Mining Graph Patterns Efficiently via Randomized Summaries *

Chen Chen[1] Cindy X. Lin[1] Matt Fredrikson[2] Mihai Christodorescu[3] Xifeng Yan[4] Jiawei Han[1]

[1]University of Illinois at Urbana-Champaign
{cchen37, xidelin2, hanj}@cs.uiuc.edu
[2]University of Wisconsin at Madison
mfredrik@cs.wisc.edu
[3]IBM T. J. Watson Research Center
mihai@us.ibm.com
[4]University of California at Santa Barbara
xyan@cs.ucsb.edu

ABSTRACT

Graphs are prevalent in many domains such as Bioinformatics, social networks, Web and cyber-security. Graph pattern mining has become an important tool in the management and analysis of complexly structured data, where example applications include indexing, clustering and classification. Existing graph mining algorithms have achieved great success by exploiting various properties in the *pattern space*. Unfortunately, due to the fundamental role subgraph isomorphism plays in these methods, they may all enter into a pitfall when the cost to enumerate a huge set of isomorphic embeddings blows up, especially in large graphs.

The solution we propose for this problem resorts to reduction on the *data space*. For each graph, we build a *summary* of it and mine this shrunk graph instead. Compared to other data reduction techniques that either reduce the number of transactions or compress between transactions, this new framework, called SUMMARIZE-MINE, suggests a third path by *compressing within transactions*. SUMMARIZE-MINE is effective in cutting down the size of graphs, thus decreasing the embedding enumeration cost. However, compression might lose patterns at the same time. We address this issue by generating *randomized* summaries and repeating the process for multiple rounds, where the main idea is that true patterns are unlikely to miss from all rounds. We provide strict probabilistic guarantees on pattern loss likelihood. Experiments on real malware trace data show that SUMMARIZE-MINE is very efficient, which can find interesting malware fingerprints that were not revealed previously.

*The work was supported in part by the U.S. National Science Foundation grants IIS-08-42769, IIS-08-47925 and BDI-05-15813, NASA grant NNX08AC35A, and the Air Force Office of Scientific Research MURI award FA9550-08-1-0265.

VLDB '09, August 24-28, 2009, Lyon, France

1. INTRODUCTION

Recent years have witnessed the prevalence of graph data in many scientific and commercial applications, such as Bioinformatics, social networks, Web and cyber-security, partly because graphs are able to model the most complex data structures. As illustrated by the enhancement made to many core tasks of these domains, *e.g.*, indexing [29] and classification [14, 6], mining graph patterns that frequently occur (for at least *min_sup* times) can help people get insight into the structures of data, which is well beyond traditional exercises of frequent patterns, such as association rules [1].

However, the emergence of bulky graph datasets places new challenges for graph data mining. For these scenarios, the target graphs are often too large which may severely restrict the applicability of current pattern mining technologies. For example, one emerging application of frequent graph patterns is to analyze the behavior graphs of malicious programs. One can instrument malicious binaries to generate system call graphs, where each node is a system call event. By comparing the subtle differences between graphs generated by malware and benign programs, it is possible to find those graph fingerprints that are common and unique in malicious programs [5]. Unfortunately, due to the bulkiness and complexity of system call graphs, we found that none of the state-of-art mining algorithms can serve this new and critical task well. Similar problems are also encountered for biological networks and social networks.

Existing frequent subgraph mining algorithms, like those developed in [15, 28, 13], achieved great success using strategies that efficiently traverse the *pattern space*; during this process, frequent patterns are discovered after checking a series of subgraph isomorphisms against the database. However, as we argue in this paper, these methods ignore the important fact that isomorphism tests are sometimes expensive to perform. The key issue here is a huge set of isomorphic embeddings that may exist. In order to check the occurrences of a pattern in a large graph, one often needs to enumerate exponentially many subgraphs. This situation is further worsened by the possible overlaps among subgraphs. Looking at $G_1, G_2, \ldots$ in Figure 1, subgraphs such as triangles might share a substantial portion in common, while only one different node/edge would require them to be examined twice, which quickly blows up the total cardinality.

We use a simple example to demonstrate the above scenario. Suppose we have 1,000,000 length-2 paths in a large graph and we would like to check if it has a triangle inside. These one million paths have to be checked one by one because each of them has the potential to grow into a full embedding of the triangle pattern. The same dilemma exists for any pattern that includes a length-2 path. Such a huge number of possible embeddings become a severe bottleneck for graph pattern mining tasks.

Now let us consider possible ways to reduce the number of embeddings. In particular, since many embeddings overlap substantially, we explore the possibility of somehow "merging" these embeddings so that the overall cardinality is significantly reduced. As Figure 1 depicts, merging of embeddings is achieved by binding vertices with identical labels into a single node and collapsing the network correspondingly into a smaller version. As suggested by previous studies [25, 3], the above process indeed provides a *graph summary* that generalizes our view on the data to a higher level, which can facilitate analysis and understanding, similar to what OLAP (On-Line Analytical Processing) does for relational databases.

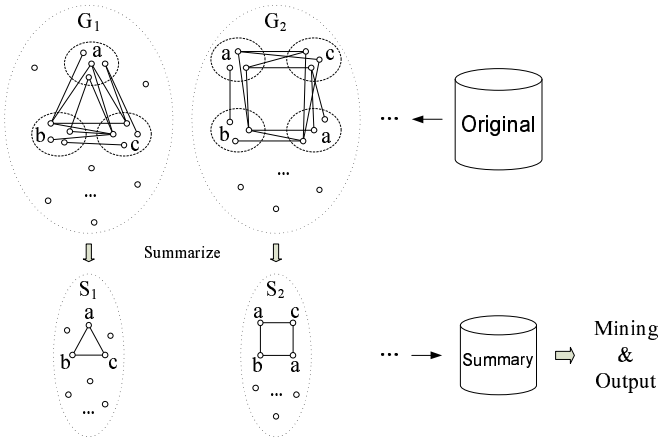

Figure 1: The Summarize-Mine Framework

Graph summarization leads to a dramatic cut-down of graph size as well as the total number of embeddings, which makes subgraph isomorphism cheaper to perform. This forms the main idea of our SUMMARIZE-MINE framework: In Figure 1, we first summarize the original graphs $\{G_1, G_2, \ldots\}$ into small summaries $\{S_1, S_2, \ldots\}$, which are then mined for frequent patterns, where state-of-art algorithms should now perform well. However, the price paid here is the possible loss of patterns, *i.e.*, there could exist false positives and false negatives. For false positives, one can always verify their frequency against the original database and discard those failing ones (interestingly, as we shall discuss later, based on the relationship between G_i and S_i, a lot of verification efforts can be transferred to the small-sized S_i, as well); for false negatives, we choose to generate summaries in a *randomized* manner and repeat the process for multiple rounds. Intuitively, true patterns are unlikely to miss from all rounds.

Recapitulating the above discussions, we outline the contributions made in this paper as follows.

First, a previously neglected issue in frequent graph pattern mining, *i.e.*, the intrinsic difficulty to perform embedding enumeration in large graphs, is examined, which could easily block many downstream applications. Compared to previous studies that focus on the efficient traversal of *pattern space*, the perspective of this work is *data space* oriented, which leads to an innovative SUMMARIZE-MINE framework. The power and efficiency of our algorithm is validated by extensive experiments on real program analysis data, which can find interesting malware fingerprints that were not revealed previously.

Second, the data reduction principle we adopt is to *compress information within transactions*. It eliminates the shortcoming of lossy summarization by a *randomizing* technique, which repeats the whole process for multiple rounds and achieves strict probabilistic guarantees. This is novel compared to other methods that either reduce the number of transactions (*e.g.*, sampling [26]) or compress between transactions (*e.g.*, FP-Growth [8] losslessly compresses the whole dataset into an FP-tree for frequent itemset mining).

Third, our proposed method of reducing data within transactions supplemented by randomized mechanisms marks an additional dimension that is orthogonal to the state-of-art pattern mining technologies. In this sense, one can freely combine SUMMARIZE-MINE with other optimizations suggested in the past to further enhance their performance, and the idea is not restricted to graphs, which can also be extended to sequences, trees, *etc.*.

