Mining Top-K Large Structural Patterns in a Massive Network

Feida Zhu
Singapore Management University
fdzhu@smu.edu.sg

Qiang Qu
Peking University
quqiang@cis.pku.edu.cn

David Lo
Singapore Management University
davidlo@smu.edu.sg

Xifeng Yan
University of California at Santa-Barbara
xyan@cs.ucsb.edu

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

Philip S. Yu
University of Illinois at Chicago
psyu@cs.uic.edu

ABSTRACT

With ever-growing popularity of social networks, web and bio-networks, mining large frequent patterns from a single huge network has become increasingly important. Yet the existing pattern mining methods cannot offer the efficiency desirable for large pattern discovery. We propose SpiderMine, a novel algorithm to efficiently mine top-K largest frequent patterns from a single massive network with any user-specified probability of $1-\epsilon$. Deviating from the existing edge-by-edge (*i.e.*, incremental) pattern-growth framework, SpiderMine achieves its efficiency by unleashing the power of small patterns of a bounded diameter, which we call "spiders". With the spider structure, our approach adopts a probabilistic mining framework to find the top-k largest patterns by (i) identifying an affordable set of promising growth paths toward large patterns, (ii) generating large patterns with much lower combinatorial complexity, and finally (iii) greatly reducing the cost of graph isomorphism tests with a new graph pattern representation by a multi-set of spiders. Extensive experimental studies on both synthetic and real data sets show that our algorithm outperforms existing methods.

1. INTRODUCTION

Graph data arise naturally in a wide range of application domains including bioinformatics, semantic web [1], link analysis [2] and terrorist-network detection. In both literature and practice, the terms "graph" and "network" nowadays often share the same meaning. We would therefore use the two terms interchangeably in this paper. It has long been recognized that graph patterns, often the smaller ones, are highly useful in many important stages of graph data analysis and graph database management such as indexing [3, 4]. As a result, a number of efficient algorithms have been developed to mine frequent patterns in the graph-transaction setting where the input is a graph database consisting of a large set of graphs and the support of a pattern is the number of graphs in the database that contain the pattern, e.g., [5, 6, 7, 8, 9, 10, 11]. However, the recent years have seen an ever-growing number of applications on single large graphs involving homogeneous or heterogenous attributed nodes with complicated connections, such as social networks, transportation networks, bio-networks and cyber-physical networks. Moreover, graph patterns of large sizes have become increasingly important in many applications for the following reasons: (I) *Large patterns are a natural result of ever larger graph data.* For social network analysis on a network like that of Facebook or Twitter, it has been shown that functional communities could reach size up to ≈ 150 [12], much larger than what can be mined by existing methods considering the input graph size. Similarly, for web structure mining in today's Internet, one should expect the real web structures mined for any domain to be fairly complicated. (II) *Large patterns are more informative in characterizing large graph data.* For example, in DBLP co-authorship network, small patterns, e.g., several authors collaborate on a paper, are almost ubiquitous. As shown in our experiments, it is only with large patterns could we discover interesting common collaborative patterns, or distinguish distinct patterns, across different research communities. Take another example from software engineering, large patterns uncovered from program structure data would reveal software *backbones* which are critical in analyzing large software packages and understanding legacy systems [13, 14, 15, 16, 17]. Appendix 8.4 further discusses the applications of large network patterns.

Despite the significance of the task, mining large patterns from single large networks is a hard problem presenting challenges from a number of aspects. First, the single-graph setting, which is the focus of this paper, introduces additional complexity in pattern support computation. Algorithms developed for the graph-transaction setting cannot be used to solve the single-graph setting due to the complexity of overlapping embeddings. Second, the well-known difficulties inherent in graph mining, i.e., the exponentially high combinatorial complexity, only get exacerbated when the sizes grow significantly larger for both the input graph

Proceedings of the VLDB Endowment, Vol. 4, No. 11

and the output patterns. Our solution to this dilemma is to mine only the top-K largest patterns by identifying a manageable set of promising growth paths leading to the largest patterns probabilistically, and recovering them efficiently. Our approach, which we call SpiderMine, is based on the concept of r-*spider*, or *spider* in short. A spider is a frequent subgraph with a head vertex u such that all other vertices of the subgraph are reachable from u by a path up to length r. The efficiency of SpiderMine comes from the power of using these spiders to attack the three bottlenecks in large pattern discovery as mentioned earlier. In particular, SpiderMine uses random selection of seed spiders and iterative spider-growth combined with pattern merging to identify large patterns such that given any user-specified error threshold ϵ, with probability at least $1-\epsilon$, SpiderMine returns the top-K largest patterns in a single large network.

