TR 04-024

GREWA Scalable Frequent Subgraph Discovery Algorithm

Michihiro Kuramochi and George Karypis

June 22, 2004

GREW—A Scalable Frequent Subgraph Discovery Algorithm[*]

Michihiro Kuramochi and George Karypis
Department of Computer Science and Engineering/
Army High Performance Computing Research Center/
Digital Technology Center
University of Minnesota
{kuram, karypis}@cs.umn.edu

Abstract

Existing algorithms that mine graph datasets to discover patterns corresponding to frequently occurring subgraphs can operate efficiently on graphs that are sparse, contain a large number of relatively small connected components, have vertices with low and bounded degrees, and contain well-labeled vertices and edges. However, there are a number of applications that lead to graphs that do not share these characteristics, for which these algorithms highly become unscalable.

In this paper we propose a heuristic algorithm called GREW *to overcome the limitations of existing complete or heuristic frequent subgraph discovery algorithms.* GREW *is designed to operate on a large graph and to find patterns corresponding to connected subgraphs that have a large number of vertex-disjoint embeddings. Our experimental evaluation shows that* GREW *is efficient, can scale to very large graphs, and find non-trivial patterns that cover large portions of the input graph and the lattice of frequent patterns.*

Keywords frequent pattern discovery, frequent subgraph, graph mining.

1 Introduction

In the last five years, an ever increasing body of research has focused on developing efficient algorithms to mine frequent patterns in large graph datasets. Starting with AGM by Inokuchi et al. [15], originally developed in 1999–2000, the scalability of these frequent pattern/subgraph mining algorithms has continuously been improved and a number of different algorithms have been developed [20, 17, 1, 32, 11, 13, 33, 19] that employ different mining strategies, are designed for different input graph representations, and find patterns that have different characteristics and satisfy different constraints. As a result, it is now possible for certain application domains to discover frequently occurring subgraphs in a reasonable amount of time. Moreover, since these patterns can be used as input to other data mining tasks (e.g., clustering and classification [6, 14]), the frequent pattern discovery algorithms play an important role in further expanding the use of data mining techniques to graph-based datasets.

A key characteristic of all of these algorithms is that they are complete in the sense that they are guaranteed to find all subgraphs that satisfy the specific constraints. Even though completeness is intrinsically a very important and desirable property, one can not ignore the fact that it also imposes very strong limitations on the types of graph datasets that can be mined in a reasonable amount of time. In general, the complete algorithms that are available today, can only operate efficiently on input datasets that are sparse, contain a large number of relatively small connected components, have vertices with low and bounded degrees, and contain well-labeled vertices and edges. Almost all datasets used during the experimental evaluations of complete algorithms satisfy these requirements, and they are often derived form chemical compounds [17, 1, 32, 13, 33, 19]. These restrictions are a direct consequence of the fact that due to the completeness requirements these algorithms must perform numerous subgraph isomorphism operations (or their equivalent) explicitly or implicitly that are known to be NP-complete [8]. On the other hand, existing heuristic algorithms, which are not guaranteed to find the complete set of subgraphs, as SUBDUE [2] and GBI [34], tend to find

[*]This work was supported in part by NSF ACI-0133464 and ACI-0312828; the Digital Technology Center at the University of Minnesota; and by the Army High Performance Computing Research Center (AHPCRC) under the auspices of the Department of the Army, Army Research Laboratory (ARL) under Cooperative Agreement number DAAD19-01-2-0014. The content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

an extremely small number of patterns and are not significantly more scalable. For example, the results reported in a recently published study showed that SUBDUE was able to find 3 subgraphs in 5,043 seconds, while VSIGRAM (a recently developed complete algorithm) was able to find 3,183 patterns in just 63 seconds from a graph containing 33,443 vertices and 11,244 edges [22].

There are many application domains that lead to datasets that have inherently structural or relational characteristics, are suitable for graph-based representations, and can greatly benefit from graph-based data mining algorithms (e.g., network topology, VLSI circuit design, protein-protein interactions, biological pathways, web graph, etc). However, the characteristics of the graph datasets derived from these application domains are such that they violate many of the implicit requirements of existing complete algorithms, as they are either significantly less sparse, contain very large connected components, have a high degree variations, and/or contain very few vertex and edge labels. As a result, these graphs can not be effectively mined by existing algorithms.

To overcome the limitations of existing algorithms (either complete or heuristic) we developed an algorithm called GREW. GREW is a heuristic algorithm, designed to operate on a large graph and to find patterns corresponding to connected subgraphs that have a large number of vertex-disjoint embeddings. Because of its heuristic nature, the number of patterns discovered by GREW is significantly smaller than those discovered by complete algorithms. However, as our experiments will show, GREW can operate effectively on very large graphs (containing over a quarter of a million vertices) and still find long and meaningful/interesting patterns. At the same time, compared to existing heuristic algorithms GREW is able to find significantly more and larger patterns at a fraction of their runtime.

GREW discovers frequent subgraphs by repeatedly merging the embeddings of existing frequent subgraphs that are connected by one or multiple edges. The key to GREW's efficiency is that it maintains the location of the embeddings of the previously identified frequent subgraphs by rewriting the input graph. This idea of edge contraction and graph rewriting is similar in spirit to that used by other heuristic algorithms [2, 34, 23]. However, GREW substantially extends it by (i) allowing the concurrent contraction of different types of edge, (ii) employing a set of heuristics that allows it to find longer and denser frequent patterns, and (iii) using a representation of the rewritten graph that retains all the information present in the original graph. The first two extensions allows GREW to simultaneously discover multiple patterns and find longer patterns in fewer iterations, whereas the third extension enables GREW to precisely identify whether or not there is an embedding of a particular subgraph, and thus guarantee a lower bound on the frequency of each pattern that it discovers.

We experimentally evaluate the performance of GREW on four different datasets containing 29,014–255,798 vertices that are derived from different domains including cocitation analysis, VLSI, and web link analysis. Our experiments show that GREW is able to quickly find a large number of non-trivial size patterns.

The rest of this paper is organized as follows. Section 2 provides necessary terminology and notation. Section 3 provides a detailed description of GREW and its various computational steps. Section 4 illustrates how GREW can be used as the building block in developing algorithms to solve some general mining problems. Section 5 shows a detailed experimental evaluation of GREW on datasets from different domains and compares it against existing algorithms. Section 6 surveys the related research and contrasts it with GREW. Finally, Section 7 provides concluding remarks.

2 Definitions and Notation

A **graph** $G = (V, E)$ is made of two sets, the set of vertices V and the set of edges E. Each edge is represented as an unordered pair of vertices. Throughout this paper we assume that a graph is undirected, and that the vertices and edges in a graph are **labeled**. The labels of an edge e and a vertex v are denoted by $l(e)$ and $l(v)$ respectively. Each vertex (or edge) of a graph is not required to have a unique label and the same label can be assigned to many vertices (or edges) in the same graph.