Finally, nowadays, extremely huge networks such as those of Internet cyber-attacks and on-line social network websites (*e.g.*, Facebook and MySpace) are not uncommon; sometimes, they even cannot fit in main memory, which makes it very hard for people to access and analyze. To this extent, the usage of SUMMARIZE-MINE can be viewed from another perspective: Considering the increasingly important role of summarization as a necessary preprocessing step, we have made a successful initial attempt to analyze how this procedure would impact the underlying patterns (frequent substructures being a special instance). It is crucial for the applications to understand when and to what degree a compressed view can represent the original data in terms of its patterns.

The rest of this paper is organized as follows. Preliminaries and the overall SUMMARIZE-MINE framework are outlined in Sections 2 and 3. The major technical investigations, including probabilistic analysis of false negatives, verification of false positives and iterating multiple times to ensure result completeness, are given in Sections 4, 5 and 6, respectively. Section 7 presents experimental results, Section 8 discusses related work, and Section 9 concludes this study.

2. PRELIMINARIES

In this paper, we will use the following notations. For a graph g, $V(g)$ is its vertex set, $E(g) \subseteq V(g) \times V(g)$ is its edge set, and l is a label function mapping a vertex or an edge to a label.

DEFINITION 1. *(Subgraph Isomorphism). For two labeled graphs g and g', a* subgraph isomorphism *is an* injective *function $f : V(g) \to V(g')$, such that 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. Under these conditions, f is called an* embedding *of g in g', and g is called a* subgraph *of g', denoted as $g \subseteq g'$.*

DEFINITION 2. *(Frequent Subgraph). Given a graph database* $D = \{G_1, G_2, \ldots, G_n\}$ *and a graph pattern* p*, let* D_p *be the set of graphs in* D *where* p *appears as a subgraph. We define the* support *of* p *as* $sup(p) = |D_p|$*, whereas* D_p *is referred as* p*'s* supporting graphs *or* p*'s* projected database*. With a predefined threshold* min_sup*,* p *is said to be* frequent *if* $sup(p) \geq min\_sup$*.*

The problem studied in this paper is essentially the same as that of a well-studied graph mining task: finding all frequent subgraphs in a database D, except that the graphs in D are now associated with large size. As we mentioned in the introduction, our proposal is to perform summarization at first.

DEFINITION 3. *(Summarized Graph). Given a labeled graph* G*, suppose its vertices* $V(G)$ *are partitioned into groups, i.e.,* $V(G) = V_1(G) \cup V_2(G) \cup \cdots \cup V_k(G)$*, such that 1)* $V_i(G) \cap V_j(G) = \phi$ $(1 \leq i \neq j \leq k)$*, 2) all vertices in* $V_i(G)$ $(1 \leq i \leq k)$ *have the same labels. Now, we can* summarize G *into a compressed version* S*, written as* $S \prec G$*, where 1)* S *has exactly* k *nodes* $v_1, v_2, \ldots, v_k$ *that correspond to each of the groups (i.e.,* $V_i(G) \mapsto v_i$*), while the label of* v_i *is set to be the same as those vertices in* $V_i(G)$*, and 2) an edge* (v_i, v_j) *with label* l *exists in* S *if and only if there is an edge* $(u, \hat{u})$ *with label* l *between some vertex* $u \in V_i(G)$ *and some other vertex* $\hat{u} \in V_j(G)$*.*

Based on the above definition, *multi-edge* becomes possible for a summarized graph, *i.e.*, there might be more than one labeled edge that exist between two vertices $v_i, v_j \in V(S)$, if there is an edge $(u_1, \hat{u_1})$ with label l_1 and another edge $(u_2, \hat{u_2})$ with label $l_2 \neq l_1$ such that u_1, u_2 is in the node group $V_i(G)$ and $\hat{u_1}, \hat{u_2}$ is in the node group $V_j(G)$. To find patterns on top of such summaries, slight modifications are needed because traditional graph mining algorithms in general assume *simple graphs* (*i.e.*, no self-loops and multi-edges). We shall get back to this issue later as the discussion proceeds.

3. THE SUMMARIZE-MINE FRAMEWORK

Given a graph database $D = \{G_1, G_2, \ldots, G_n\}$, if we summarize each $G_i \in D$ to $S_i \prec G_i$, then a summarized database $D' = \{S_1, S_2, \ldots, S_n\}$ is generated. Denote the collection of frequent subgraphs corresponding to D and D' as $FP(D)$ and $FP(D')$, respectively. In this section, we are going to examine the relationship between these two pattern sets and investigate the possibility to shift mining from D to D'.

Intuitively, we expect that $FP(D)$ and $FP(D')$ are similar to each other if the summarization from D to D' is properly conducted. As for the portion that is different between them, there are two cases.

DEFINITION 4. *(False Negatives). A subgraph* p *frequent in* D *but not frequent in* D'*, i.e.,* $p \in FP(D)$ *and* $p \notin FP(D')$*, is called a* false negative *caused by summarization.*

DEFINITION 5. *(False Positives). A subgraph not frequent in* D *but frequent in* D'*, i.e.,* $p \notin FP(D)$ *and* $p \in FP(D')$*, is called a* false positive *caused by summarization.*

For the rest of this section, we are going to discuss how these two types of errors can be remedied, which finally gives rise to a novel SUMMARIZE-MINE framework.

3.1 Recovering False Negatives

False negatives include those patterns that are missed out after we summarize the graphs. In Figure 2, we explain the reason behind. Suppose p is a graph pattern such that $p \subseteq G_i$, and correspondingly f is an embedding of p in G_i. Consider the summary $S_i \prec G_i$, f will disappear if there exist two nodes $u, \hat{u} \in V(p)$ whose images in $V(G_i)$, *i.e.*, $f(u)$ and $f(\hat{u})$, are merged together as we shrink G_i into S_i. This will cause the support of p to decrease upon summarization.

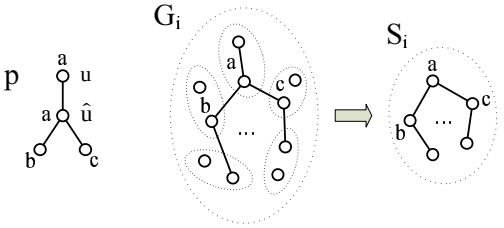

Figure 2: The Cause of False Negatives

So, how should we avoid false negatives? To begin with, it is easy to prove the following lemma.

LEMMA 1. *For a pattern* p*, if each of its vertices bears a different label, then* p*'s supporting graph set in the summarized database* D' *is no smaller than that in the original database* D*, i.e.,* $D'_p \supseteq D_p$*.*

PROOF. Suppose $G_i \in D_p$, *i.e.*, $p \subseteq G_i$, let f be an embedding of p in G_i. Obviously, for p's vertices $u_1, \ldots, u_m$, their corresponding images $f(u_1), \ldots, f(u_m)$ in G_i must have different labels, and thus $f(u_1), \ldots, f(u_m)$ should belong to m separate groups, which end up as distinct nodes $v_1, \ldots, v_m$ in the summarized graph $S_i \prec G_i$. Define another injective function $f' : V(p) \rightarrow V(S_i)$ by mapping u_j to v_j $(1 \leq j \leq m)$. Based on Definition 3, it is easy to verify that whenever there is an edge $(u_{j_1}, u_{j_2}) \in E(p)$ with label l, there exists a corresponding edge $(v_{j_1}, v_{j_2}) \in E(S_i)$ bearing the same label. Now, f' represents a qualified embedding of p in S_i. More generally, $p \subseteq S_i$ will hold for each G_i's shrunk version S_i if $G_i \in D_p$, indicating that D'_p is at least as large as D_p. ∎

Based on Lemma 1, false negatives can only happen for those patterns with at least two identically labeled vertices. Meanwhile, from the proof above, we conclude that even if two vertices $u_1, u_2 \in V(p)$ possess the same label, as long as their images $f(u_1), f(u_2)$ are not merged by summarization, the embedding f is still preserved. According to these observations, we could partition nodes into identically labeled groups on a *random* basis, where for those vertices with same labels in pattern p, they have a substantial probability $q(p)$ to stay in different groups, which guarantees that no embeddings will be destroyed. Facing such probabilistic pattern loss, we decide to deliberately lower the support threshold in the summarized database by a small margin to $min\_sup' < min\_sup$: As we shall prove in Section 4, this will then insure a high probability P for patterns to remain frequent in D'. Finally, to further reduce the false negative rate, we can perform randomized summarization for multiple times in an independent fashion, because the overall pattern missing probability $(1 - P)^t$ will quickly converge

to 0 as the number of iterations t increases. The details of false negative analysis are given in Section 4.

