function and a pruning condition. In this sense, gPLS is very similar to the branch-and-bound mining approach in gboost.

Partial Least Squares Regression. This part is a brief introduction to partial least squares regression (PLS). Assume there are n training examples $(x_1, y_1), ..., (x_n, y_n)$. The output y_i is assumed to be centralized $\sum_i y_i = 0$. Denote by X the design matrix, where each row corresponds to x_i^T. The regression function of PLS is

$$f(x) = \sum_{i=1}^{m} \alpha_i w_i^T x,$$

where m is the pre-specified number of components that form a subset of the original space, and w_i are weight vectors that reduce the dimensionality of x, satisfying the following orthogonality condition,

$$w_i^T X^T X w_j = \begin{cases} 1 & (i = j) \\ 0 & (i \neq j) \end{cases}.$$

Basically w_i are learned in a greedy way first, then the coefficients α_i are obtained by least squares regression without any regularization. The solutions to α_i and w_i are

$$\alpha_i = \sum_{k=1}^{n} y_k w_i^T x_k, \tag{3.5}$$

and

$$w_i = \arg\max_{w} \frac{(\sum_{k=1}^{n} y_k w^T x_k)^2}{w^T w},$$

subject to $w^T X^T X w = 1$, $w^T X^T X w_j = 0$, $j = 1, ..., i-1$.

Next we present an alternative derivation of PLS called *non-deflation sparse PLS*. Define the i-th latent component as $t_i = X w_i$ and T_{i-1} as the matrix of latent components obtained so far, $T_{i-1} = (t_1, ..., t_{i-1})$. The residual vector is computed by

$$r_i = (I - T_{i-1} T_{i-1}^T) y.$$

Then multiply it with X^T to obtain

$$v = \frac{1}{\eta} X^T (I - T_{i-1} T_{i-1}^T) y.$$

The non-deflation sparse PLS follows this idea.

In graph mining, it is useful to have sparse weight vectors w_i such that only a limited number of patterns are used for prediction. To this aim, we introduce the sparseness to the pre-weight vectors v_i as

$$v_{ij} = 0, \;\; if \; |v_{ij}| \leq \epsilon, \;\; j = 1, .., d.$$

Due to the linear relationship between v_i and w_i, w_i becomes sparse as well. Then we can sort $|v_{ij}|$ in the descending order, take the top-k elements and set all the other elements to zero.

It is worthwhile to notice that the residual of regression up to the $(i-1)$-th features,

$$r_{ik} = y_k - \sum_{j=1}^{i-1} \alpha_j w_j^T x_k, \tag{3.6}$$

is equal to the k-th element of r_i. It can be verified by substituting the definition of α_j in Eq.(3.5) into Eq.(3.6). So in the non-deflation algorithm, the pre-weight vector v is obtained as the direction that maximizes the covariance with residues. This observation highlights the resemblance of PLS and boosting algorithms.

Graph PLS: Branch-and-Bound Search. In this part, we discuss how to apply the non-deflation PLS algorithm to graph data. The set of training graphs is represented as (G_1, y_1),...,(G_n, y_n). Let $\mathcal{P}$ be the set of all patterns, then the feature vector of each graph G_i is encoded as a $|\mathcal{P}|$-dimensional vector x_i. Since $|\mathcal{P}|$ is a huge number, it is infeasible to keep the whole design matrix. So the method sets X as an empty matrix first, and grows the matrix as the iteration proceeds. In each iteration, it obtains the set of patterns p whose pre-weight $|v_{ip}|$ is above the threshold, which can be written as

$$P_i = \{p || \sum_{j=1}^{n} r_{ij} x_{jp}| \geq \epsilon\}. \tag{3.7}$$

Then the design matrix is expanded to include newly introduced patterns. The pseudo code of gPLS is described in Algorithm 16.

The pattern search problem in Eq.(3.7) is exactly the same as the one solved in gboost through a branch-and-bound search. In this problem, the gain function is defined as $s(p) = |\sum_{j=1}^{n} r_{ij} x_{jp}|$. The pruning condition is described as follows.