Our main technical contributions are: (1) We propose the concept of r-spider. We observe the fact that large patterns share these small components and thus can be obtained much efficiently by assembling them in a well-designed manner. We also propose a new graph pattern representation based on spiders to reduce the cost of graph isomorphism test. (2) We propose a novel and efficient mining algorithm, SpiderMine, based on the concept of spiders, to mine the top-K largest patterns from single graphs with user-specified probability. (3) We conduct extensive experiments on both synthetic and real data to analyze the performance of SpiderMine and demonstrate its superiority over existing methods in terms of effectiveness and efficiency. With real data, we also illustrate the application of large patterns in social network analysis and software engineering.

Road-map. Related work is discussed in Section 2. Section 3 gives the problem setting and formulation. Section 4 provides an outline of our algorithm and the underlying design idea. Section 5 presents experimental results on synthetic data sets. We conclude our paper in Section 6. Appendix section 8 gives discussion, proof and algorithm details as well as more extensive experimental results on both synthetic and real data.

2. RELATED WORK

Mining in single-graph setting has not been as well studied as in graph-transaction setting due to the complexity of support computation. As we focus on the single-graph setting, works developed for single graphs are presented with higher priority. SUBDUE [18] is probably the most well-known algorithm for mining frequent subgraphs in a single graph. It achieves its efficiency by using approximation and finding patterns that can compress the original graph by substituting the patterns with a single vertex. The heuristics used in SUBDUE makes it hard to find frequent patterns of larger sizes. As shown in [19] and verified by our experiments, SUBDUE focuses on small patterns with relatively high frequency and scales poorly as the data size increases. Vanetik et al. [20] proposed an algorithm to use the maximum number of edge-disjoint embeddings of a graph pattern as the measure of its support. A level-by-level approach is used to mine all frequent subgraphs from a single labeled graph. The authors only showed the completeness of the mining result and discussed little on particular heuristics for efficiency improvement other than the downward closure property of support. The performance of their algorithm has only been shown on data of very small scale (around 100 edges). An algorithm called SEuS was proposed by Ghazizadeh and Chawathe in [21] which uses a data structure *summary* to collapse all vertices of the same label together and prune infrequent candidates. This technique is useful at the presence of a relatively small number of highly frequent subgraphs, and less powerful at handling a large number of frequent subgraphs with low frequency. Our experiments show that SEuS returns mostly small structures for almost all datasets. Kuramochi and Karypis [22] gave a mining framework to find the complete set of frequent patterns in a large sparse graph. Following an enumeration-and-verification paradigm, subgraphs of size k are joined for growth only if they share a certain subgraph of size $k-1$, reducing the repeated visits to the same node in the pattern lattice. A structure called "anchor-edge-list" is used to roughly mark the edge-disjoint embeddings of a frequent pattern in order to reduce the cost of graph isomorphism test. While this algorithm showed good performance on small graphs or large sparse graphs, mining complete results in general large graphs or scale-free graphs common in web and social network analysis is inherently infeasible. SpiderMine therefore strives only to return the top-K largest patterns and exploits a novel graph representation by spider-set which not only significantly reduces the number of unnecessary graph isomorphism tests but also provides a compact and efficient way to exactly store all embeddings of a large frequent pattern. The same authors of [22] also proposed in [19] an algorithm GREW to mine incomplete set of subgraph patterns by iteratively merging subgraphs connected with one or multiple edges. While GREW could discover some large patterns quickly, no guarantee is given on the pattern quality in relation to the complete pattern set. SpiderMine is designed to return the top-K largest patterns in the input graph with a high probability of $1-\epsilon$ for any user-specified error bound ϵ. Note also that both algorithms in [22] and [19] find only patterns with edge-disjoint or even vertex-disjoint ([19]) embeddings. A different yet more general support definition based on "harmfulness" of an embedding overlapping was proposed by Fiedler and Borgelt in [23]. SpiderMine adopts this general support definition for a wider range of applications. The authors of [23] proposed MoSS for mining complete patterns in single graphs, which, as any other algorithm mining for the complete pattern set, suffers from the same scalability issue as the input graph size grows.