Given a graph $G = (V, E)$, a graph $G_s = (V_s, E_s)$ is a **subgraph** of G if $V_s \subseteq V$ and $E_s \subseteq E$, and is denoted by $G_s \subseteq G$. The subgraph G_s is said to be **covered** by G. If a subgraph $G_s \subseteq G$ is isomorphic to another graph H, then G_s is called an **embedding** of H in G. In this paper, a subgraph is often called a **pattern**. The total number of embeddings of G_s in a graph G is called the **raw frequency** of G_s.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** (denoted by $G_1 \simeq G_2$) if they are topologically identical to each other, that is, there is a vertex mapping from V_1 to V_2 such that each edge in E_1 is mapped to a single edge in E_2 and vice versa. In the case of labeled graphs, this mapping must also preserve the labels on the vertices and edges. When a set of graphs $\{G_i\}$ are isomorphic to each other, they all are said to belong to the same **equivalence class**. When the equivalence class of G_i represents an edge, the class is called an **edge-type**.

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of **subgraph isomorphism** is to find an isomorphism between G_2 and a subgraph of G_1. In other words, the subgraph isomorphism problem is to determine whether or not G_2 is embedded in G_1.

Given a subgraph G_s and a graph G, two embeddings of G_s in G are called **identical** if they use the same set of edges of G, **edge-disjoint** if they do not have any edges of G in common, and **vertex-disjoint** if no vertices of G in common. Given a set of all embeddings of a particular subgraph G_s in a graph G, the **overlap graph** of G_s is a graph obtained by creating a vertex for each non-identical embedding and creating an edge for each pair of non-vertex-disjoint embeddings.

Contraction of an edge $e = uv$ of a graph $G = (V, E)$ is to merge two endpoints u and v together into a new vertex w by removing the edge e, while keeping all the other edges incident to u and v. The remaining edges that used to be incident to either u or v are connected to w after the contraction. The newly added vertex w represents the original edge e. Note that, if there are multiple edges between two vertices u and v, the contraction of e removes only e. The rest of the multiple edges between u and v become loops around w after the contraction.

2.1 Canonical Labeling

One of the key operations required by any frequent subgraph discovery algorithm is a mechanism by which to check whether two subgraphs are identical or not. One way of performing this check is to perform a graph isomorphism operation. However, in cases in which many such checks are required among the same set of subgraphs, a better way of performing this task is to assign to each graph a unique *code* (i.e., a sequence of bits, a string, or a sequence of numbers) that is invariant on the ordering of the vertices and edges in the graph. Such a code is referred to as the **canonical label** of a graph $G = (V, E)$ [29, 7], and we will denote it by $\mathrm{cl}(G)$. By using canonical labels, we can check whether or not two graphs are identical by checking to see whether they have identical canonical labels. Moreover, the canonical labels allow us to uniquely identify a particular vertex in a subgraph, regardless of the original vertex and edge ordering in the subgraph.

Even though the worst-case complexity of canonical labeling may be exponential on the number of vertices, its average-time complexity can be reduced by using various heuristics to narrow down the search space or by using alternate canonical label definitions that take advantage of special properties that may exist in a particular set of graphs [25, 24, 7]. As part of our earlier research we have developed such canonical labeling algorithm that fully makes use of edge and vertex labels for fast processing and various vertex invariants to reduce the complexity of determining the canonical label of a graph [20, 21]. Our algorithm can compute the canonical label of graphs containing up to 50 vertices extremely fast and will be the algorithm used to compute the canonical labels of the different subgraphs in this paper.

Note that the patterns found by GREW are often larger than those found by the complete algorithms (as GREW can operate with much smaller minimum frequency threshold). It is necessary to adopt an efficient canonical labeling scheme to handle large subgraphs in a reasonable amount of time. Naive methods that require to permute columns and rows of the adjacency matrix of a subgraph without partitioning vertices do no scale well.

3 GREW—Scalable Frequent Subgraph Discovery Algorithm

GREW is a heuristic algorithm, designed to operate on a large graph and to find patterns corresponding to connected subgraphs that have a large number of vertex-disjoint embeddings. Specifically, the patterns that GREW finds satisfy the following two properties:

Property 1 The number of vertex-disjoint embeddings of each pattern is guaranteed to be at least as high as the user-supplied minimum frequency threshold.

Property 2 If a vertex contributes to the support of multiple patterns $\{G_1, G_2, \ldots, G_k\}$ of increasing size, then G_i is a subgraph of G_{i+1} for $i = 1, \ldots, k-1$.

The first property ensures that the patterns discovered by GREW will be frequent. However, GREW is not guaranteed to find all the vertex-disjoint embeddings of each pattern that it reports, nor is it guaranteed to find all the patterns that have a sufficiently large number of vertex-disjoint embeddings. As a result, GREW will tend to undercount the frequency of the patterns that it discovers and will miss some patterns. Moreover, the second property imposes some additional constraints on the types of patterns that it can discover, as it does not allow each vertex to contribute to the support of patterns that do not have a subgraph/supergraph relationship. As a result of these properties, the number of patterns that GREW discovers is significantly smaller to those discovered by complete algorithms such as SIGRAM [22].

GREW discovers frequent subgraphs in an iterative fashion. During each iteration, GREW identifies vertex-disjoint embeddings of subgraphs that were determined to be frequent in previous iterations and merges certain subgraphs that are connected to each other via one or multiple edges. This iterative frequent subgraph merging process continues until there are no such candidate subgraphs whose combination will lead to a larger frequent subgraph. Note that unlike existing subgraph growing methods used by complete algorithms [16, 19, 32, 22], which increase the size of each successive subgraph by one edge or vertex at a time, GREW, in each successive iteration, can potentially double the size of the subgraphs that it identifies.

The key feature that contributes to GREW's efficiency is that it maintains the location of the embeddings of the previously identified frequent subgraphs by rewriting the input graph. As a result of this graph rewriting, the vertices involved in each particular embedding are *collapsed* together to form a new vertex (referred to as **multi-vertex**), whose label uniquely identifies the particular frequent subgraph that is supported by them. Within each multi-vertex, the edges that are not part of the frequent subgraph are added as loop edges. To ensure that the rewritten graph contains all the information present in the original graph, these newly created loop-edges, as well as the edges of the original graph that are incident to a multi-vertex, are augmented to contain information about (i) the label of the incident vertices, and (ii) their actual end-point vertices within each multi-vertex (with respect to the original graph). Using the above representation, GREW identifies the sets of embedding-pairs to be merged by simply finding the frequent edges that have the same augmented edge label. In addition, GREW obtains the next level rewritten graph by simply contracting together the vertices that are incident to the selected edges.

3.1 Graph Representation

GREW represents the original input graph G as well as the graphs obtained after each successive rewriting operation in a unified fashion. This representation, referred to as the *augmented graph*, is designed to contain all necessary information by which we can recover the original input graph G from any intermediate graph obtained after a sequence of rewriting operations.

Each vertex v and edge e of the augmented graph $\hat{G}$ has a label associated with it, which is denoted by $\hat{l}(v)$ and $\hat{l}(e)$, respectively. In addition, unlike the original graph that is simple, the augmented graph can contain loops. Furthermore, there can be multiple loop edges associated with each vertex and there can be multiple edges connecting the same pair of vertices. However, whenever there exist such multiple loops or edges, the augmented label of each individual edge will be different from the rest.