Theorem 12.11. *Define $\tilde{y}_i = sgn(r_i)$. For any pattern p' such that $p \subseteq p'$, $s(p') < \epsilon$ holds if*

$$\max\{s^+(p), s^-(p)\} < \epsilon, \tag{3.8}$$

where

$$s^+(p) = 2 \sum_{\{i|\tilde{y}_i=+1, x_{i,j}=1\}} |r_i| - \sum_{i=1}^{n} r_i,$$

$$s^-(p) = 2 \sum_{\{i|\tilde{y}_i=-1, x_{i,j}=1\}} |r_i| + \sum_{i=1}^{n} r_i.$$

Algorithm 16 gPLS

Input: Training examples $(G_1, y_1), (G_2, y_2), ..., (G_n, y_n)$
Output: Weight vectors w_i, $i = 1, ..., m$

1: $r_1 = y$, $X = \varnothing$;
2: **for** $i = 1, ..., m$ **do**
3: $P_i = \{p || \sum_{j=1}^{n} r_{ij} x_{jp}| \geq \epsilon\}$;
4: X_{P_i}: design matrix restricted to P_i;
5: $X \leftarrow X \cup X_{P_i}$;
6: $v_i = X^T r_i / \eta$;
7: $w_i = v_i - \sum_{j=1}^{i-1} (w_j^T X^T X v_i) w_j$;
8: $t_i = X w_i$;
9: $r_{i+1} = r_i - (y^T t_i) t_i$;

3.4 LEAP: A Structural Leap Search Approach

Yan et al. [31] proposed an efficient algorithm which mines the most significant subgraph pattern with respect to an objective function. A major contribution of this study is the proposal of a general approach for significant graph pattern mining with non-monotonic objective functions. The mining strategy, called LEAP (Descending Leap Mine), explored two new mining concepts: (1) *structural leap search*, and (2) *frequency-descending mining*, both of which are related to specific properties in pattern search space. The same mining strategy can also be applied to searching other simpler structures such as itemsets, sequences and trees.

Structural Leap Search. Figure 12.4 shows a search space of subgraph patterns. If we examine the search structure horizontally, we find that the subgraphs along the neighbor branches likely have similar compositions and frequencies, hence similar objective scores. Take the branches A and B as an example. Suppose A and B split on a common subgraph pattern g. Branch A

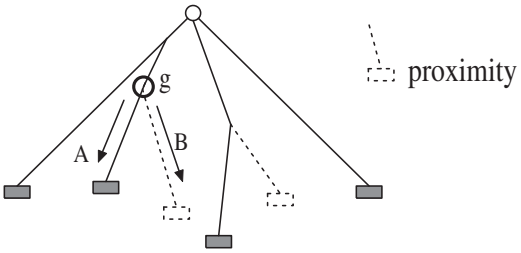

Figure 12.4. Structural Proximity

contains all the supergraphs of $g \diamond e$ and B contains all the supergraphs of g except those of $g \diamond e$. For a graph g' in branch B, let $g'' = g' \diamond e$ in branch A.

LEAP assumes each input graph is assigned either a positive or a negative label (*e.g.*, compounds active or inactive to a virus). One can divide the graph dataset into two subsets: a positive set D_+ and a negative set D_-. Let $p(g)$ and $q(g)$ be the frequency of a graph pattern g in positive graphs and negative graphs. Many objective functions can be represented as a function of p and q for a subgraph pattern g, as $F(g) = f(p(g), q(g))$.

If in a graph dataset, $g \diamond e$ and g often occur together, then g'' and g' might also occur together. Hence, likely $p(g'') \operatorname{sim} p(g')$ and $q(g'') \operatorname{sim} q(g')$, which means similar objective scores. This is resulted by the structural and embedding similarity between the starting structures $g \diamond e$ and g. We call it **structural proximity**: Neighbor branches in the pattern search tree exhibit strong similarity not only in pattern composition, but also in their embeddings in the graph datasets, thus having similar frequencies and objective scores. In summary, a conceptual claim can be drawn,

$$g' \operatorname{sim} g'' \Rightarrow F(g') \operatorname{sim} F(g''). \tag{3.9}$$

According to structural proximity, it seems reasonable to skip the whole search branch once its nearby branch is searched, since the best scores between neighbor branches are likely similar. Here, we would like to emphasize "likely" rather than "surely". Based on this intuition, if the branch A in Figure 12.4 has been searched, B could be "leaped over" if A and B branches satisfy some similarity criterion. The length of leap can be controlled by the frequency difference of two graphs g and $g \diamond e$. The leap condition is defined as follows.