Although SpiderMine is designed to handle the harder case of mining in single-graph setting, it can be adapted to graph-transaction setting with no difficulty. Many efficient algorithms have been developed to find the complete frequent pattern set, e.g., AGM by Inokuchi et al., [7, 24], FSG by Kuramochi and Karypis, [11], Borgelt and Berthold, [5], gSpan by Yan and Han, [6] and FFSM by Huan et al., [8]. All these algorithms suffer from the fact that due to combinatorial complexity, the size of the complete pattern set is exponential even for graphs of moderate sizes. SPIN [10] and MARGIN [25] find all maximal patterns. Unfortunately, the number of all maximal patterns could still be too large to handle as a maximal pattern could in fact be of any size. In the worst case, all closed patterns could be maximal. As pointed out in [26], most of the time, what is really needed, if not at all feasible, is a manageable set of patterns meeting users' constraints. ORIGAMI as proposed in [26] is such an algorithm, which aims to find a representative pattern set. Our algorithm SpiderMine, on the other hand, serves a dif-

ferent purpose which is to find a set of large patterns. The two algorithms cannot replace each other as shown in our experiments, ORIGAMI would return a mixed set of small and medium-sized patterns at the cost of missing most of the large distinct patterns. Other works include structural leap search introduced by Yan et al. [27], which adopts structural similarity to mine significant graph patterns efficiently and directly from two graph datasets. Yet the leap search in [27] follows an edge-by-edge growing strategy, while the merge operation employed in SpiderMine could jump with multiple edges in the search space, thus significantly shortening the search time. Large patterns are important in various domains other than graphs. For example, the problem of mining large patterns has also been studied in item-set settings [28].

3. PROBLEM FORMULATION

As a convention, the *vertex set* of a graph G is denoted by $V(G)$ and the *edge set* by $E(G)$. The size of a graph P is defined by the number of edges of P, written as $|P|$. In our setting, a graph $G = (V(G), E(G))$ is associated with a labeling function $l_G : V(G) \mapsto \Sigma, \Sigma = \{\varsigma_1, \varsigma_2, \ldots, \varsigma_k\}$. Graph isomorphism in our problem setting requires matching of the labels for each mapped pair of vertices. Our method can also be applied to graphs with edge labels.

DEFINITION 1. **(Labeled Graph Isomorphism)** *Two labeled graphs G and G' are isomorphic if there exists a bijection $f : V(G) \mapsto V(G')$, such that $\forall u \in V(G)$, $l_G(u) = l_{G'}(f(u))$ and $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(G')$.* ∎

We use $G \cong_L G'$ to denote that two labeled graphs G and G' are isomorphic. Given two graphs P and G, a subgraph G' of G is called an *embedding* of P in G if $P \cong_L G'$. For a single graph G and a pattern P, we use e_P to denote a particular embedding of a pattern P, and the set of all embeddings of P is denoted as $E[P]$. We denote as P_{sup} the support set for a pattern P. In single graph setting, $P_{sup} = E[P]$ while in graph transaction setting P_{sup} is the set of graphs of the database each containing at least one embedding of P. For a graph pattern P and a vertex $v \in V(P)$, if the shortest distance between v and any other vertex in $V(P)$ is at most r, we say P is *r-bounded* from v. r is also called the *radius* of P. The *diameter* of a connected graph G is the maximum over the shortest distances between all pairs of vertices in $V(G)$, and is denoted as $diam(G)$. In real applications, it has been recently observed that the diameter of a graph is often bounded by a constant which is not too large and even shrinks over time [2]. For example, in DBLP, the effective diameter, i.e. the 90^{th} percentile distance, is bounded by 9. In IMDB data, the diameter is bounded by 10 [29]. Effective methods have also been proposed to gauge the diameter of a graph with fairly good accuracy [30]. As such we assume a user-specified upper bound D_{max} for pattern diameter and focus on mining patterns with diameters bounded by D_{max}. We now define our problems in the single graph setting.

DEFINITION 2. **[Top-K Largest Patterns With Bounded Diameter]** *Given a graph G, a support threshold σ and a diameter upper bound D_{max}, the problem of mining* Top-K Largest Patterns With Bounded Diameter *is to mine the top-K largest subgraphs P of G such that $|P_{sup}| \geq \sigma$ and $diam(P) \leq D_{max}$.* ∎

Since mining the complete pattern set is infeasible and it is extremely difficult to obtain the exact solution for the top-K largest patterns without computing the complete pattern set, we use a randomized framework to compute the top-K largest patterns with high probability.