The label of each vertex v in the augmented graph depends on whether or not it corresponds to a single vertex or a multi-vertex obtained after collapsing together a set of vertices that are used by an embedding of a particular subgraph. In the former case, the label of the vertex is identical to its label in the original graph, whereas in the latter case, its label is determined by the canonical labeling (Section 2.1) of its corresponding subgraph. This canonical-labeling-based approach ensures that the multi-vertices representing the embeddings of the same subgraphs will be uniquely assigned the same label.

To properly represent edges that are connected to multi-vertices, the augmented graph representation assigns a label to each edge that is a tuple of five elements. For an edge $e = uv$ in an augmented graph $\hat{G}$, this tuple is denoted by $(\hat{l}(u), \hat{l}(v), l(e), e.\text{epid}(u), e.\text{epid}(v))$, where $\hat{l}(u)$ and $\hat{l}(v)$ are the labels of the vertices u and v in $\hat{G}$, $l(e)$ is the original label of the edge e, and $e.\text{epid}(u)$ and $e.\text{epid}(v)$ are two numbers, referred to as **endpoint identifiers**, that uniquely identify the specific pair of G's vertices within the subgraphs encapsulated by u and v that e is incident to. The endpoint identifiers are determined by first ordering the original vertices in u and v according to their respective canonical labeling, and then using their rank as the endpoint identifier. If an endpoint is not a multi-vertex, but just a plain vertex, the endpoint identifier is always set to zero.

Since the endpoint identifiers are derived from the canonical labels of u and v, it is easy to see that this approach will correctly assign the same five elements to all the edges that have the same original label and connect the same pair of subgraphs at exactly the same vertices. However, to ensure that topologically equivalent edges can be quickly identified by comparing their tuple representation, the order of the tuple's elements must be determined in a consistent fashion. For this reason, given an edge $e = uv$, the precise tuple representation is defined as follows:

$$\begin{aligned}
&(\hat{l}(u), \hat{l}(v), l(e), e.\text{epid}(u), e.\text{epid}(v))\\
&\qquad \text{if } \hat{l}(u) < \hat{l}(v), \text{ or}\\
&\qquad \text{if } \hat{l}(u) = \hat{l}(v) \text{ and } e.\text{epid}(u) \le \text{epid}(v)\\
&\text{or}\\
&(\hat{l}(v), \hat{l}(u), l(e), e.\text{epid}(v), e.\text{epid}(u))\\
&\qquad \text{if } \hat{l}(u) > \hat{l}(v), \text{ or}\\
&\qquad \text{if } \hat{l}(u) = \hat{l}(v) \text{ and } e.\text{epid}(u) > \text{epid}(v)
\end{aligned}$$

This consistent tuple representation ensures that all the edges that share the same label in the augmented graph correspond to identical subgraphs in the original graph.

Note that loops and multiple edges can also be represented by these five-element tuples, and the augmented graph representation treats them like ordinary edges.

3.2 GREW-SE—Single-Edge Collapsing

The simplest version of GREW, which is referred to as GREW-SE, operates on the augmented graph and repeatedly identifies frequently occurring edges and contracts them in a heuristic fashion.

The overall structure of GREW-SE is shown in Algorithm 1. It takes as input the original graph G and the minimum frequency threshold f, and on completion, it returns the set of frequent subgraphs $\mathcal{F}$ that it identified. During each iteration (loop starting at line 5), it scans the current augmented graph $\hat{G}$ and determines the set of edge-types $\mathcal{E}$ that occur at least f times in $\hat{G}$. This is achieved by comparing the labels of the various edges in $\hat{G}$ and determining those edge-types that occur at least f times.

Algorithm 1 GREW-SE(G, f)

```
 1: ▷ $G$ is the input graph.
 2: ▷ $f$ is the minimum frequency threshold.
 3: $\mathcal{F} \leftarrow \emptyset$
 4: $\hat{G} \leftarrow$ augmented graph representation of $G$
 5: while true do
 6:     $\mathcal{E} \leftarrow$ all edge-types in $\hat{G}$ that occur at least $f$ times
 7:     order $\mathcal{E}$ in decreasing frequency
 8:     for each edge-type $e$ in $\mathcal{E}$ do
 9:         $G_o \leftarrow$ overlap graph of $e$
10:         ▷ each vertex in $G_o$ corresponds to an embedding of $e$ in $\hat{G}$
11:         $\mathcal{M}_{\text{MIS}} \leftarrow$ obtain MIS for $G_o$
12:         $e.f \leftarrow |\mathcal{M}_{\text{MIS}}|$
13:         if $e.f \geq f$ then
14:             $\mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$
15:             for each embedding $m$ in $\mathcal{M}_{\text{MIS}}$ do
16:                 mark $m$
17:     if no marked edge in $\hat{G}$ then
18:         break
19:     update $\hat{G}$ by rewriting all of its marked edges
20: return $\mathcal{F}$
```

From the discussion in Section 3.1, we know that each of these edge-types represent identical subgraphs, and as a result each edge-type in $\mathcal{E}$ can lead to a frequent subgraph. However, because some vertices can be incident to multiple embeddings of the same (or different) frequent edge-types, the frequencies obtained at this step represent upper-bounds, and the actual number of the vertex-disjoint embeddings can be smaller. For this reason, GREW-SE further analyzes the embeddings of each edge-type to select a maximal set of embeddings that do not share any vertices with each other or with embeddings selected previously for other edge-types. This step (loop starting at line 8) is achieved by constructing the overlap graph G_o for the set of embeddings of each edge-type e and using a greedy maximal independent set algorithm [10] to quickly identify a large number of vertex-disjoint embeddings. If the size of this maximal set is greater than the minimum frequency threshold, this edge-type survives the current iteration and the embeddings in the independent set are marked. Otherwise the edge-type is discarded as it does not lead to a frequent subgraph in the current iteration. After processing all the edge-types, the contraction operations are performed, graph $\hat{G}$ is updated, and the next iteration begins.

In order to illustrate some of the steps performed by GREW-SE, let us take the simple example shown in Figure 1 in which the original graph (Figure 1 (a)) contains two squares connected to each other by an edge. Assume all the edges have the same label at the beginning, while there are two distinct vertex labels (the white and the slightly shaded ones). Edge contraction process proceeds as illustrated in (b), (c), (d) and (e), assuming the edge-types are selected in decreasing order of the raw frequency (each edge-type is represented by a capital letter and the raw frequency of each edge-type is also shown in the figure) . Every time an edge is contracted, a new multi-vertex is created whose label identifies a subgraph that the multi-vertex represents (shown by the difference in shading and filling pattern of vertices). Note that at the end of this sequence of edge collapsing, the two squares originally existed in Figure 1 (a) are represented by two black vertices in Figure 1 (e).

3.3 GREW-ME—Multi-Edge Collapsing

As discussed in Section 3.1, a result of successive graph rewriting operations is the creation of multiple loops and multiple edges in $\hat{G}$. In many cases, there may be the same set of multiple edges connecting similar pairs of vertices in $\hat{G}$, all of which can be collapsed together to form a larger frequent subgraph. GREW-SE can potentially identify such a subgraph by collapsing a sequence of single-edges (which after the first iteration, each successive iteration will involve loop edges). However, this will require multiple iterations and owing to the heuristic nature of the overall algorithm, it may fail to *orchestrate* the proper sequence of steps.