3.2 Discarding False Positives

Given a graph pattern p, there is also possibility for its support to increase upon summarization. Figure 3 shows a "faked" embedding of p formed in the summarized graph S_i, where two sets of edges originally adjacent to different vertices with label a are now attached to the same node.

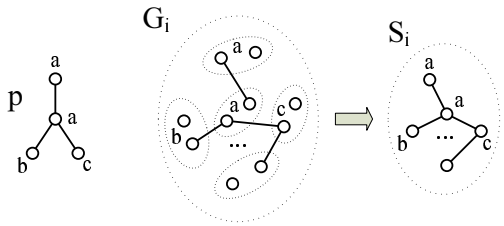

Figure 3: The Cause of False Positives

It is much easier to deal with false positives. For false negatives, we must provide a mechanism to recover true patterns that have disappeared after summarization, while for false positives, we only need an efficient *verification* scheme to check the result set and get rid of those entries that are actually infrequent.

A straightforward verification scheme computes the support of every $p \in FP(D')$ in the original database D: If $sup(p)$ is smaller than min_sup, we discard p from the output. Interestingly, there is a better way to verify patterns by leveraging the summaries: The embedding of p in the summarized graph actually reveals its possible locations in the original graph, which can be used to speed up the process. Technical details will be covered in Section 5.

3.3 The Overall Algorithm Layout

With randomization and verification, the SUMMARIZE-MINE framework is outlined as follows.

1. **Summarization**: For each G_i in a graph database D, randomly partition its vertex set $V(G_i)$: For vertices with label $l_j (1 \le j \le L)$, where L represents the total number of labels, we assign x_j groups. This will result in x_j nodes with label l_j in the corresponding summary graph. As the application characteristics vary, we can control the summarization process by changing $x_1, \ldots, x_L$ to best cope with the situation.

2. **Mining**: Apply any state-of-art frequent subgraph mining algorithm on the summarized database $D' = \{S_1, S_2, \ldots, S_n\}$ with a slightly lowered support threshold min_sup', which generates the pattern set $FP(D')$.

3. **Verification**: Check patterns in $FP(D')$ against the original database D, remove those $p \in FP(D')$ whose support in D is less than min_sup and transform the result collection into R'.

4. **Iteration**: Repeat steps 1-3 for t times. To guarantee that the overall probability of missing any frequent pattern is bounded up by ϵ, we set the number of rounds t as $\lceil \frac{\log \epsilon}{\log(1-P)} \rceil$, where $1-P$ is the false negative rate in one round.

5. **Result Combination**: Let $R'_1, R'_2, \ldots, R'_t$ be the patterns obtained from different iterations, the final result is $R' = R'_1 \cup R'_2 \cup \cdots \cup R'_t$.

Compared to the true pattern set R that would be mined from the original database D if there are enough computing resources, no false positives exist, *i.e.*, $R' \subseteq R$, and the probability for a pattern $p \in R$ to miss from R' is at most ϵ. Note that, the verification step here is put after the mining step for clarity purposes. As we shall see later, these two steps can also be interleaved, where verifications are performed on-the-fly: Whenever a pattern p is discovered, SUMMARIZE-MINE verifies it immediately if p has not been discovered and verified by previous iterations.

In the following, we will start from probabilistic analysis of false negatives in Section 4, followed by Section 5, which focuses on the verification of false positives, and Section 6, which discusses iterative SUMMARIZE-MINE as well as result combination.

4. BOUNDING THE FALSE NEGATIVE RATE

As we proved in Lemma 1 of Section 3.1, for a pattern p and a graph G_i in the original database, if p is a subgraph of G_i through embedding f, then as G_i is summarized into S_i, f disappears from S_i if $f(u_1), \ldots, f(u_m)$ are disseminated into less than m groups and thus correspond to less than m vertices in S_i. Suppose there are m_j and x_j vertices with label l_j in p and S_i, respectively, we have the following lemma.

LEMMA 2. *For a graph pattern p, if $p \subseteq G_i$, then p is also a subgraph of $S_i \prec G_i$ with probability at least*

$$\frac{P_{x_1}^{m_1} \cdots P_{x_L}^{m_L}}{x_1^{m_1} \cdots x_L^{m_L}},$$

given that the grouping and merging of nodes that transform G_i into S_i is performed on a completely random basis. Here, $P_{x_j}^{m_j}$ represents the number of permutations, which is equal to $\binom{x_j}{m_j} m_j!$.

PROOF. Consider an embedding f through which p is a subgraph of G_i. The probability that all m_j vertices with label l_j are assigned to m_j different groups (and thus f continues to exist) is:

$$\frac{x_j}{x_j} \frac{x_j - 1}{x_j} \cdots \frac{x_j - m_j + 1}{x_j} = \frac{P_{x_j}^{m_j}}{x_j^{m_j}}.$$

Multiplying the probabilities for all L labels (because the events are independent), we have:

$$\text{Prob}\big[p \subseteq S_i\big] \ge \frac{P_{x_1}^{m_1} \cdots P_{x_L}^{m_L}}{x_1^{m_1} \cdots x_L^{m_L}}.$$

Here, x_j must be at least as large as m_j to make the product of probabilities meaningful, and during implementation, there is often no problem for us to make $x_j >> m_j$ so that vertices with identical labels will not collide with high probability. ∎

To simplify analysis, if we stick with a particular set of x_j's $(1 \le j \le L)$ when summarizing different graphs in the database, the probability bound in Lemma 2 can be written as $q(p)$, since its value only depends on the label distribution of pattern p, which holds for any $G_i \in D_p$. Now, because

of those embeddings that disappear due to summarization, it is well expected that the pattern support will experience some drop, with Theorem 1 characterizing the probability of seeing a particular dropping magnitude.

THEOREM 1. *Suppose a pattern p's support in the original database is s, i.e., $|D_p| = s$, for any $s' \leq s$, the probability that p's support in D' falls below s' upon summarization can be bounded as follows:*

$$\text{Prob}\big[|D'_p| \leq s'\big] \leq \sum_{T=0}^{s'} \binom{s}{T} q(p)^T [1-q(p)]^{s-T}.$$

PROOF. For each $G_i \in D_p$, we focus on a particular subgraph embedding f_i and define an indicator variable I_i such that $I_i = 1$ if f_i continues to exist in S_i and $I_i = 0$ otherwise. Then,

$$\text{Prob}\big[|D'_p| > s'\big] \geq \text{Prob}\Big[\sum_{G_i \in D_p} I_i > s'\Big],$$

because whenever $\sum_{G_i \in D_p} I_i > s'$, there must be more than s' subgraph embeddings that are preserved in the summarized database and thus $|D'_p| > s'$. We have:

$$\begin{aligned} &\text{Prob}\big[|D'_p| \leq s'\big] \leq \text{Prob}\Big[\sum_{G_i \in D_p} I_i \leq s'\Big] \\ = &\sum_{T=0}^{s'} \text{Prob}\Big[\sum_{G_i \in D_p} I_i = T\Big]. \end{aligned}$$