DEFINITION 3. **[Approximate Top-K Largest Patterns With Bounded Diameter]** *Given a graph G, a support threshold σ, a diameter bound D_{max} and an error threshold ϵ, the problem of mining* Approximate Top-K Largest Patterns With Bounded Diameter *is to mine a set S of K patterns such that, with probability at least $1 - \epsilon$, S contains all the top-K largest subgraphs P of G such that $|P_{sup}| \geq \sigma$ and $diam(P) \leq D_{max}$.* ∎

4. OUR APPROACH

The challenges of the problem are the following: (1) How to identify the top-K largest patterns with a high probability? and (2) How to quickly reach the large patterns? We address these two questions in the following subsections.

4.1 Approximate Top-K Large Patterns

As trying all the possible growth paths is unaffordable, we have to identify a small set of highly potential ones which would lead to the large patterns with good chance. Our solution is based on the following observation: *large patterns are composed of a large number of small components which would eventually become connected after certain rounds of growth.* The more of such small components of a large pattern we can identify, the higher chance we can recover it. Thus, we first mine all such small frequent patterns, which we call *spiders* that will be formally defined later. Compared with small patterns, large patterns contain far more spiders as their subgraphs. It follows that if we pick spiders uniformly at random from the complete spider set, the chance that we pick some spider within a large pattern is accordingly higher. Moreover, if we carefully decide on the number of spiders we would randomly pick, the probability that multiple spiders within P would be chosen is higher if P is a larger pattern than a smaller one. We denote the set of all spiders within P which are initially picked in the random draw as H_P. According to our observation, for any two spiders in H_P, there must be a pattern growth path such that along the path their super-patterns will be able to merge. And we are going to catch that as follows. Once we picked all the spiders, they will be grown to larger patterns in λ iterations where λ will be determined by D_{max}. In each iteration, each spider will be grown in a procedure called SpiderGrow(), which always expands the current pattern by appending spiders to its boundary such that the pattern's radius is increased by r. Also, in each iteration, two patterns will be merged if some of their embeddings are found to overlap and the resulting merged pattern is frequent enough. Now for any large pattern P, we have the following Lemma,

LEMMA 1. *For any pattern P with diameter upper-bound D_{max}, let SpiderGrow(Q) be a procedure which grows a pattern Q such that the radius of Q is increased by r, then all patterns growing out of H_P which are sub-patterns of P must have merged into one sub-pattern of P after $\lambda = \frac{D_{max}}{2r}$ iterations of running SpiderGrow(Q).*

This means that as long as we pick more than one spider within a large pattern P in the initial random draw, i.e., $|H_P| > 1$, we can guarantee we will not miss P by retaining all the merged patterns. On the other hand, for smaller patterns, the probability that more than one spider within the pattern get picked in the random draw is much lower than that of large patterns. As such, keeping only the merged patterns at the end of the iterations would highly likely prune away patterns that would grow only toward small patterns. Thus after the pruning, we are left with a small number of candidates each of which, with high probability, is a subgraph of large patterns. We then use SpiderGrow() again to further extend these candidates until no larger patterns can be found.

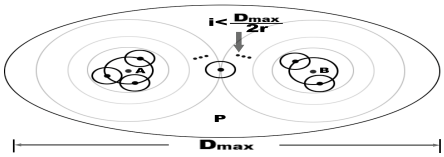

Figure 1: SpiderMine

To formally present our approach, we first define the concept of a spider. Formally, an r-spider is defined as follows.

DEFINITION 4. [**r-spider**] *Given a frequent pattern P in graph G and a vertex $u \in V(P)$, if P is r-bounded from u, we call P an r-spider with head u.* ∎

Algorithm 1 SpiderMine

Input: input graph G, error bound ϵ, support threshold σ
diameter bound D_{max}, number of patterns returned K
Output: Set of K patterns S
1: $S \leftarrow \emptyset$;
2: $T(G) \leftarrow$ InitSpider(G, r, σ);
3: /* mine all patterns of diameter up to $2r$;*/
4: Compute M based on $T(G)$, ϵ and K;
5: $S \leftarrow$ RandomSeed$(T(G), M)$;
6: /*randomly select M spiders as the seed for growth*/
7: **For** $i = 1$ **To** $\frac{D_{max}}{2r}$
8: $S \leftarrow$ SpiderGrow(S, σ);
9: /* grow patterns by spiders and merge whenever possible*/
10:Prune unmerged patterns from S;
11:**Do**
12: $S \leftarrow$ SpiderGrow(S, σ);
13:**Until** no new patterns can be found
14:$S \leftarrow$ top K largest patterns in S;
15:**Return** S;

Our algorithm works in the following three stages. An illustration is given in Figure 1. The main algorithm of SpiderMine is shown in Algorithm 1 with all details elaborated in Appendix. The random drawing size M is an internal parameter computed according to user-specified K and ϵ, with details given in the next subsection. The discussion on setting the spider radius size r is given in Subsection 4.2.