To address this problem, we developed the GREW-ME algorithm that in addition to collapsing vertices connected via a single edge, it also analyzes the sets of multiple edges connecting pairs of vertices to identify any frequent subsets of edges. This is achieved by using a traditional frequent closed itemset mining algorithm (e.g., [27, 36, 28]) as follows. For each pair of vertices that are connected via multiple edges (or a single vertex with multiple loops), GREW-ME creates a list that contains the multiple edge-types that are involved, and treats each list as a transaction whose items corresponds to the multiple edges. Then, by running a closed frequent itemset mining algorithm, GREW-ME finds all the frequent sets of edges whose raw frequency is above the minimum threshold. Each of these multiple sets of edges is treated as a different edge-type, and GREW-ME proceeds in a fashion identical to GREW-SE.

3.4 Discussion

As discussed in the beginning of Section 3, the frequent subgraphs discovered by GREW satisfy properties 1 and 2. In general, these properties can be satisfied by multiple sets of frequent subgraphs that in many cases are in conflict with each other. That is, frequent subgraphs from different sets cannot be arbitrarily combined into a new set and still satisfy these two properties.

To a large extent, the particular set of frequent subgraphs that are identified by GREW is determined by (i) the order in which the different edge-types are analyzed to determine their vertex-disjoint embeddings, and (ii) the particular set of maximal independent embeddings that are selected by the maximal independent set algorithm. The order of the edge-types is important because during each iteration, a vertex of $\hat{G}$ can only participate in a single con-

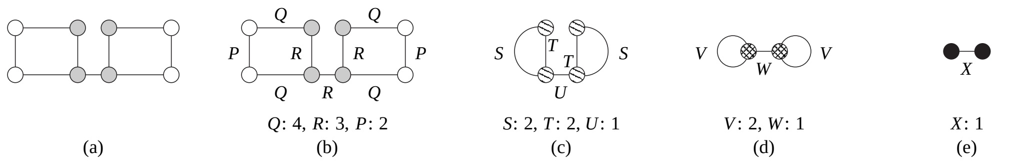


Figure 1. Sequence of edge collapsing operations performed by GREW-SE

traction operation. As a result, edge-types considered earlier have a greater chance in leading to a large number of vertex-disjoint embeddings and thus resulting in a frequent subgraph. At the same time, the particular maximal set of independent embeddings that is selected can directly or indirectly impact the discovered subgraphs. The direct impact occurs when the size of the selected independent set happens to be smaller than the minimum frequency threshold (when the maximum size independent set is above the threshold), in which case the subgraph will not be identified as frequent. The indirect impact occurs when the chosen independent set negatively affects the number of vertex-disjoint embeddings of some of the edge-types that are considered afterward.

GREW performs each of these steps by employing relatively simple heuristics that are designed to maximize the frequency of the discovered patterns. Specifically, GREW orders the different edge-types in decreasing raw frequency and uses a greedy maximal independent set algorithm to identify the largest possible independent set. However, each of these two steps can be performed using different heuristics that can guide/bias GREW to find/prefer one set of subgraphs over the others.

Specifically, in the course of developing GREW, we experimented with a number of different schemes for each of these two steps. For example, we developed a scheme that instead of using the raw frequency of each edge-type, it used the greedy maximal independent set algorithm to estimate the number of vertex-disjoint embeddings that it has, and sorted the edge-types based on this estimate. Similarly, we developed a scheme that selected the set of independent embeddings such that the rewriting on those embeddings will lead to the identical or similar local structure after the contraction. The motivation behind this heuristic is to prefer the embeddings that have a better chance to grow into a larger frequent subgraph in subsequent iterations. Our experiments with both of these schemes showed that they led to different subgraphs, although the overall number and size of the discovered subgraphs did not change dramatically.

4 Mining a Graph using GREW

A key limitation of GREW is that it tends to find a very small number of frequent subgraphs compared to complete algorithms [23, 22]. This is primarily due to the fact that its graph rewriting-based approach substantially constraints the sets of frequent subgraphs that are allowed to have overlapping embeddings (property 2 in Section 3), and secondary due to the fact that it underestimates the frequency of each subgraph and consequently it may miss subgraphs that are actually frequent.

However, because of its low computational requirements, GREW can be used as a building block to develop *meta-strategies*, which can be used to effectively mine very large graphs in reasonable amount of time. In the rest of this section we present two such schemes. The first uses GREW to find a diverse set of frequent subgraphs that can cover the entire lattice of frequent subgraphs, whereas the second scheme uses GREW to find a set of frequent subgraphs whose embeddings can cover as much of the input graph as possible.

The first set of subgraphs can be used to obtain the same type of patterns as most obtained by complete algorithms, whereas the second set of subgraphs can be used to obtain a concise representation of the input graph which can be employed for summarization purposes or as an input to other data mining tasks.

4.1 Covering the Pattern Lattice

The approach that we adopted to find a diverse set of frequent subgraphs is composed of two elements. First, since GREW tends to undercount the frequency of the subgraphs that it discovers, we mine the input graph with a lower frequency threshold. Second, since GREW finds a limited set of patterns, we invoke GREW multiple times, each time biasing it toward discovering a different set of frequent subgraphs, and return the union of the frequent subgraphs that were found across the multiple runs.

We developed two different schemes for biasing each successive run. Both of them rely on the observations made in Section 3.4, which identified GREW's key operations that affect the set of subgraphs that it identifies. The first scheme, instead of considering the different edge-types in decreasing order of their raw frequency, it considers them in a random order, which is different for each successive run. Since each run of GREW will tend to favor different sets of edge-types, the frequent subgraphs identified in successive runs will be somewhat different. We will refer to this as the *simple randomization scheme*.

The second scheme uses the same random traversal strategy as before, but also tries to bias the patterns to different edges from those used earlier. This is done by maintaining statistics as to which and how many times each edge was used to support a frequent subgraph identified by earlier iterations, and biases the algorithm that selects the maximal independent set of embeddings so that it will prefer edges that have not been used (or used infrequently) in previously discovered subgraphs. Thus, this scheme tries to learn from previous invocations, and for this reason it will be referred to as the *adaptive randomization scheme*.

4.2 Covering the Graph

The approach that we adopted to find frequent subgraphs whose embeddings can cover as much of the original graph as possible combines the randomization schemes used earlier with the well-known sequential covering paradigm used for building rule-based classifiers [26].

Specifically, after each run of the simple randomization scheme, every edge that belongs to an embeddings of a frequent subgraph is removed. The size of the input subgraph swiftly decreases, and the removal of these edges forces the algorithm to use unused edges. As shown in Section 5.3.1, this scheme enables us to cover the input dataset with frequent subgraphs by performing a small number of different runs.

5 Experimental Evaluation

In this section, we study the performance of the proposed algorithms with various parameters and real datasets. All experiments were done on Intel Pentium 4 processor (2.6 GHz) machines with 1 Gbytes main memory, running the Linux operating system. All the reported runtimes are in seconds.