Let $I(G, g, g \diamond e)$ be an indicator function of a graph G: $I(G, g, g \diamond e) = 1$, for any supergraph g' of g, if $g' \subseteq G$, $\exists g'' = g' \diamond e$ such that $g'' \subseteq G$; otherwise 0. When $I(G, g, g \diamond e) = 1$, it means if a supergraph g' of g has an embedding in G, there must be an embedding of $g' \diamond e$ in G. For a positive dataset D_+, let $D_+(g, g \diamond e) = \{G | I(G, g, g \diamond e) = 1, g \subseteq G, G \in D_+\}$. In $D_+(g, g \diamond e)$,

$g' \supset g$ and $g'' = g' \diamond e$ have the same frequency. Define $\Delta_+(g, g \diamond e)$ as follows,

$$\Delta_+(g, g \diamond e) = p(g) - \frac{|D_+(g, g \diamond e)|}{|D_+|}.$$

$\Delta_+(g, g \diamond e)$ is actually the maximum frequency difference that g' and g'' could have in D_+. If the difference is smaller than a threshold σ, then leap,

$$\frac{2\Delta_+(g, g \diamond e)}{p(g \diamond e) + p(g)} \leq \sigma \text{ and } \frac{2\Delta_-(g, g \diamond e)}{q(g \diamond e) + q(g)} \leq \sigma. \tag{3.10}$$

σ controls the leap length. The larger σ is, the faster the search is. Structural leap search will generate an optimal pattern candidate and reduce the need for thoroughly searching similar branches in the pattern search tree. Its goal is to help program search significantly distinct branches, and limit the chance of missing the most significant pattern.

Algorithm 17 Structural Leap Search: sLeap(D, σ, $g^\star$)

Input: Graph dataset D, difference threshold σ
Output: Optimal graph pattern candidate $g^\star$

1: $S = \{1 - \text{edge graph}\}$;
2: $g^\star = \varnothing$; $F(g^\star) = -\infty$;
3: **while** $S \neq \varnothing$ **do**
4: $S = S \setminus \{g\}$;
5: **if** g was examined **then**
6: **continue**;
7: **if** $\exists g \diamond e, g \diamond e \prec g$, $\frac{2\Delta_+(g,g\diamond e)}{p(g\diamond e)+p(g)} \leq \sigma$, $\frac{2\Delta_-(g,g\diamond e)}{q(g\diamond e)+q(g)} \leq \sigma$ **then**
8: **continue**;
9: **if** $F(g) > F(g^\star)$ **then**
10: $g^\star = g$;
11: **if** $\widehat{F}(g) \leq F(g^\star)$ **then**
12: **continue**;
13: $S = S \cup \{g' | g' = g \diamond e\}$;
14: **return** $g^\star$;

Algorithm 17 outlines the pseudo code of structural leap search (sLeap). The leap condition is tested on Lines 7-8. Note that sLeap does not guarantee the optimality of result.

Frequency Descending Mining. Structural leap search takes advantages of the correlation between structural similarity and significance similarity. However, it does not exploit the possible relationship between patterns' frequency

and patterns' objective scores. Existing solutions have to set the frequency threshold very low so that the optimal pattern will not be missed. Unfortunately, low-frequency threshold could generate a huge set of low-significance redundant patterns with long mining time.

Although most of objective functions are not correlated with frequency monotonically or anti-monotonically, they are not independent of each other. Cheng et al. [4] derived a frequency upper bound of discriminative measures such as information gain and Fisher score, showing a relationship between frequency and discriminative measures. According to this analytical result, if all frequent subgraphs are ranked in increasing order of their frequency, significant subgraph patterns are often in the high-end range, though their real frequency could vary dramatically across different datasets.

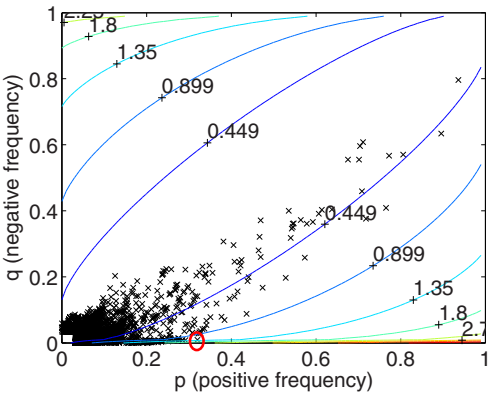

Figure 12.5. Frequency vs. G-test score

Figure 12.5 illustrates the relationship between frequency and G-test score for an AIDS Anti-viral dataset [31]. It is a contour plot displaying isolines of G-test score in two dimensions. The X axis is the frequency of a subgraph g in the positive dataset, *i.e.*, $p(g)$, while the Y axis is the frequency of the same subgraph in the negative dataset, $q(g)$. The curves depict G-test score. Left upper corner and right lower corner have the higher G-test scores. The "circle" marks the highest G-score subgraph discovered in this dataset. As one can see, its positive frequency is higher than most of subgraphs.

[Frequency Association]*Significant patterns often fall into the high-quantile of frequency.*

To profit from frequency association, an iterative frequency-descending mining method is proposed in [31]. Rather than performing mining with very low frequency, the method starts the mining process with high frequency threshold $\theta = 1.0$, calculates an optimal pattern candidate $g^\star$ whose frequency is at least θ, and then repeatedly lowers down θ to check whether $g^\star$ can be

improved further. Here, the search leaps in the frequency domain, by leveling down the minimum frequency threshold exponentially.