The difference between the left and right hand side probabilities is due to three effects: (1) there could be multiple embeddings of p in G_i, so that p's support after summarization may not decrease even if one embedding disappears, (2) an "faked" embedding like that depicted in Figure 3 might emerge to keep p as a subgraph of S_i, and (3) "faked" embeddings can also happen for a graph G_j which originally does not contain p. Now, because events are independent,

$$\text{Prob}\big[\sum_{G_i \in D_p} I_i = T\big] = \binom{s}{T} q(p)^T [1-q(p)]^{s-T},$$

where $q(p) = \text{Prob}\big[I_i = 1\big]$ for all i such that $G_i \in D_p$. Finally,

$$\begin{aligned} &\text{Prob}\big[|D'_p| \leq s'\big] \leq \sum_{T=0}^{s'} \text{Prob}\Big[\sum_{G_i \in D_p} I_i = T\Big] \\ = &\sum_{T=0}^{s'} \binom{s}{T} q(p)^T [1-q(p)]^{s-T} \end{aligned}$$

is proved. ∎

COROLLARY 1. *Assume that the support threshold is min_sup, we set a new threshold $min\_sup' < min\_sup$ for the database D' summarized from D and mine frequent subgraphs on D'. The probability for a pattern p to be a false negative, i.e., p is frequent in D but not frequent in D', is at most*

$$\sum_{T=0}^{min\_sup'-1} \binom{min\_sup}{T} q(p)^T [1-q(p)]^{min\_sup-T}.$$

PROOF. Being a false negative, we have $s = |D_p| \geq min\_sup$ and $|D'_p| \leq min\_sup' - 1$. Let $s' = min\_sup' - 1$, a direct application of Theorem 1 leads to

$$\text{Prob}\big[|D'_p| < min\_sup'\big] \leq \sum_{T=0}^{min\_sup'-1} \binom{s}{T} q(p)^T [1-q(p)]^{s-T},$$

where the right hand side corresponds to a binomial random variable $B\big(s, q(p)\big)$'s cumulative distribution function (CDF) being evaluated at s'. Denote the CDF of a binomial variable $Y \sim B(N,p)$ as $F_B(N,p;n) = \text{Prob}\big[Y \leq n\big]$, we have $F_B(N,p;n)$ monotonically decreasing in N, because Y is the sum of N independent Bernoulli random variables $X_1, \ldots, X_N \sim Ber(p)$: When more X_i's get involved, it is naturally harder to have their sum Y bounded up by some fixed number n. This leads to $F_B(N_1,p;n) \geq F_B(N_2,p;n)$ if $N_1 \leq N_2$. Finally, since $s \geq min\_sup$,

$$\begin{aligned} &F_B\big(s, q(p); s'\big) \leq F_B\big(min\_sup, q(p); s'\big) \\ = &\sum_{T=0}^{min\_sup'-1} \binom{min\_sup}{T} q(p)^T [1-q(p)]^{min\_sup-T}, \end{aligned}$$

which is combined with the inequality at the beginning to complete the proof. ∎

As the SUMMARIZE-MINE framework suggests, the false negative rate after t iterations is $(1-P)^t$. To make $(1-P)^t$ less than some small ϵ, one can either increase the number of rounds t, or decrease the one-round false negative rate $1-P$, which is achieved by lowering the support threshold min_sup' on the summarized database D'. Since increasing t and reducing min_sup' will both lead to a longer mining time, we could simultaneously control both parameters to find an optimal trade-off point where the best efficiency is achieved. This will be tested in the experiments.

5. VERIFYING FALSE POSITIVES

To implement SUMMARIZE-MINE, we take gSpan [28] as the skeleton of our mining algorithm. The main idea of gSpan is as follows: Each labeled graph pattern can be transformed into a sequential representation called *DFS code*, based on a depth-first traversal of the pattern. With a defined *lexicographical order* on the DFS code space, all subgraph patterns can be organized into a tree structure, where (1) patterns with k edges are put on the kth level, and (2) a preorder traversal of this tree would generate the DFS codes of all possible patterns in the lexicographical order. Figure 4 shows a pattern tree, where $v_1, v_2, \ldots, v_n$ are vertex patterns, p_1 is a pattern with one edge, and p_1 is a subgraph of p_2.

This DFS code-based pattern tree is used in SUMMARIZE-MINE. For each graph pattern p, we conduct the following steps.

1. We decide whether the DFS code of p is minimum according to the defined lexicographical order. Here, patterns might have different codes in the tree because of graph isomorphisms but we only need to examine one of them. In this sense, non-minimum DFS codes can be discarded since the corresponding minimum ones must have been visited by the preorder traversal.
2. We check p against the summarized graphs and get p's projected database D'_p. If $|D'_p|$ falls below min_sup', we abort the search along this branch.

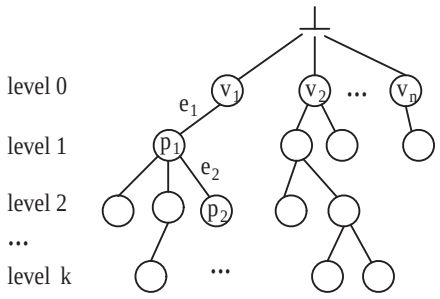

Figure 4: A Pattern Tree

3. For each summary $S_i \in D'_p$, we enumerate all embeddings of p in S_i and based on that determine every possible one-edge extension that can be added to them. These candidate patterns are inserted into the pattern tree, which will be explored later.

4. When we drive search into a particular branch, D'_p is passed down as a transaction ID-list, which will help pruning since the new projected database there can only be a subset of D'_p.

If SUMMARIZE-MINE generates a pattern tree as shown in Figure 4, we could start checking false positives from big patterns so that the verification of many smaller patterns can be avoided. Given two patterns p_1 and p_2, where p_1 is a subgraph of p_2, there is no need to verify p_1 if p_2 already passes the *min_sup* threshold, because $sup(p_1) \geq sup(p_2) \geq min\_sup$. Referring to the DFS code tree, this is done by visiting the data structure in a *bottom-up* manner, which can be easily implemented through a postorder traversal. On the other hand, it seems that adopting the opposite direction, *i.e.*, visiting the tree in a *top-down* manner through a preorder traversal, might also give us some advantages: If we verify p_1 before p_2, then there is no need to further try p_2 if p_1 already fails the test, since $min\_sup > sup(p_1) \geq sup(p_2)$. In this sense, considering the question of picking a better one from these two approaches, it really depends on how many false positives exist in the set of patterns we want to verify, which could be data-specific. Generally speaking, if there are not/too many false positives, the bottom-up/top-down approach should work well.

Summary-Guided Isomorphism Checking. During the verification process, after getting D'_p, we want to check a pattern p against each $G_i \in D$ and get its support in the original database. Suppose $G_i \succ S_i$, there are two cases: $S_i \in D'_p$ and $S_i \notin D'_p$. For the first case, the embedding of p in S_i could help us quickly find its possible embeddings in G_i. Let $f : V(p) \rightarrow V(S_i)$ be the embedding of p in S_i, where the images of p's vertices under f are $f(u_1), \ldots, f(u_m)$. Recall that G_i is summarized into S_i by merging a node group of G_i into a single vertex of S_i, we can check whether there exists a corresponding embedding of p in G_i by picking one node from each of the node groups that have been summarized into $f(u_1), \ldots, f(u_m)$, and examining their mutual connections. This should be more efficient than blindly looking for a subgraph isomorphism of p in G_i, without any clue about the possible locations. For the second case, there is no embedding of p in S_i to leverage, can we also confirm that $p \not\subseteq G_i$ by looking at the summary only? Let us choose a subgraph $p' \subseteq p$ such that each of p''s vertices bears a different label, and test p' against S_i; based on Lemma 1, since the embeddings of p' can never be missed upon summarization, if we can confirm that $p' \not\subseteq S_i$, then it must be true that $p' \not\subseteq G_i$ and there is no hope for p, a supergraph of p', to exist in G_i, either. Concerning implementation, we can always make p' as big as possible to increase the pruning power. Finally, we have transformed isomorphism tests against the original large graph G_i to its small summary S_i, thus taking advantage of data reduction.

6. ITERATIVE SUMMARIZE-MINE

In this section, we combine the summarization, mining, and verification procedures together and put them into an iterative framework. As discussed previously, adding more iterations can surely reduce the probability of false negatives; however, it introduces some problems, as well. For example, the final step of SUMMARIZE-MINE is to merge all R'_k's $(k = 1, 2, \ldots, t)$ into a combined set R', which requires us to identify what the individual mining results have in common so that only one copy is retained. Furthermore, due to the overlap among $R'_1, R'_2, \cdots$, a large number of patterns might be repeatedly discovered and verified.

One solution to this problem is to represent the patterns in each R'_k by their DFS codes, which are then sorted in lexicographical order, facilitating access and comparison. However, this approach is still costly. Our proposed strategy is as follows: Since $R'_1, R'_2, \ldots$ are mined from successive random summarizations of the original graph database D, it is expected that R'_k's would not be too different from each other because they are all closely related to $FP(D)$, the "correct" set of frequent subgraphs that would be mined from D. This hints us to unify the mining process of different iterations into a single data structure, *i.e.*, use only one pattern tree $\mathcal{T}$ to drive the mining ahead.