5.1 Datasets

We evaluated the performance of GREW on four different datasets, each obtained from a different domain. The basic characteristics of these datasets are shown in Table 1. Note that even though some of these graphs consist of multiple connected components, GREW treats them as one large graph.

The *Aviation* dataset is obtained from the SUBDUE web site[1]. This dataset is originally from the Aviation Safety Reporting System Database. The directed edges in the original graph data were converted into undirected ones. The *VLSI* dataset was obtained from the International Symposium on Physical Design '98 (ISPD98) benchmark `ibm05`[2] and corresponds to the netlist of a real circuit. The netlist

[1] http://cygnus.uta.edu/subdue/databases/index.html

[2] http://vlsicad.cs.ucla.edu/~cheese/ispd98.html

Table 1. Datasets used in the experiments

Dataset	Vertices	Edges	Labels		Connected
			Vertex	Edge	Components
Aviation	101,185	133,113	6,173	52	1,049
Citation	29,014	294,171	742	1	3,018
VLSI	29,347	81,353	11	1	1
Web	255,798	317,247	3,438	1	25,685

was converted into a graph by using a star-based approach to replace each net (i.e., hyperedge) by a set of edges. The *Citation* dataset was created from the citation graph used in KDD Cup 2003[3]. Each vertex in this graph corresponds to a document and each edge corresponds to a citation relation. Because our algorithms are for undirected graphs, the direction of these citations was ignored. Since the original graph has no meaningful labels on either the vertices or the edges, we assigned only to vertices labels obtained from the subdomains of the first author's email address. Self-citations based on this vertex assignment were removed. Finally, the *Web* dataset was obtained from the 2002 Google Programming Contest[4]. The original dataset contains various webpages and links from various "edu" domain. We converted the dataset into an undirected graph in which each vertex corresponds to a web-page and an edge to a hyperlink between web-pages. In creating this graph, we kept only the links between "edu" domains that connected sites from different subdomains. Every edge has an identical label (i.e., unlabeled), whereas each vertex was assigned a label corresponding to the subdomain of the web-server.

5.2 Single Invocation Performance

Table 2 shows the runtime, the number of frequent patterns found, and the size of the largest frequent patterns obtained by GREW-SE and GREW-ME for the four datasets. It also shows the faction of the vertices and edges in the input graph that are not covered by any of the frequent patterns. In calculating these values, we excluded the vertices and the edges that can not be frequent because of their labels.

Both GREW-SE and GREW-ME can find large frequent patterns in a reasonable amount of time. For example, GREW-SE can mine the Web dataset, which contains over 250,000 vertices, with the minimum frequency of five in around four minutes. Looking at the characteristics of the algorithms, as the minimum frequency threshold decreases, we can see that, as expected, they are able to find both a larger number of frequent patterns and patterns that are in general longer.

Comparing the relative performance of GREW-SE and GREW-ME, we can see that overall, they perform quite similarly, as they both find similar number of patterns, and their longest patterns are of similar sizes. However, there

[3] http://www.cs.cornell.edu/projects/kddcup/datasets.html

[4] http://www.google.com/programming-contest/

Table 2. GREW-SE and GREW-ME

Dataset	Method	f	t [sec]	#	Size	Unused vs	Unused es
Aviation	GREW-SE	1000	336	38	13	22	43
		500	213	72	16	41	57
		200	129	117	32	49	62
		100	2151	175	53	56	68
	GREW-ME	1000	370	38	13	22	43
		500	396	72	16	41	57
		200	ME				
		100	ME				
Citation	GREW-SE	100	9	87	3	55	63
		50	20	150	7	63	78
		20	48	306	16	65	90
		10	105	533	31	59	94
		5	683	1061	63	50	96
	GREW-ME	100	18	86	3	55	63
		50	34	150	7	63	78
		20	74	305	8	64	91
		10	130	546	25	59	94
		5	555	1077	47	50	96
VLSI	GREW-SE	100	20	54	17	14	77
		50	20	84	18	9	75
		20	23	145	43	6	73
		10	23	239	27	4	72
		5	29	445	33	3	70
	GREW-ME	100	23	55	15	14	77
		50	23	90	18	8	74
		20	24	146	20	6	73
		10	26	238	36	4	71
		5	38	417	44	3	70
Web	GREW-SE	100	6	296	1	85	80
		50	15	554	3	84	85
		20	45	1254	7	80	86
		10	89	2430	9	75	86
		5	259	4822	13	69	84
	GREW-ME	100	8	296	1	85	80
		50	20	553	3	84	85
		20	66	1256	5	80	86
		10	220	2461	14	74	86
		5	ME				

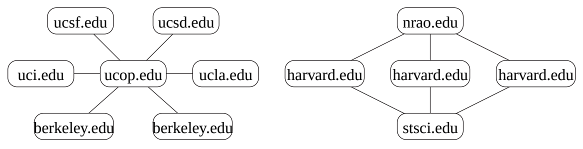
Note. "ME" indicates the computation was aborted because of memory exhaustion.
f: the minimum frequency threshold
t: runtime in seconds
#: the number of frequent patterns discovered
Size: the size of the largest frequent patterns found
Unused: the fraction (%) of vertices (vs) and edges (es) in the input graph that are not covered by any of the frequent patterns

are some dataset dependencies. For example, GREW-SE performs better for the Citation dataset, whereas GREW-ME performs better for the VLSI dataset. In terms of runtime, GREW-ME is somewhat slower than GREW-SE. This is because (i) GREW-ME incurs the additional overhead of finding closed frequent itemsets, and (ii) it processes a larger number of distinct edge-types (as each closed itemset is represented by a different edge-type). In addition, the memory overhead associated with storing these larger number of edge-types is the reason why GREW-ME run out of memory for some parameter combinations with the Aviation and Web datasets. We are currently developing alternative implementations and explore schemes to select the most promising closed itemsets, to reduce the memory overhead of GREW-ME.

Note that from the results in Table 2 we can see that the percentage of unused vertices and edges does not monotonically decrease as we decrease the minimum frequency. This is primarily an artifact of the way that we compute these statistics, as the fraction of the used vertices/edges over the number of vertices/edges that meet the minimum frequency threshold. As a result, as the minimum frequency decreases, the baseline frequencies will tend to increase, leading to higher fractions. In all cases, the absolute number of vertices and edges that were used over the entire set of frequent patterns decreases with the minimum frequency.

Example Subgraphs To illustrate the types of subgraphs that GREW can discover, we analyzed the subgraphs that were identified in the Web dataset. Recall from Section 5.1 that each vertex in this graph corresponds to an actual web-page, each edge to a hyperlink between two web-pages, and the vertex-labels to the subdomain of the server that hosts the web-page. Moreover, this graph was constructed by removing any hyperlinks between web-pages that have the same subdomain. As a result, a frequently occurring subgraph will represent a particular cross-linking structure among a specific set of institutions that occurs often, and it can identify common cross-university collaborations, interdisciplinary teams, or topic-specific communities.