Algorithm 18 Frequency-Descending Mine: fLeap(D, ε, $g^\star$)

Input: Graph dataset D, converging threshold ε
Output: Optimal graph pattern candidate $g^\star$

1: $\theta = 1.0$;
2: $g = \varnothing$; $F(g) = -\infty$;
3: **do**
4: $\quad g^\star = g$;
5: $\quad g$=fpmine(D, θ);
6: $\quad \theta = \theta/2$;
7: **while** $(F(g) - F(g^\star) \geq \varepsilon)$
8: **return** $g^\star = g$;

Algorithm 18 (fLeap) outlines the frequency-descending strategy. It starts with the highest frequency threshold, and then lowers the threshold down till the objective score of the best graph pattern converges. Line 5 executes a frequent subgraph mining routine, fpmine, which could be FSG [20], gSpan [32] *etc.* fpmine selects the most significant graph pattern g from the frequent subgraphs it mined. Line 6 implements a simple frequency descending method.

Descending Leap Mine. With structural leap search and frequency-descending mining, a general mining pipeline is built for mining significant graph patterns in a complex graph dataset. It consists of three steps as follows.

Step 1. perform structural leap search with threshold $\theta = 1.0$, generate an optimal pattern candidate $g^\star$.

Step 2. repeat frequency-descending mining with structural leap search until the objective score of $g^\star$ converges.

Step 3. take the best score discovered so far; perform structural leap search again (leap length σ) without frequency threshold; output the discovered pattern.

3.5 GraphSig: A Feature Representation Approach

Ranu and Singh [24] proposed GraphSig, a scalable method to mine significant (measured by p-value) subgraphs based on a feature vector representation of graphs. The first step is to convert each graph into a set of feature vectors where each vector represents a region within the graph. Prior probabilities of

features are computed empirically to evaluate statistical significance of patterns in the feature space. Following the analysis in the feature space, only a small portion of the exponential search space is accessed for further analysis. This enables the use of existing frequent subgraph mining techniques to mine significant patterns in a scalable manner even when they are infrequent. The major steps of GraphSig are described as follows.

Sliding Window across Graphs. As the first step, random walk with restart (abbr. RWR) is performed on each node in a graph to simulate sliding a window across the graph. RWR simulates the trajectory of a random walker that starts from the target node and jumps from one node to a neighbor. Each neighbor has an equal probability of becoming the new station of the walker. At each jump, the feature traversed is updated which can either be an edge label or a node label. A restart probability α brings the walker back to the starting node within approximately $\frac{1}{\alpha}$ jumps. The random walk iterates till the feature distribution converges. As a result, RWR produces a continuous distribution of features for each node where a feature value lies in the range $[0, 1]$, which is further discretized into 10 bins. RWR can therefore be visualized as placing a window at each node of a graph and capturing a feature vector representation of the subgraph within it. A graph of m nodes is represented by m feature vectors. RWR inherently takes proximity of features into account and preserves more structural information than simply counting occurrence of features inside the window.

Calculating P-value of A Feature Vector. To calculate p-value of a feature vector, we model the occurrence of a feature vector $\underline{x}$ in a feature vector space formulated by a random graph. The frequency distribution of a vector is generated using the prior probabilities of features obtained empirically. Given a feature vector $\underline{x} = [x_1, ..., x_n]$, the probability of $\underline{x}$ occurring in a random feature vector $\underline{y} = [y_1, ..., y_n]$ can be expressed as a joint probability

$$P(\underline{x}) = P(y_1 \geq x_1, ..., y_n \geq x_n). \tag{3.11}$$

To simplify the calculation, we assume independence of the features. As a result, Eq.(3.11) can be expressed as a product of the individual probabilities, where

$$P(\underline{x}) = \prod_{i=1}^{n} P(y_i \geq x_i). \tag{3.12}$$

Once $P(\underline{x})$ is known, the support of $\underline{x}$ in a database of random feature vectors can be modeled as a binomial distribution. To illustrate, a random vector can be viewed as a trial and $\underline{x}$ occurring in it as “success". A database consisting m feature vectors will involve m trials for $\underline{x}$. The support of $\underline{x}$ in the database

is the number of successes. Therefore, the probability of $\underline{x}$ having a support μ is

$$P(\underline{x}; \mu) = C_m^{\mu} P(\underline{x})^{\mu} (1 - P(\underline{x}))^{m-\mu}. \quad (3.13)$$