The advantages of doing so are two-fold. First, if a single data structure is maintained, and we incrementally modify $\mathcal{T}$ (*i.e.*, recover false negatives that have been wrongly omitted by previous rounds) as the mining of multiple iterations proceeds, then the problem of merging $R'_1, R'_2, \ldots$ is automatically solved, because there is only one pattern tree, which stores the combined result. Second, in such an integrated manner, intermediate calculations achieved in earlier rounds may help the processing of later rounds, which cannot be achieved if consecutive iterations are separated. The below example demonstrates this.

Our setting is as follows: In round 1, a pattern tree holding R'_1 is generated, which is drawn on the left hand side of Figure 5, *i.e.*, $\mathcal{T}_1$. Then, as we go into round 2, some patterns missed from the first iteration will be recovered (specifically, p_3 and p_4), which update $\mathcal{T}_1$ into a new tree $\mathcal{T}_2$ that is drawn on the right hand side.

Now, suppose we have finished round 1 and verified the patterns in R'_1, *e.g.*, p_1, p_2, by checking their support against D, the corresponding projected databases, *i.e.*, D_{p_1}, D_{p_2}, become known to us. These two support sets, represented as ID-lists, are stored with tree nodes p_1, p_2 for later use. Note that, since ID-lists only record integer identifiers of the transaction entries, they have moderate size and can be easily maintained/manipulated. The same strategy has been widely used in other graph data management tasks, *e.g.*, indexing [29].

Moving on to round 2, we start from the tree's root p_1

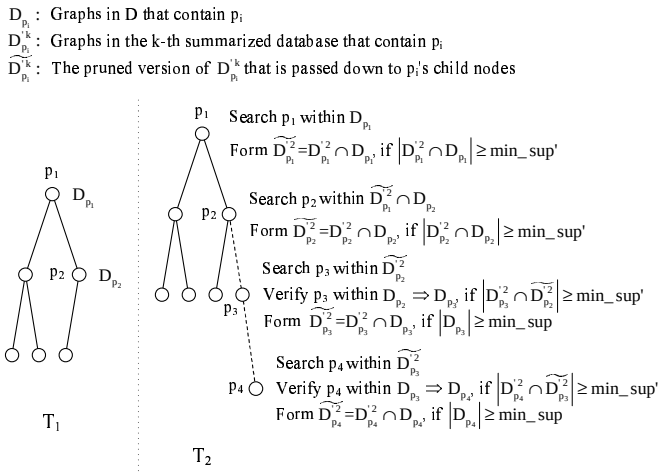

Figure 5: The First Two Iterations of Summarize-Mine *with* Verified ID-lists

(note that, although p_1 has been verified and shown to be frequent by round 1, we cannot bypass it in round 2, because patterns such as p_3 and p_4, which are infrequent in round 1 but frequent in round 2, have to be grown from it), where the first thing to do is checking p_1's support against the 2nd-round summarized database $D'^2 = \{S_1^2, S_2^2, \ldots, S_n^2\}$. Interestingly, it is only necessary to test p_1 with regard to those graphs in D_{p_1}, *i.e.*, what we finally obtain could be a subset of $D'^2_{p_1}$ that is confined within D_{p_1}, *i.e.*, $D'^2_{p_1} \cap D_{p_1}$. This turns out to be OK: For any pattern p_* that would be subsequently grown along the branch of p_1, $p_1 \subseteq p_* \Rightarrow D_{p_*} \subseteq D_{p_1}$ because of the Apriori principle; thus, when p_* is finally verified against the original database D, its support graphs D_{p_*} will be confined within D_{p_1}, anyway. This means that an early pruning by the ID-list of D_{p_1}, which is readily available after round 1, should not have any impact on the rest of the mining process.

We draw a few more steps in Figure 5 regarding the utilization of pre-verified ID-lists when it comes to patterns p_3, p_4, and the corollary below proves that the optimizations proposed would not change any theoretical foundation of Summarize-Mine.

Corollary 2. *The probability bound developed in Corollary 1 still holds if verified ID-lists are used to prune the mining space.*

Proof. Given a pattern p, the bound in Corollary 1 is directly related to whether p's embeddings for each of its support graphs in the original database D, *i.e.*, D_p, would diminish or not upon randomized summarizations. As shown above (think p as p_1 in Figure 5), when we utilize verified ID-lists for optimizations, entries in D_p are not filtered out for sure, which means that all deductions made in the corresponding proofs now continue to hold. ∎

The described pruning techniques should be applied as early as possible in order to boost performance. In this sense, we shall start verification right after a new pattern is discovered, so that its ID-list might be used even in the current iteration. Putting everything together, the pseudo-code of Summarize-Mine is given in Algorithm 1.

Algorithm 1 Summarize-Mine with Verified ID-lists

$D'^k = \{S_1^k, \ldots, S_n^k\}$: The kth-round summarized database.
p: The graph we are visiting on the pattern tree.
PD: Projected database passed from the caller.
$p.err$: A flag stored with p, it equals *true* if p has already failed to pass a verification test in previous iterations.
$p.IDs$: The ID-list stored with p, it equals ϕ if p is discovered for the first time and thus has not been verified.

```
sMine(p, PD) {
1:  if p.err == true then return;
2:  if p's DFS code is not minimum then return;
3:  if p.IDs ≠ φ then PD' = PD ∩ p.IDs;
4:  else PD' = PD;
5:  foreach graph ID i ∈ PD' do
6:    if p ⊈ S_i^k then PD' ← PD' − {i};
7:    else enumerate the embeddings of p in S_i^k; *
8:  if |PD'| < min_sup' then return;
9:  else if p.IDs == φ then
10:   verify p against the original database D;
11:   if |D_p| < min_sup then p.err = true; return;
12:   else
13:     store the IDs according to D_p in p.IDs;
14:     PD' = PD' ∩ p.IDs;
15: foreach p' ∈ pGrow do
16:   if p' is not a child of p in T then insert p' under p;
17:   sMine(p', PD');
}
```

*Steps 4-6 enumerate all possible one-edge extensions that can be made to p, which we denote as $pGrow$.

To start Algorithm 1, we call sMine(p_1, D), where p_1 is the root of the pattern tree and D includes every graph in the database. In order to grow all possible patterns, p_1 should be a null graph with zero vertices, as Figure 4 depicts. In line 1, we return immediately if the same p has been shown to be infrequent by previous iterations; and the reason for setting such a flag is to guarantee that unsuccessful verifications will not be performed repeatedly. Line 2 checks whether a given DFS code is minimum. Lines 3-4 conduct a pre-pruning if p has been verified in the past and thus an ID-list is readily available. Lines 5-7 proceed like normal frequent subgraph mining algorithms, where support is computed by checking isomorphic embeddings against the current projected database, and all possible one-edge extensions of p are recorded during this process. If the support does not pass the lowered min_sup' threshold on summarized databases, the algorithm returns immediately (line 8); otherwise, we verify p (line 10) if it has not been verified so far (line 9), mark $p.err$ as *true* and return if p cannot pass the min_sup threshold after verification (line 11). If p indeed can pass min_sup (line 12), we store the ID-list (line 13) and use it to immediately prune the projected database (line 14) that will be passed on when sMine is called for those patterns grown from p (lines 15-17).

We discuss some variations of Algorithm 1 in the following. First, we have been verifying patterns in the same order as they are mined out, which corresponds to a top-down scheme. As we mentioned in Section 5: The verified ID-lists of each node in the pattern tree can also be obtained in a

bottom-up manner, while the only shortcoming here is that, pruning must be delayed until the next iteration, because bottom-up checking can only happen when the whole pattern tree is ready. Second, there are costs, as well as benefits, to calculate and store the exact IDs of every pattern's support graphs in D. Interestingly, suppose we choose bottom-up verification, then for a pattern tree $\mathcal{T}$, we could have only verified those leaf nodes, while all internal nodes are guaranteed to be frequent (because they have even higher support), without any calculations. Thus, it is not always necessary to maintain the ID-lists.