Figure 2 shows two representative examples of the subgraphs discovered by GREW. The first subgraph (Figure 2(a)) has a star topology and connects together various web-servers that are part of California's University System. The star-node corresponds to web-servers that are part of the University of California's Office of the President with various web-servers that are located at Berkeley, UCI, UCLA, UCSD, and UCSF. The second subgraph (Figure 2(b)) has a more complex topology with a higher degree of connectivity and connects together various web-servers at Harvard, National Radio Astronomy Observatory (nrao.edu), and Space Telescope Science Institute (stsci.edu). An analysis of the complete uniform resource locators (URLs) of the embeddings of this subgraph showed that all the web-pages had to do with astronomy and astrophysics. These examples suggest that the patterns that GREW finds are interesting and can be used to gain insights on the underlying graph datasets.



(a) University of California System (b) Astronomy related web pages

Figure 2. Examples of patterns discovered by GREW

5.3 Multiple Invocation Performance

As discussed in Section 4, GREW can be used as a building block to develop effective meta-strategies that mine very large graphs and discover a large set of frequent subgraphs that satisfy certain characteristics. In the rest of this section we evaluate the performance of the randomization and sequential covering strategies discussed in that section.

5.3.1 Simple and Adaptive Randomization Schemes

Table 3 shows the performance and characteristics of the subgraphs discovered by multiple runs (ranging from one to ten) of GREW-SE and GREW-ME for the cases in which these multiple runs were performed using the simple and the adaptive randomization schemes.

From these results we can see that the two randomization schemes are quite effective in allowing GREW to find a larger number of frequent subgraphs. As the number of runs increases, both the number of frequent patterns and the size of the largest frequent patterns increase monotonically. As expected, there is a certain degree of overlap between the patterns found by different runs, and for this reason the distinct number of subgraphs does not increase linearly. In addition, the set of vertices and edges of the original graph that are covered by the discovered frequent subgraphs also increases with the number of runs. For all the datasets, after invoking GREW-SE and GREW-ME ten times, the resulting set of frequent subgraphs *cover* more than 50% of the vertices and/or edges of the input graph. This suggests that GREW is able to find a diverse set of frequent subgraphs that captures a good fractions of the input graph.

Comparing the relative performance of GREW-SE and GREW-ME within the context of this randomization framework, we can see that GREW-ME tends to find a larger number of distinct frequent subgraphs whose maximum size is larger than the subgraphs discovered by GREW-SE. This indicates that GREW-ME's ability to identify multiple edges and collapse vertices that are connected by them becomes more effective in increasing the diversity of the discovered patterns in the context of this randomization strategy.

Finally, comparing the relative performance of the two randomization schemes we can see that, as expected, the adaptive randomization scheme improves the pattern discovery process as it finds a larger number of distinct patterns than the simple randomization. However, the overall size of the patterns identified by both schemes remains the same, as they both cover similar fractions of the vertices and/or edges of the input graph.

Pattern Lattice Coverage The static and adaptive randomization schemes were introduced as a meta-strategy to identify the set of frequent subgraphs that can be used to approximate the subgraphs discovered by a complete algorithm; that is, cover the entire pattern lattice. Unfortunately, it is not possible to evaluate the effectiveness of these schemes on the four datasets used in our benchmarks, because owing to their size and/or density, complete algorithms cannot finish in a reasonable amount of time. For this reason, we evaluated the effectiveness of this meta-strategy on a smaller dataset consisting of various chemical compounds (containing 51,101 vertices and 54,887 edges with 28 vertex-labels and 3 edge-labels), for which the existing complete algorithms can operate effectively.

Table 4 shows the results obtained by the complete algorithm VSIGRAM and GREW-SE with the static randomization scheme. This table shows the results obtained by VSIGRAM with a minimum frequency threshold of 100, and the results obtained by GREW-SE after 1, 10, and 100 runs and a minimum frequency thresholds of 100, 50, and 10. The last column of this table (labeled "Coverage") shows the fraction of the frequent patterns in the lattice that are covered by any of the frequent patterns discovered by GREW-SE (see Section 2 for the definition of coverage).

These results show that the strategy discussed in Section 4.1, which invokes GREW multiple times while decreasing the minimum frequency threshold, is quite effective in identifying the set of subgraphs that covers a very large fraction of the overall lattice of frequent subgraphs. Moreover, the overall amount of time required by GREW is smaller than that required by VSIGRAM. For example, even after running GREW-SE 100 times with a minimum frequency threshold 10, its runtime is still about half of that of VSIGRAM, while the patterns found by GREW-SE can cover 97% of the frequent pattern lattice.

Table 4. Coverage of Frequent Pattern Lattice

VSIGRAM			GREW-SE					Coverage
f	t[sec]	#	f	Runs	t[sec]	#	Size	[%]
100	1161	13649	100	1	4	62	7	3
				10	49	164	10	3
				100	579	343	10	9
			50	1	9	102	9	3
				10	46	338	10	8
				100	594	765	12	33
			10	1	3	485	40	46
				10	59	2272	40	70
				100	560	8992	51	97

Note. for GREW-SE, the simple randomization is used.
f: the minimum frequency threshold
t: runtime in seconds
Runs: the number of runs
#: the number of frequent patterns discovered
Size: the size of the largest frequent patterns found
Coverage: the fraction (%) of frequent patterns in the lattice that are covered by any of the frequent patterns discovered by GREW-SE

5.3.2 Sequential Covering Scheme

Table 5 shows the results obtained by GREW-SE and GREW-ME when their successive runs were performed using the se-

Table 3. Simple and Adaptive Randomization

Dataset	f	Method	Runs	Simple Randomization					Adaptive Randomization				
				t [sec]	#	Size	Unused vs	Unused es	t [sec]	#	Size	Unused vs	Unused es
Aviation	100	GREW-SE	1	23	245	14	54	67	18	245	14	54	67
			2	46	386	14	44	59	35	386	14	44	59
			5	261	795	17	28	47	166	793	19	30	48
			10	423	1426	22	20	39	260	1476	19	19	39
		GREW-ME	1	49	233	14	55	68	37	233	14	55	68
			2	193	363	22	44	59	144	363	22	44	59
			5	345	754	22	28	47	252	754	22	23	47
			10	615	1422	22	19	40	634	1434	40	18	39
Citation	10	GREW-SE	1	39	659	5	54	94	41	659	5	54	94
			2	82	881	5	45	92	86	881	5	44	92
			5	231	1340	7	36	88	224	1365	5	35	87
			10	461	1912	7	31	82	453	1940	8	30	80
		GREW-ME	1	188	658	5	56	94	189	658	5	56	94
			2	367	940	6	46	92	358	936	7	46	92
			5	899	1527	7	37	88	916	1519	9	36	87
			10	1843	2311	8	32	82	2683	2319	9	30	80
VLSI	10	GREW-SE	1	12	394	20	6	73	12	389	21	6	73
			2	25	712	20	2	61	24	674	34	1	58
			5	74	1372	20	1	42	62	1452	34	0	34
			10	146	2335	21	0	27	140	2416	34	0	16
		GREW-ME	1	37	509	20	10	73	37	509	18	10	74
			2	83	959	22	3	58	74	933	21	3	57
			5	235	2049	30	1	38	202	1925	22	0	31
			10	440	3547	30	0	25	362	3403	27	0	14
Web	10	GREW-SE	1	298	2716	9	74	86	199	2716	9	74	86
			2	393	3268	9	69	82	395	3273	13	67	80
			5	992	4095	15	62	74	994	4155	13	58	70
			10	1970	4871	15	56	67	1974	4881	13	51	61
		GREW-ME	1	805	2719	14	74	86	550	2719	14	74	86
			2	1084	3249	14	69	82	978	3257	14	67	80
			5	2578	4138	16	62	74	2464	4158	14	58	70
			10	5074	4945	16	57	67	5175	4979	15	51	61

Note. f: the minimum frequency threshold
Runs: the number of randomized runs
t: runtime in seconds
#: the number of frequent patterns discovered
Size: the size of the largest frequent patterns found
Unused: the fraction (%) of vertices (vs) and edges (es) in the input graph that are not covered by any of the frequent patterns

quential covering scheme, which was introduced as a meta-strategy to identify a set of frequent subgraphs that covers as many of the vertices and/or edges of the input graph (Section 4.2). The combinations of tunable parameters of the experiment are identical to the ones used in Section 5.3.1.