The probability distribution function (abbr. pdf) of $\underline{x}$ can be generated from Eq.(3.13) by varying μ in the range $[0, m]$. Therefore, given an observed support μ_0 of $\underline{x}$, its p-value can be calculated by measuring the area under the pdf in the range $[\mu_0, m]$, which is

$$p\text{-}value(x, \mu_0) = \sum_{i=\mu_0}^{m} P(\underline{x}; i). \quad (3.14)$$

Identifying Regions of Interest. With the conversion of graphs into feature vectors, and a model to evaluate significance of a graph region in the feature space, the next step is to explore how the feature vectors can be analyzed to extract the significant regions. Based on the feature vector representation, the presence of a "common" sub-feature vector among a set of graphs points to a common subgraph. Similarly, the absence of a "common" sub-feature vector indicates the non-existence of any common subgraph. Mathematically, the *floor* of the feature vectors produces the "common" sub-feature vector.

Definition 12.12 (Floor of vectors). *The floor of a set of vectors* $\{\underline{v_1}, ..., \underline{v_m}\}$ *is a vector* $\underline{v_f}$ *where* $v_{f_i} = min(v_{1_i}, ..., v_{m_i})$ *for* $i = 1, ..., n$, n *is the number of dimensions of a vector. Ceiling of a set of vectors is defined analogously.*

The next step is to mine common sub-feature vectors that are also significant. Algorithm 19 presents the FVMine algorithm which explores closed sub-vectors in a bottom-up, depth-first manner. FVMine explores all possible common vectors satisfying the significance and support constraints.

With a model to measure the significance of a vector, and an algorithm to mine closed significant sub-feature vectors, we integrate them to build the significant graph mining framework. The idea is to mine significant sub-feature vectors and use them to locate similar regions which are significant. Algorithm 20 outlines the GraphSig algorithm.

The algorithm first converts each graph into a set of feature vectors and puts all vectors together in a single set D' (lines 3-4). D' is divided into sets, such that D'_a contains all vectors produced from RWR on a node labeled a. On each set D'_a, FVMine is performed with a user-specified support and p-value thresholds to retrieve the set of significant sub-feature vectors (line 7). Given that each sub-feature vector could describe a particular subgraph, the algorithm scans the database to identify the regions where the current sub-feature vector occurs. This involves finding all nodes labeled a and described by a feature vector such that the vector is a super-vector of the current sub-feature vector $\underline{v}$ (line 9). Then the algorithm isolates the subgraph centered

Algorithm 19 FVMine($\underline{x}$, S, b)

Input: Current sub-feature vector $\underline{x}$, supporting set S of $\underline{x}$,
current starting position b
Output: The set of all significant sub-feature vectors A

1: **if** $p\text{-}value(\underline{x}) \leq maxPvalue$ **then**
2: $A \leftarrow A + x$;
3: **for** $i = b$ to m **do**
4: $S' \leftarrow \{\underline{y} | y \in S, y_i > x_i\}$;
5: **if** $|S'| < min\_sup$ **then**
6: **continue**;
7: $\underline{x}' = floor(S')$;
8: **if** $\exists j < i$ such that $x'_j > x_j$ **then**
9: **continue**;
10: **if** $p\text{-}value(ceiling(S'), |S'|) \geq maxPvalue$ **then**
11: **continue**;
12: $FVMine(\underline{x}', S', i)$;

at each node by using a user-specified radius (line 12). This produces a set of subgraphs for each significant sub-feature vector. Next, maximal subgraph mining is performed with a high frequency threshold since it is expected that all of graphs in the set contain a common subgraph (line 13). The last step also prunes out false positives where dissimilar subgraphs are grouped into a set due to the vector representation. For the absence of a common subgraph, when frequent subgraph mining is performed on the set, no frequent subgraph will be produced and as a result the set is filtered out.

4. Mining Representative Orthogonal Graphs

In this section we will discuss ORIGAMI, an algorithm proposed by Hasan et al. [10], which mines a set of α-orthogonal, β-representative graph patterns. Intuitively, two graph patterns are α-orthogonal if their similarity is bounded by a threshold α. A graph pattern is a β-representative of another pattern if their similarity is at least β. The orthogonality constraint ensures that the resulting pattern set has controlled redundancy. For a given α, more than one set of graph patterns qualify as an α-orthogonal set. Besides redundancy control, representativeness is another desired property, *i.e.*, for every frequent graph pattern not reported in the α-orthogonal set, we want to find a representative of this pattern with a high similarity in the α-orthogonal set.