Consider whether or not to compute verified ID-lists, plus that both top-down and bottom-up verification schemes can be selected, there are four cases in total:

- ID-list+top-down: It corresponds to Algorithm 1.
- ID-list+bottom-up: Here, though verification result can only be used in the next iteration, we are not sure whether this shortcoming can be overcome by the relative edge if bottom-up verification is faster than its top-down counterpart. We will reexamine this issue in experiments.
- No ID-list+top-down: This scenario does not make much sense, because in top-down verification, the ID-lists of all patterns in the tree can be obtained as a by-product. So, why not take this "free lunch" to boost performance?
- No ID-list+bottom-up: Figure 6 illustrates the situation. We adopt bottom-up postorder traversal to verify false positives, while successive iterations are essentially independent to each other, except that they share the same tree $\mathcal{T}$ to hold the mining result.

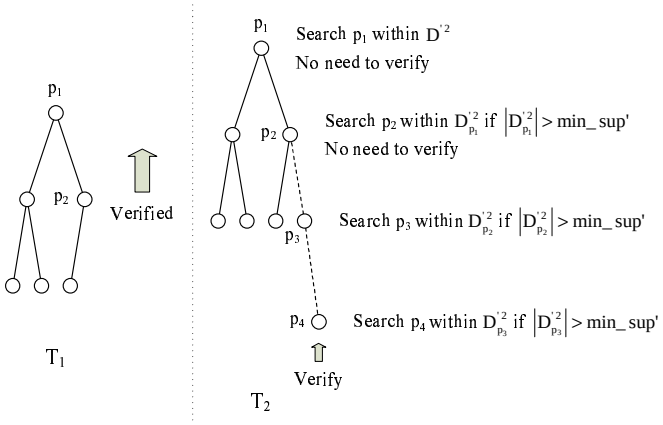

Figure 6: The First Two Iterations of Summarize-Mine *without* Verified ID-lists

7. EXPERIMENTAL RESULTS

In this section, we will provide empirical evaluations of SUMMARIZE-MINE. We have two kinds of datasets: a real dataset and a synthetic dataset. To be more concrete, we shall use the real dataset to show the effectiveness and efficiency of our algorithm, while the synthetic dataset will demonstrate the parameter setting mechanism, as well as the method's scalability. All experiments are done on a Microsoft Windows XP machine with an Intel Core 2 Duo 2.5G CPU and 3GB main memory. Programs are written in Java.

The mining process works by randomly summarizing a graph database, finding patterns from the summaries, and then verifying obtained patterns. As we briefly discussed in Section 2, to handle edges with multiple labels during the mining step, we modify gSpan and store a label list with each edge in the graph: A pattern matching will be successful as long as the pattern's corresponding edge label is covered by this list. For the verification step, we shall try alternative schemes (*e.g.*, top-down, bottom-up), and the optimization that leverages summary-guided isomorphism checking (see Section 5) will be adopted by default.

7.1 Real Dataset

Program Analysis Data. Program dependence graphs appear in software-security applications that perform characteristic analysis of malicious programs [5]. The goal of such analysis is to identify subgraphs that are common to many malicious programs, since these common subgraphs represent typical attacks against system vulnerabilities, or to identify contrast subgraphs that are present in malicious programs but not in benign ones, since these contrast subgraphs are useful for malware detection. In our experience and as reported by anti-malware researchers, these representative program subgraphs have less than 20 vertices.

We collected dependence graphs from 6 malware families, including W32.Virut, W32.Stration, W32.Delf, W32.Ldpinch, W32.Poisonivy and W32.Parite. These families exhibit a wide range of malicious behaviors, including behaviors associated with network worms, file-infecting viruses, spyware and backdoor applications. In a dependence graph, vertices are labeled with program operations of interest and the edges represent dependency relationships between operations. For example, when the operations are system or library calls, then an edge with label $y = f(x)$ between two vertices v_1 and v_2 captures the information that the system call at v_1 assigns the variable x and the second system call uses the variable y whose value is derived from x. Such dependence graphs are quite large in practice, sometimes with vertex counts up to $20,000$ and edge counts an order of magnitude higher (up to $220,000$ based on our observation). For the experiment data we use, the average number of nodes for all graphs is around 1,300.

Before we move on, let us assume for now that all parameters in Section 7.1 are already set to the optimal values. Detailed discussions on how this is achieved will be covered in Section 7.2.

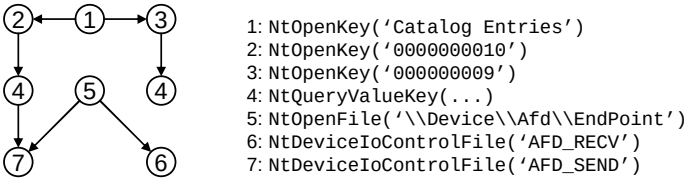

Figure 7: A Sample Malware Pattern

Figure 7 shows a graph pattern discovered from the Stration family of malware. Stration is a family of mass-mailing worms that is currently making its way across the Internet. It functions as a standard mass-mailing worm by collecting email addresses saved on a host and sending itself to the recipients, which does display some characteristics of spyware as shown in the figure. The displayed signature corresponds

to a malware reading and leaking certain registry settings related to the network devices.

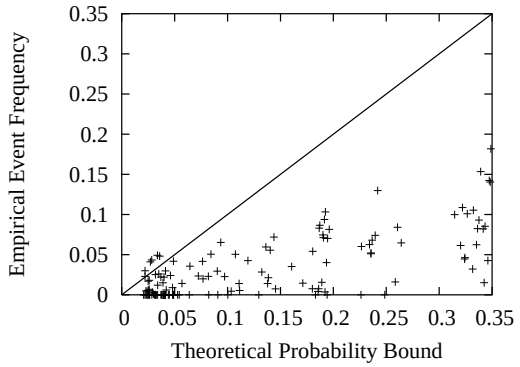

Figure 8: Theoretical Guarantee

In Figure 8, we plot the probability bound predicted in Theorem 1 against the empirical event frequency that is observed in experiments. Suppose there are X_j nodes with label l_j in a graph $G_i \in D$, we set x_j as $round\big(a_i \cdot \frac{X_j}{\sum_{j=1}^{L} X_j}\big)$, where a_i is the number of nodes to be kept for each database graph. In this way, labels that appear more often in the original graphs will also have more presence in their summarized versions, which is reasonable. Let A be the average number of nodes for graphs in the original database and a be the corresponding number after summarization, the summarization ratio is defined as $\alpha = A/a$. We set $min\_sup = 55\%$ (note that, for a graph dataset with big transaction size, min_sup is often set relatively high since small structures are very easy to be contained by a large graph; thus, there would be too many patterns if the support threshold is low), $min\_sup' = 45\%$, $\alpha = 8$, and randomly pick 300 patterns from the output of iteration 1. For each pattern p, we count its support $s = |D_p|$ in the original database D, compute $q(p)$ based on the distribution of p's vertex labels according to Lemma 2, and fix $s' = 70\% \cdot s$ to calculate the theoretical guarantee of $\text{Prob}\big[|D'_p| \leq s'\big]$ as given in the right hand side of Theorem 1, which is drawn on the x-axis. Then, we further generate 100 copies of D' based on randomized summarization, obtain the percentage of times in which p's support $|D'_p|$ really falls below s', and draw it on the y-axis. Patterns whose vertices are all associated with distinct labels have been omitted, because they can never miss.

It can be seen that our probabilistic guarantee is quite safe, where only very few points exist whose empirical frequencies go beyond the corresponding theoretical bounds, which is possible, because the frequency values calculated by such random sampling may not represent true probabilities. On the other hand, it also shows that real false negative rate is often not that high. So, we probably do not have to be too conservative when setting the new support threshold min_sup', due to the three effects we pointed out in the proof of Theorem 1.