From these results we can see that in general, this scheme identifies patterns that cover a larger fraction of the input graph, when compared to the characteristics of the subgraphs discovered by the two randomization strategies (Table 3). However, we should also note that the size of the largest patterns found by the sequential covering scheme is smaller than the size of the patterns found by the simple and the adaptive randomization scheme. Considering the mechanism of the sequential covering scheme, this is inevitable. The simple and adaptive randomization schemes can find patterns whose embeddings may overlap. However, the embeddings of the patterns found by the sequential covering scheme do not overlap. Because of this reason, the set of patterns found by the sequential covering scheme tend to have less diversity and be smaller.

5.4 Comparison with SUBDUE

We ran SUBDUE [12] version 5.1.0 (with the default set of parameters) on our four benchmark datasets and measured the runtime, the number of patterns discovered, their size, and their frequency. Although we gave SUBDUE eight hours to mine each of the datasets, SUBDUE could only finish within the eight hour window for the Aviation dataset. It took SUBDUE more than 6 hours and discovered three most interesting patterns according to the MDL principle. Their sizes are either 9 or 10 and their frequencies are all 13. On the other hand, as shown in Table 3, GREW-SE and GREW-ME with the adaptive randomization scheme can find patterns up to size 19 and 40, respectively whose frequency is at least 100, which is about 10 times more than the frequency of the best three patterns reported by SUBDUE. The runtime of GREW-SE and GREW-ME are also significantly shorter than that of SUBDUE. GREW-SE spends 260 seconds and GREW-ME spends 634 seconds for the 10 randomized runs.

Table 5. Sequential Covering Scheme

Dataset	f	Method	Runs	t [sec]	#	Size	Unused vs	Unused es
Aviation	100	GREW-SE	1	7	245	14	54	67
			2	9	274	14	49	63
			5	17	337	14	36	53
			10	28	403	14	22	41
		GREW-ME	1	17	233	14	55	68
			2	52	267	14	49	69
			5	435	332	14	37	53
			10	ME				
Citation	10	GREW-SE	1	41	659	5	54	94
			2	81	905	7	44	92
			5	224	1434	8	34	86
			10	452	9993	9	29	78
		GREW-ME	1	188	658	5	56	94
			2	372	936	7	45	92
			5	902	1571	7	34	86
			10	1799	2283	9	30	78
VLSI	10	GREW-SE	1	11	389	21	6	73
			2	21	740	21	1	52
			5	36	1163	21	0	15
			10	41	1213	21	0	2
		GREW-ME	1	36	509	18	10	74
			2	73	899	18	2	53
			5	139	1554	22	0	14
			10	147	1625	22	0	1
Web	10	GREW-SE	1	472	2716	9	74	86
			2	877	3412	9	67	80
			5	1963	4412	11	57	69
			10	3783	5078	11	51	60
		GREW-ME	1	1179	2719	14	74	86
			2	1952	3398	14	67	80
			5	3476	4410	14	57	69
			10	5768	5084	14	51	61

Note. "ME" indicates the computation was aborted
f: the minimum frequency threshold
Runs: the number of runs
t: runtime in seconds
#: the number of frequent patterns discovered
Size: the size of the largest frequent patterns found
Unused: the fraction (%) of vertices (vs) and edges (es) in the input graph that are not covered by any of the frequent patterns

6 Related Research

The previous research on finding frequent subgraphs in graph datasets falls under two categories. The first category contains algorithms that find subgraphs that occur multiple times in a single large graph [12, 9, 31, 22] and are directly related to the algorithms presented in this paper, whereas the second category contains algorithms that find subgraphs that occur frequently across a database of small graphs [34, 5, 15, 18, 20, 17, 32, 1], Between these two classes of algorithms, those developed for the latter problem are in general more mature as they have moderate computational requirements and scale to large datasets.

The most well-known algorithm for finding recurring subgraphs in a single large graph is the SUBDUE system, originally developed in 1994, but has been improved over the years [12, 2, 4, 3]. SUBDUE is an approximate algorithm and finds patterns that can compress the original input graph by substituting those patterns with a single vertex. In evaluating the extent to which a particular pattern can compress the original graph it uses the minimum description length (MDL) principle, and employs a heuristic the beam search to narrow the search-space. These approximations improve its computational efficiency but at the same time it prevents it from finding subgraphs that are indeed frequent. Motoda et al. developed an algorithm called GBI [34] which is similar to SUBDUE and later proposed the improved version called B-GBI [23] adopting the beam search. B-GBI is the closest algorithm to our study, in the sense that both perform the same basic operation to identify frequent patterns based on edge contraction. However, while B-GBI focuses on one edge-type at a time when collapsing the embeddings of the edge-type in a greedy manner, GREW identifies and contracts more than one edge-types concurrently using a greedy MIS algorithm. Because B-GBI works on a single edge-type at at time, it uses the beam search to compensate the greedy nature of the algorithm. On the other hand, we adopted the randomized process to increase the diversity of frequent patterns to be found, on top of the concurrent edge collapsing scheme of GREW. Furthermore, our algorithm employs various heuristics such as multi-edge collapsing and the adaptive and sequential schemes to ensure the coverage of the input graph by the frequent patterns. Unfortunately, because the current implementation of B-GBI is mainly designed for a set of graphs, not for a single large graph, it can not be directly compared with GREW. Ghazizadeh and Chawathe [9] developed an algorithm called SEuS that uses a data structure called *summary* to construct a compact representation of the input graph. This summary is obtained by collapsing together all the vertices of the input graph that have the same label and is used to quickly prune infrequent candidates. As the authors indicate, this summary data-structure is useful only when the input graph contains a relatively small number of frequent subgraphs with high frequency, and is not effective if there are large number of frequent subgraphs with low frequency. Vanetik, Gudes and Shimony [31] presented an algorithm for finding all frequently occurring subgraphs from a single labeled undirected graph using the maximum number of edge-disjoint embeddings of a graph as a measure of its frequency. Each subgraph is represented by its minimum number of edge-disjoint paths (*path number*), and use a level-by-level approach to grow the patterns based on their path-number. They presented a limited number of experiments illustrating the feasibility of such candidate subgraph generation approach; however, their experiments involved graphs with a very small number of vertices (around 100) making it impossible to determine its scalability. Kuramochi and Karypis developed an algorithm called SIGRAM to find frequent connected subgraphs from a single labeled sparse undirected graph [22]. SIGRAM followed the definition of the frequency that Venetik, Gudes and Shimony proposed [31], and further proposed heuristics to accelerate the mining process in the horizontal and vertical paradigm.