The set of representative orthogonal graph patterns is a compact summary of the complete set of frequent subgraphs. Given user specified thresholds $\alpha, \beta \in$

Algorithm 20 GraphSig(D, min_sup, $maxPvalue$)

Input: Graph dataset D, support threshold min_sup,
p-value threshold $maxPvalue$
Output: The set of all significant sub-feature vectors A

1: $D' \leftarrow \varnothing$;
2: $A \leftarrow \varnothing$;
3: **for** each $g \in D$ **do**
4: $D' \leftarrow D' + RWR(g)$;
5: **for** each node label a in D **do**
6: $D'_a \leftarrow \{\underline{v} | \underline{v} \in D', label(\underline{v}) = a\}$;
7: $S \leftarrow FVMine(floor(D'_a), D'_a, 1)$;
8: **for** each vector $\underline{v} \in S$ **do**
9: $V \leftarrow \{u | u\ is\ a\ node\ of\ label\ a, \underline{v} \subseteq vector(u)\}$;
10: $E \leftarrow \varnothing$;
11: **for** each node $u \in V$ **do**
12: $E \leftarrow E + CutGraph(u, radius)$;
13: $A \leftarrow A + Maximal\_FSM(E, freq)$;

$[0, 1]$, the goal is to mine an α-orthogonal, β-representative graph pattern set that minimizes the set of unrepresented patterns.

4.1 Problem Definition

Given a collection of graphs D and a similarity threshold $\alpha \in [0, 1]$, a subset of graphs $\mathcal{R} \subseteq D$ is α-orthogonal with respect to D iff for any $G_a, G_b \in \mathcal{R}$, $sim(G_a, G_b) \leq \alpha$ and for any $G_i \in D \backslash \mathcal{R}$ there exists a $G_j \in \mathcal{R}$, $sim(G_i, G_j) > \alpha$.

Given a collection of graphs D, an α-orthogonal set $\mathcal{R} \subseteq D$ and a similarity threshold $\beta \in [0, 1]$, $\mathcal{R}$ represents a graph $G \in D$, provided that there exists some $G_a \in \mathcal{R}$, such that $sim(G_a, G) \geq \beta$. Let $\Upsilon(\mathcal{R}, D) = \{G | G \in D\ s.t.\ \exists G_a \in \mathcal{R}, sim(G_a, G) \geq \beta\}$, then $\mathcal{R}$ is a β-representative set for $\Upsilon(\mathcal{R}, D)$.

Given D and $\mathcal{R}$, the residue set of $\mathcal{R}$ is the set of unrepresented patterns in D, denoted as $\triangle(\mathcal{R}, D) = D \backslash \{\mathcal{R} \cup \Upsilon(\mathcal{R}, D)\}$.

The problem defined in [10] is to find the α-orthogonal, β-representative set for the set of all maximal frequent subgraphs $\mathcal{M}$ which minimizes the residue set size. The mining problem can be decomposed into two subproblems of *maximal subgraph mining* and *orthogonal representative set generation*, which are discussed separately. Algorithm 21 shows the algorithm framework of ORIGAMI.

Algorithm 21 ORIGAMI(D, min_sup, α, β)

Input: Graph dataset D, minimum support *min_sup*, α, β
Output: α-orthogonal, β-representative set $\mathcal{R}$

1: EM=Edge-Map(D);
2: $\mathcal{F}_1$=Find-Frequent-Edges(D, min_sup);
3: $\widehat{\mathcal{M}} = \phi$;
4: **while** stopping_condition() $\neq$ **true do**
5: M=Random-Maximal-Graph(D, $\mathcal{F}_1$, EM, min_sup);
6: $\widehat{\mathcal{M}} = \widehat{\mathcal{M}} \cup M$;
7: $\mathcal{R}$=Orthogonal-Representative-Sets($\widehat{\mathcal{M}}$, α, β);
8: **return** $\mathcal{R}$;

4.2 Randomized Maximal Subgraph Mining

As the first step, ORIGAMI mines a set of maximal subgraphs, on which the α-orthogonal, β-representative graph pattern set is generated. This is based on the observation that the number of maximal frequent subgraphs is much fewer than that of frequent subgraphs, and the maximal subgraphs provide a synopsis of the frequent ones to some extent. Thus it is reasonable to mine the representative orthogonal pattern set based on the maximal subgraphs rather than the frequent ones. However, even mining all of maximal subgraphs could be infeasible in some real world applications. To avoid this problem, ORIGAMI first finds a sample $\widehat{\mathcal{M}}$ of the complete set of maximal frequent subgraphs $\mathcal{M}$.