In Figure 9, we draw the running time with regard to min_sup' after fixing $min\_sup = 55\%$, $\alpha = 8$, and compare relative performances of the three strategies we proposed in Section 6. Here, two iterations are processed, while one can also increase the number of rounds t to further bring down the pattern miss rate. Based on the testing results, it

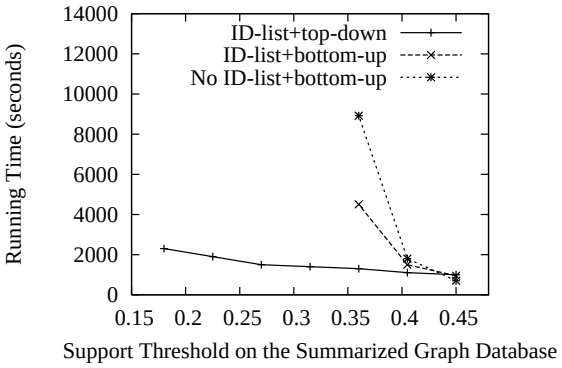

Figure 9: Three Alternative Strategies

seems that we are better off using verified ID-lists, because they are very effective in pruning false positives. Suppose a pattern p is mined from D' and after verifying it against D we find that p's support in the original database is less than min_sup, then for ID-list+top-down, we will terminate immediately without growing to p's supergraphs. However, considering No ID-list+bottom-up, as long as the support of these supergraphs in D' is greater than min_sup', they will all be generated and then verified as a batch at the end of each iteration. The advantage of such pre-pruning starts to prevail when min_sup' becomes smaller, which induces more false positives. Based on similar reasoning, the curve for ID-list+bottom-up turns out to appear in the middle, since pruning cannot happen in the first round but it can act in the second round. Finally, due to its general superiority, for the rest of this section, we shall use ID-list+top-down as our default implementation of SUMMARIZE-MINE, without further notices.

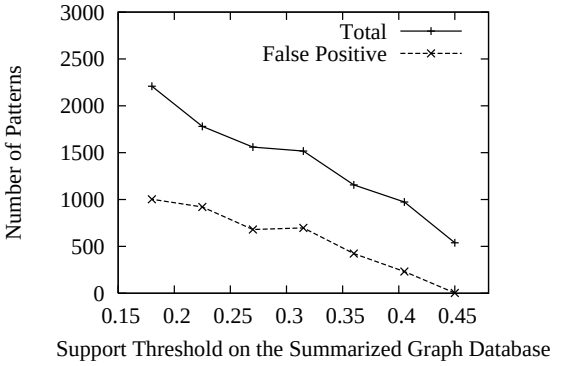

Figure 10: Number of Output Patterns

Figure 10 shows the corresponding number of patterns based on the same setting as Figure 9, and we also add another curve depicting the fraction of false positives that is verified and discarded by the ID-list+top-down strategy. As expected, when min_sup' is reduced, false negatives decrease while false positives increase. The gap between these two curves corresponds to the number of subgraphs that are truly frequent in the original database D, which gradually widens as we move to the left of the picture, since SUMMARIZE-MINE can now catch more patterns above min_sup'.

Accordingly, the price paid for this is an increased cost to mine the summarized database D' and verify against D.

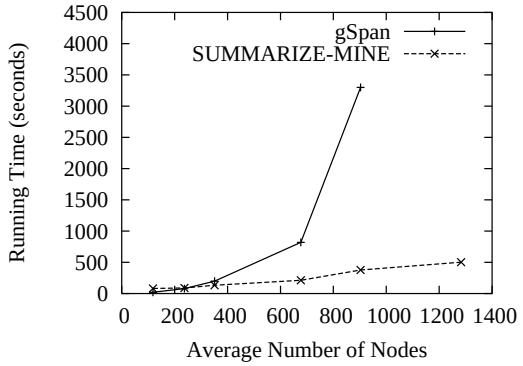

Figure 11: Efficiency w.r.t. Transaction Size

We compare the performance of gSpan, a state-of-art graph miner, with SUMMARIZE-MINE in Figure 11. For this experiment, a series of connected subgraphs are randomly drawn from each transaction, so that we can run both algorithms on graphs with different size and see whether there exists any trend. All other settings are the same as Figure 9, except that we only run one iteration here. Obviously, when the transaction size goes up, it becomes harder and harder for gSpan to work, where we have omitted the rightmost point of this curve since gSpan cannot finish within 3 hours. In comparison, SUMMARIZE-MINE remains somewhat stable, which is natural, because the embedding enumeration issue becomes much worse for large graphs, and our algorithm is specifically designed to tackle this problem.

7.2 Synthetic Dataset

Generator Description. The synthetic graph generator follows a similar mechanism as the one used to generate itemset transactions, where we can set the number of graphs (D), average size of graphs (T), number of seed patterns (L), average size of seed patterns (I) and number of distinct vertex/edge labels (V/E). To begin with, a set of L seed patterns are generated randomly, whose size is determined by a Poisson distribution with mean I; then, seed patterns are randomly selected and inserted into a graph one by one until the graph reaches its size, which is the realization of another Poisson variable with mean T. Due to lack of space, we refer interested readers to [15] for further details.

Figure 12 considers the problem of optimally setting the new support threshold min_sup' to achieve best algorithmic efficiency while ensuring a specific probabilistic guarantee, *i.e.*, the overall false negative rate is at most $\epsilon = 0.05$. Considering the total running time, intuitively, with a low min_sup', we would miss fewer patterns in one round and thus may require a smaller number of iterations to reach the desired ϵ; however, it is also true that more time has to be spent in each round. So, what is the best tradeoff? Since one-round miss rate as predicted by Corollary 1 is monotonically decreasing in $q(p)$, we can make the following statement. Focusing on a particular value of $q(p) = \theta$, if under this setting, we can guarantee that the overall false negative rate $(1 - P)^t$ is at most ϵ, then for all patterns p' with $q(p') \geq \theta$, the probability for them to miss from the output must be less than ϵ, too. This θ value can be adjusted to

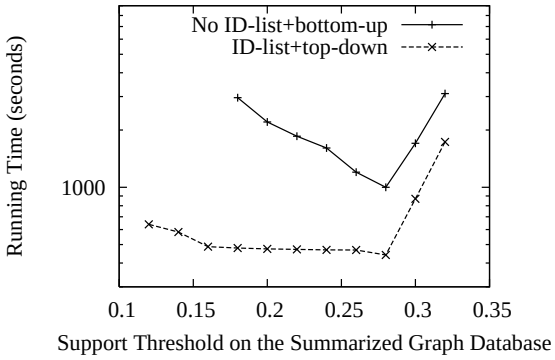

Figure 12: The Optimal Setting of min_sup'

tune SUMMARIZE-MINE accordingly toward larger/smaller patterns or patterns with more/less identically labeled vertices.

Setting $\theta = 0.8$ (which we think is reasonable for the mining task in hand), the total number of rounds t can be easily determined based on a given value of min_sup': Here, t is calculated by the formula $t = \lceil \frac{\log \epsilon}{\log(1-P)} \rceil$, where $1 - P$ has been substituted by the probability bound given in Corollary 1. Running SUMMARIZE-MINE for t iterations, we can draw the total computation time against min_sup', which is shown in Figure 12. The synthetic dataset we take is D400T500L200I5V5E1, *i.e.*, 400 transactions with 500 vertices on average, which are generated by 200 seed patterns of average size 5; the number of possible vertex/edge labels is set to 5/1. Considering the graphs we generated above, each transaction has approximately the same size, and thus it is reasonable to retain an equal number of $a = 50$ vertices for all summaries. min_sup is set to 40%. Finally, the lowest running time turns out to be reached at $min\_sup' = 28\%$ for both ID-list+top-down and No ID-list+bottom-up, where because of its ability to pre-prune at the very beginning, ID-list+top-down is not influenced much when min_sup' becomes low, which enables us to include more points for the corresponding curve when it is extended to the left. Also, the running time is not quite sensitive to parameter choices, as long as min_sup' is not too high.

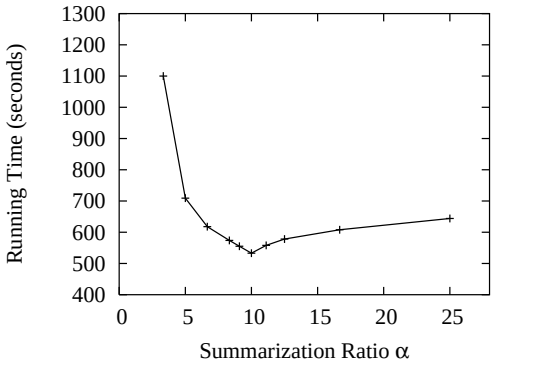

Figure 13: Running Time w.r.t. Summarization Ratio

In Figure 13, we analyze the impact of summarization ratio on our algorithm. The dataset is D500T500L200I5V5E1.