Five different algorithms have been developed capable of finding all frequently occurring subgraphs in a database of graphs with reasonable computational efficiency. These are the AGM algorithm developed by Inokuchi et al [15, 17], the FSG algorithm developed by Kuramochi and Karypis [20, 21], the chemical substructure discovery algorithm developed by Borgelt and Berthold [1], the gSpan algorithm developed by Yan and Han [32], and most recently FFSM by Huan, Wan and Prins [13]. The AGM algorithm initially developed to find frequently induced subgraphs [15] and later extended to find arbitrary frequent subgraphs [17] discovers the frequent subgraphs using a breadth-first approach, and grows the frequent subgraphs one-vertex-at-a-time. To distinguish a subgraph from another, it uses a canonical labeling scheme based on the adjacency matrix representation. The FSG algorithm initially presented in [20], with subsequent improvements presented in [21], uses a breadth-first approach to discover the lattice of frequent subgraphs. The size of these subgraphs is grown by adding one-edge-at-a-time, and the frequent pattern lattice is used to prune non downward closed candidate subgraphs. FSG employs a number of techniques to achieve high computational performance including efficient canonical labeling, efficient candidate subgraph generation algorithms, and various optimizations during frequency counting. The chemical substructure mining algorithm developed by Borgelt and Berthold [1], finds frequent substructures (connected subgraphs) using a depth-first approach similar to that used by dEclat [35]. To reduce the number of subgraph isomorphism operations, it keeps the embeddings of previously discovered subgraphs and tries to extend the embeddings by one edge. However, despite these optimizations, the reported speed of the algorithm is slower than that achieved by FSG and gSpan. This is mainly due to the fact that their candidate subgraph generation scheme does not ensure that the same subgraph is generated only once, and the algorithm generates and counts the frequency of the same subgraph multiple times. gSpan [32] finds the frequently occurring subgraphs also following a depth-first approach. Unlike the algorithm by Borgelt and Berthold, every time a candidate subgraph is generated, its canonical label is computed. If the computed label is the minimum one, the candidate is saved for further exploration of the depth search. If not, the candidate is discarded because there must be another path to the same candidate. By doing so, gSpan avoids redundant candidate generation. To ensure that these subgraph comparisons are done efficiently, they use a canonical labeling scheme based on depth-first traversals. In addition, gSpan does not keep the information about all previous embeddings of frequent subgraphs which saves the memory usage. However, all embeddings are identified on the fly, and use them to project the dataset in a fashion similar to that used by [1]. According to the reported performance in [32], gSpan and FSG are comparable on a chemical compound dataset used in the Predictive Toxicology Evaluation Challenge (PTE) [30], whereas gSpan performs better than FSG on synthetic datasets. FFSM [13] incorporates the join-based candidate generation scheme used by the horizontal algorithms into the vertical frequent subgraph mining paradigm proposed by gSpan. By the combination of the candidate generation and extension, FFSM is able to prune unnecessary candidates aggressively. It is reported that the speed of FFSM outperforms gSpan by up to a factor of seven with chemical compound datasets.

7 Conclusions

In this paper we presented a heuristic algorithm called GREW to find frequent connected subgraphs from a single undirected input graph, and evaluated its efficiency and scalability by various experiments using graphs directly created from the four real datasets. Our results showed that GREW is highly scalable, can operate on very large graphs, and find a large and diverse set of patterns.

GREW's low computational requirements allowed us to develop effective randomization strategies that were designed to find a large set of patterns that can cover either the lattice of frequent subgraphs or the input graph. This suggests that GREW can be used as an effective building block in developing task-specific pattern discovery algorithms with applications to clustering and classification.

Acknowledgment

We thank Dr. Hiroshi Motoda, Dr. Takashi Washio, Dr. Tetsuya Yoshida and Dr. Takashi Matsuda at the Institute of Scientific and Industrial Research, Osaka University for their useful comments and discussions, and for kindly providing the implementation of B-GBI.

References

[1] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

[2] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[3] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[4] D. J. Cook, L. B. Holder, and S. Djoko. Knowledge discovery from structural data. *Journal of Intelligent Information Systems*, 5(3):229–245, 1995.

[5] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of the 4th ACM SIGKDD International Conference on Knowledge*

Discovery and Data Mining (KDD-98), pages 30–36. AAAI Press, 1998.

[6] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure based approaches for classifying chemical compounds. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM)*, 2003.

[7] S. Fortin. The graph isomorphism problem. Technical Report TR96-20, Department of Computing Science, University of Alberta, 1996.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[9] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In *Proc. of the 5th International Conference on Discovery Science*, 2002.

[10] M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, 1997.

[11] H. Hofer, C. Borgelt, and M. R. Berthold. Large scale mining of molecular fragments with wildcards. In *Proc. of the 5th International Symposium on Intelligent Data Analysis (IDA 2003)*, volume 2810 of *Lecture Notes in Computer Science*, pages 376–385, 2003.

[12] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the SUBDUE system. In *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, pages 169–180, 1994.

[13] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomophism. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM'03)*, 2003.

[14] J. Huan, W. Wang, A. Washington, J. Prins, R. Shah, and A. Tropsha. Accurate classification of protein structural families using coherent subgraph analysis. In *Proceedings of the 9th Pacific Symposium on Biocomputing (PSB)*, 2004.

[15] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, pages 13–23, Lyon, France, September 2000.

[16] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, March 2003.

[17] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.

[18] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in HIV data. In *Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-01)*, pages 136–143, 2001.

[19] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*. in press.

[20] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, November 2001.

[21] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. Technical Report 02-026, University of Minnesota, Department of Computer Science, 2002.

[22] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *Proc. of the 2004 SIAM International Conference on Data Mining (SDM04)*, 2004.

[23] T. Matsuda, H. Motoda, T. Yoshida, and T. Washio. Mining patterns from structured data by beam-wise graph-based induction. In *Proc. of the 5th International Conference on Discovery ScienceDiscoveery (DS 2002)*, volume 2534 of *Lecture Notes in Computer Science*, pages 422–429. Springer-Verlag, 2002.

[24] B. D. McKay. Nauty users guide. http://cs.anu.edu.au/~bdm/nauty/.

[25] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[26] T. M. Mitchell. *Machine learning*. McGraw Hill, New York, 1996.

[27] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of International Conference on Database Theory (ICDT)*, pages 398–416, 1999.

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

[29] R. C. Read and D. G. Corneil. The graph isomorph disease. *Journal of Graph Theory*, 1:339–363, 1977.

[30] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1–6, 1997.

[31] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, pages 458–465, 2002.

[32] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

[33] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, 2003.

[34] K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, 1995.

[35] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute, 2001.

[36] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. Technical Report 99-10, Department of Computer Science, Rensselaer Polytechnic Institute, October 1999.