The goal is to find a set of maximal subgraphs, $\widehat{\mathcal{M}}$, which is as diverse as possible. To achieve this goal, ORIGAMI avoids using combinatorial enumeration to mine maximal subgraph patterns. Instead, it adopts a random walk approach to enumerate a diverse set of maximal subgraphs from the positive border of such maximal patterns. The randomized mining algorithm starts with an empty pattern and iteratively adds a random edge during each extension, until a maximal subgraph M is generated and no more edges can be added. This process walks a random chain in the partial order of frequent subgraphs. To extend an intermediate pattern, $S_k \subseteq M$, it chooses a random vertex v from which the extension will be attempted. Then a random edge e incident on v is selected for extension. If no such edge is found, no extension is possible from the vertex. When no vertices can have any further extension in S_k, the random walk terminates and $S_k = M$ is the maximal graph. On the other hand, if a random edge e is found, the other endpoint v' of this edge is randomly selected. By adding the edge e and its endpoint v', a candidate subgraph pattern S_{k+1} is generated and its support is computed. This random walk process repeats until

no further extension is possible on any vertex. Then the maximal subgraph M is returned.

Ideally, the random chain walks would cover different regions of the pattern space, thus would produce dissimilar maximal patterns. However, in practice, this may not be the case, since duplicate maximal subgraphs can be generated in the following ways: (1) multiple iterations following overlapping chains, or (2) multiple iterations following different chains but leading to the same maximal pattern. Let's consider a maximal subgraph M of size n. Let $e_1e_2...e_n$ be a sequence of random edge extensions, corresponding to a random chain walk leading from an empty graph ϕ to the maximal graph M. The probability of a particular edge sequence leading from ϕ to M is given as

$$P[(e_1e_2...e_n)] = P(e_1)\prod_{i=2}^{n} P(e_i|e_1...e_{i-1}). \tag{4.1}$$

Let $ES(M)$ denote the set of all valid edge sequences for a graph M. The probability that a graph M is generated in a random walk is proportional to

$$\sum_{e_1e_2...e_n \in ES(M)} P[(e_1e_2...e_n)]. \tag{4.2}$$

The probability of obtaining a specific maximal pattern depends on the number of chains or edge sequences leading to that pattern and the size of the pattern. According to Eq.(4.1), as a graph grows larger, the probability of the edge sequence becomes smaller. So this random walk approach in general favors a maximal subgraph of smaller size than one of larger size. To avoid generating duplicate maximal subgraphs, a termination condition is designed based on an estimate of the collision rate of the generated patterns. Intuitively the collision rate keeps track of the number of duplicate patterns seen within the same or across different random walks. As a random walk chain is traversed, ORIGAMI maintains the signature of the intermediate patterns in a bounded size hash table. As an intermediate or maximal subgraph is generated, its signature is added to the hash table and the collision rate is updated. If the collision rate exceeds a threshold ϵ, the method could (1) abort further extension along the current path and randomly choose another path; or (2) trigger the termination condition across different walks, since it implies that the same part of the search space is being revisited.

4.3 Orthogonal Representative Set Generation

Given a set of maximal subgraphs $\widehat{\mathcal{M}}$, the next step is to extract an α-orthogonal β-representative set from it. We can construct a meta-graph $\Gamma(\widehat{\mathcal{M}})$ to measure similarity between graph patterns in $\widehat{\mathcal{M}}$, in which each node rep-

resents a maximal subgraph pattern, and an edge exists between two nodes if their similarity is bounded by α. Then the problem of finding an α-orthogonal pattern set can be modeled as finding a maximal clique in the similarity graph $\Gamma(\widehat{\mathcal{M}})$.

For a given α, there could be multiple α-orthogonal pattern sets as feasible solutions. We could use the size of the residue set to measure the goodness of an α-orthogonal set. An optimal α-orthogonal β-representative set is the one which minimizes the size of the residue set. [10] proved that this problem is NP-hard.

Given the hardness result, ORIGAMI resorts to approximate algorithms to solve the problem which guarantees local optimality. The algorithm starts with a random maximal clique in the similarity graph $\Gamma(\widehat{\mathcal{M}})$ and tries to improve it. At each state transition, another maximal clique which is a local neighbor of the current maximal clique is chosen. If the new state has a better solution, the new state is accepted as the current state and the process continues. The process terminates when all neighbors of the current state have equal or larger residue sizes. Two maximal cliques of size m and n (assume $m \geq n$) are considered neighbors if they share exactly $n - 1$ vertices. The state transition procedure selectively removes one vertex from the maximal clique of the current state and then expands it to obtain another maximal clique which satisfies the neighborhood constraints.