We vary α from 3.33 to 25 (outer loop), while min_sup' is implicitly tuned to the best possible value as we did in Figure 12 (inner loop). It can be seen that, $\alpha = 10$ happens to be the optimal position: When we summarize more, data graphs become smaller, which makes it faster to mine frequent subgraphs over the summaries; however, in the meantime, topology collapsing also introduces more false negatives and false positives, where additional computing resources must be allocated to deal with them. In this sense, it is important to run SUMMARIZE-MINE at the best trade-off point; and as we can see from the figure, there are actually a broad range of summarization ratios with reasonable performance.

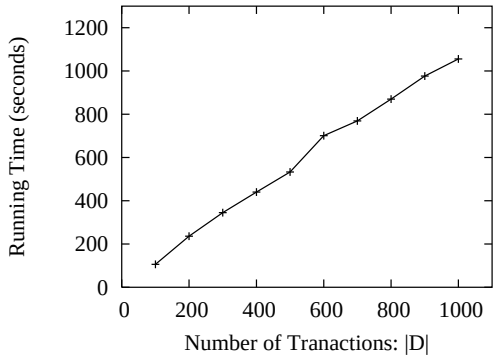

Figure 14: Scalability Test

Taking D($|D|$)T500L200I5V5E1, we also test the efficiency of our algorithm over ten datasets by varying the number of transactions $|D|$ from 100, 200 up to 1,000, which is shown in Figure 14. We use $min\_sup = 40\%$, $\alpha = 10$, while min_sup' and number of rounds t are tuned and optimally set as we did in Figure 12. As demonstrated, the implementation is highly efficient, which can finish in hundreds of seconds, and SUMMARIZE-MINE is linearly scalable with regard to the data size.

8. RELATED WORK

Many efficient frequent subgraph mining algorithms have been proposed, including FSG [15], gSpan [28], AGM [13], followed by Path-Join, MoFa, FFSM, GASTON, *etc.*, and the wealth of literature cannot be fully enumerated here. Owing to more recent development, now we are also able to mine maximal graph patterns [12], significant graph patterns [10], and patterns with topological constraints [20]. All these methods directly take the input graph database without any data reduction. This strategy works fine for a database consisting of small graphs. However, when the graphs contain a large number of pattern embeddings, all these methods could not perform efficiently, as we analyzed in the introduction.

There is another line of research [16, 4] that specifically mines frequent graph patterns from a single large network. Their major contribution is to define the pattern support in a single graph G, *i.e.*, how many times should we count a pattern, given all its embeddings in G that might overlap? These methods are often restricted to sparse networks or networks with a good number of labels, which limits the number of embeddings.

There have been a few studies on how to improve the efficiency of graph mining in general. However, they approach the problem from different angles, and none of them could tackle the intrinsic difficulty of embedding enumeration in bulky graph datasets. To name a few, [27] introduces structural leap search and leverages structural similarity to mine significant graph patterns. [9] invents a randomized heuristic to traverse the pattern space, where a collection of representative patterns are found. It analyzes how to reduce pattern candidates, based on the observation that many of them are quite similar. These two methods still work on the pattern space: Instead of doing a normal traversal, they can either perform "leap" or pick "delegates". To improve the mining speed on a large sparse graph, [23] decides to incorporate parallel processing techniques, which are orthogonal to the focus of SUMMARIZE-MINE.

The concept of summarizing large graphs in order to facilitate processing and understanding is not new [11]. [22] studies the problem of compressing Web graphs so that the link information can be efficiently stored and easily manipulated for fast computation of PageRank; [24] further analyzes how the sketches can help calculate approximate personalized PageRank. [2] develops statistical summaries that analyze simple graph characteristics like degree distributions and hop-plots. [19] approximates a large network by condensing its nodes and edges, which can preserve the original topological skeleton within a bounded error. Recently, [25] suggests a semantics-oriented way to summarize graphs by grouping vertices based on their associated attributes, which reflects the inherent structures and promotes easy user navigation; [3] further integrates this notion into a generic topological OLAP framework, where a graph cube can be built. The mining algorithm we developed in this paper can be further combined with all these studies to examine how structured patterns are presented on the summarized level.

Regarding other data reduction techniques that can be applied, we have pointed out sampling [26] and FP-Growth [8] as two examples that either reduce the number of transactions or compress between transactions, which are different from our compression method that takes effect within transactions. For a given pattern, because subgraph isomorphism checking and associated embedding enumerations happen inside a target graph, any method that cannot dig into individual transactions does not help. For instance, if we want to sample, then the sampling of nodes/edges/substructures must keep their original characteristics intact, so as to preserve the underlying patterns. This may require us to assume a generic graph generation model like the one given in [18]. In contrast, SUMMARIZE-MINE does not need such assumptions, the theoretical bound we developed is only conditional on the random grouping and merging of nodes, which can be easily implemented.

Finally, within a bigger context, the method of creating and leveraging synopsis to facilitate data processing has received significant attention in the broad database field [7, 30]. There is a recent work [17] on bursty sequence mining that transforms consecutive, identically-labeled items within the same transaction into intervals for the purpose of length reduction. However, as the data becomes more complex and takes the form of graphs, compression based on randomized mechanisms plays a key role in pattern preserving, which is a major contribution of this study. For example, in XS-

KETCH [21], the same set of nodes in the XML graph are often merged together, which could cause much pattern loss if we perform mining on such kind of summaries.

9. CONCLUSIONS

In this paper, we examine an important issue in frequent graph pattern mining, the intrinsic difficulty to perform embedding enumeration in large graphs, which might block many important downstream applications. Mining bulky graph datasets is in general very hard, but the problem should still be solvable if the node/edge labeling is not very diverse, which limits the explosion of pattern space. As we tried to find out the bottleneck, it was observed that even for small and simple substructures, the corresponding mining process could be very slow due to the existence of thousands of isomorphic embeddings in the target graphs. So, different from previous studies, SUMMARIZE-MINE proposes a novel mining framework that focuses on *data space reduction within transactions*, and effectively turns lossy compression into a virtually lossless method by mining *randomized summaries* for multiple iterations. Experimental results on real malware data demonstrate the efficiency of our method, which can find interesting malware fingerprints that were not revealed previously. Moreover, SUMMARIZE-MINE also sheds light on how data compression may impact the underlying patterns. This will be particularly interesting, given an emerging trend of huge information networks that must adopt data reduction as a necessary preprocessing step for analytical purposes.

10. REFERENCES

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

[2] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Survey*, 38(1):1–69, 2006.

[3] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[4] J. Chen, W. Hsu, M.-L. Lee, and S.-K. Ng. Nemofinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs. In *KDD*, pages 106–115, 2006.

[5] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC/SIGSOFT FSE*, pages 5–14, 2007.

[6] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1036–1050, 2005.

[7] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes (tutorial). In *VLDB*, 2001.

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

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

[10] H. He and A. K. Singh. Efficient algorithms for mining significant substructures in graphs with quality guarantees. In *ICDM*, pages 163–172, 2007.

[11] L. B. Holder, D. J. Cook, and S. Djoko. Substucture discovery in the subdue system. In *KDD Workshop*, pages 169–180, 1994.

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

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

[14] S. Kramer, L. D. Raedt, and C. Helma. Molecular feature mining in hiv data. In *KDD*, pages 136–143, 2001.

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

[16] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[17] A. Lachmann and M. Riedewald. Finding relevant patterns in bursty sequences. *PVLDB*, 1(1):78–89, 2008.

[18] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[19] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD Conference*, pages 419–432, 2008.

[20] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *KDD*, pages 228–238, 2005.

[21] N. Polyzotis and M. N. Garofalakis. Xsketch synopses for xml data graphs. *ACM Transactions on Database Systems*, 31(3):1014–1063, 2006.

[22] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *ICDE*, pages 405–416, 2003.

[23] S. Reinhardt and G. Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IPDPS*, pages 1–8, 2007.

[24] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz. To randomize or not to randomize: Space optimal summaries for hyperlink analysis. In *WWW*, pages 297–306, 2006.

[25] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD Conference*, pages 567–580, 2008.

[26] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[27] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, pages 433–444, 2008.

[28] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

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

[30] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *ICDE*, pages 54–65, 2004.