5. Conclusions

Frequent subgraph mining is one of the fundamental tasks in graph data mining. The inherent complexity in graph data causes the combinatorial explosion problem. As a result, a mining algorithm may take a long time or even forever to complete the mining process on some real graph datasets.

In this chapter, we introduced several state-of-the-art methods that mine a compact set of significant or representative subgraphs without generating the complete set of graph patterns. The proposed mining and pruning techniques were discussed in details. These methods greatly reduce the computational cost, while at the same time, increase the applicability of the generated graph patterns. These research results have made significant progress on graph mining research with a set of new applications.

References

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. 2002 SIAM Int. Conf. Data Mining (SDM'02)*, pages 158–174, 2002.

[2] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. 2002 Int. Conf. Data Mining (ICDM'02)*, pages 211–218, 2002.

[3] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *Proc. 2008 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD'08)*, pages 858–863, 2008.

[4] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, pages 716–725, 2007.

[5] Y. Chi, Y. Xia, Y. Yang, and R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowledge and Data Eng.*, 17:190–202, 2005.

[6] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 30–36, 1998.

[7] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Trans. on Knowledge and Data Engineering*, 17:1036–1050, 2005.

[8] M. Fiedler and C. Borgelt. Support computation for mining frequent subgraphs in a single graph. In *Proc. 5th Int. Workshop on Mining and Learning with Graphs (MLG'07)*, 2007.

[9] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proc. 2nd European Conf. Computational Learning Theory*, pages 23–27, 1995.

[10] M. Al Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: Mining representative orthogonal graph patterns. In *Proc. 2007 Int. Conf. Data Mining (ICDM'07)*, pages 153–162, 2007.

[11] H. He and A. K. Singh. Efficient algorithms for mining significant substructures in graphs with quality guarantees. In *Proc. 2007 Int. Conf. Data Mining (ICDM'07)*, pages 163–172, 2007.

[12] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 169–180, 1994.

[13] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining spatial motifs from protein structure graphs. In *Proc. 8th Int. Conf. Research in Computational Molecular Biology (RECOMB)*, pages 308–315, 2004.

[14] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proc. 2003 Int. Conf. Data Mining (ICDM'03)*, pages 549–552, 2003.

[15] J. Huan, W. Wang, J. Prins, and J. Yang. SPIN: Mining maximal frequent subgraphs from graph databases. In *Proc. 2004 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'04)*, pages 581–586, 2004.

[16] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 2000 European Symp. Principle of Data Mining and Knowledge Discovery (PKDD'00)*, pages 13–23, 1998.

[17] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering frequent topological structures from graph datasets. In *Proc. 2005 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'05)*, pages 606–611, 2005.

[18] M. Koyuturk, A. Grama, and W. Szpankowski. An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, 20:I200–I207, 2004.

[19] T. Kudo, E. Maeda, and Y. Matsumoto. An application of boosting to graph classification. In *Advances in Neural Information Processing Systems 18 (NIPS'04)*, 2004.

[20] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, 2001.

[21] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11:243–271, 2005.

[22] S. Nijssen and J. Kok. A quickstart in frequent structure mining can make a difference. In *Proc. 2004 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'04)*, pages 647–652, 2004.

[23] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, 2001.

[24] S. Ranu and A. K. Singh. GraphSig: A scalable approach to mining significant subgraphs in large graph databases. In *Proc. 2009 Int. Conf. Data Engineering (ICDE'09)*, pages 844–855, 2009.

[25] H. Saigo, N. Kramer, and K. Tsuda. Partial least squares regression for graph mining. In *Proc. 2008 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'08)*, pages 578–586, 2008.

[26] L. Thomas, S. Valluri, and K. Karlapalem. MARGIN: Maximal frequent subgraph mining. In *Proc. 2006 Int. Conf. on Data Mining (ICDM'06)*, pages 1097–1101, 2006.

[27] K. Tsuda. Entire regularization paths for graph data. In *Proc. 2007 Int. Conf. Machine Learning (ICML'07)*, pages 919–926, 2007.

[28] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 458–465, 2002.

[29] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-base graph databases. In *Proc. 2004 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'04)*, pages 316–325, 2004.

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

[31] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by scalable leap search. In *Proc. 2008 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, pages 433–444, 2008.

[32] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. Data Mining (ICDM'02)*, pages 721–724, 2002.

[33] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 286–295, 2003.

[34] X. Yan and J. Han. Discovery of frequent substructures. In *D. Cook and L. Holder (eds.), Mining Graph Data*, pages 99–115, John Wiley Sons, 2007.

[35] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. 2004 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'04)*, pages 335–346, 2004.

[36] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proc. 2005 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'05)*, pages 324–333, 2005.

[37] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 71–80, 2002.