1. **Stage I: Mining Spiders**
   Mine all r-spiders from the input graph G. By the end of this stage, we know all the frequent patterns up to a diameter $2r$ with all their embeddings in G.
2. **Stage II: Large Pattern Identification**
   Randomly pick M spiders from all the spiders obtained in Stage I as the initial set of frequent subgraphs. The next step consists of $\frac{D_{max}}{2r}$ iterations. In each iteration, use SpiderGrow() to grow each of the M subgraphs by extending its boundary with selected spiders such that the radius of the subgraph is increased by r. In each iteration, if we detect that two frequent subgraphs, whose embeddings are all previously disjoint, begin to overlap on some of their embeddings as a result of growth in this iteration, we would merge them if the resulting merged subgraph is frequent. Note that we can avoid pair-wise checking for potential merging because all patterns grow with spiders as units and we only have to monitor the same spiders being used by different patterns to detect overlapping. At the end of the $\frac{D_{max}}{2r}$ iterations, keep only those frequent subgraphs which are generated as a result of merging at some iteration. Let the set we keep be S. The frequent subgraphs in S are believed to be subgraphs of large patterns with high probability.
3. **Stage III: Large Pattern Recovery**
   With high probability, each one of the top-K large patterns now has some portion of it as a pattern in S. To recover the full patterns, we grow each subgraph in S by SpiderGrow() until no more frequent patterns can be found. All the patterns discovered so far are maintained in a list sorted by their size. Return the top-K patterns.

We now show that, in Stage II of SpiderMine, how to choose M, the number of initial seed spiders, to achieve the discovery of top-K largest patterns with guaranteed probability. If more than one spider within a pattern P are chosen in the random drawing process, we say that P is successfully identified. We denote as $P_{success}$ the probability that all the top-K largest patterns are successfully identified. With proof sketch detailed in the Appendix, we have the following lemma,

LEMMA 2. *Given a network G and a user-specified K, we have* $P_{success} \geq \left(1 - (M+1)(1 - \frac{V_{min}}{|V(G)|})^M\right)^K$.

V_{min} is the minimum number of vertices in a large pattern required by users, usually an easy lower bound that a user can specify. Now to compute M, we just need to set $\left(1 - (M+1)(1 - \frac{V_{min}}{|V(G)|})^M\right)^K = 1 - \epsilon$ and solve for M. It follows that, once the user specifies K and ϵ, we could compute M accordingly, and then if we pick M spiders initially in the random drawing process, we are able to return the top-K largest patterns with probability at least $1 - \epsilon$. For example, with $\epsilon = 0.1$, $K = 10$, and $V_{min} = \frac{|V(G)|}{10}$, we get $M = 85$, which means to return top 10 largest patterns(each of size at least $\frac{|V(G)|}{10}$ if any) with probability at least 90%, we need to randomly draw 85 spiders initially. With the analysis above, it is not hard to prove the following theorem.

THEOREM 1. *Given a graph G, the error bound ϵ, the diameter upper bound D_{max}, the support threshold σ and K, with probability at least $1 - \epsilon$, SpiderMine returns a set S of top-K largest subgraphs of G such that for each $P \in S$, $|P_{sup}| \geq \sigma$ and $diam(P) \leq D_{max}$.*

4.2 Spider: Leaping Towards Large Patterns

We show why spiders could help recover large patterns efficiently by the following arguments: (1) Spiders reduce combinatorial complexity in recovering large patterns, and (2) Spiders minimize the heavy cost of graph isomorphism checking.

4.2.1 Reducing Combinatorial Complexity

It is well-known in graph mining that as the pattern sizes increase, the number of frequent subgraph patterns grows exponentially. Illustrated in the pattern lattice model, the patterns of small sizes are the "tip" of the lattice, forming a tiny part of the whole pattern space when compared with the number of patterns of larger sizes. This leads us to the following observation — *larger patterns are composed of smaller subgraphs which are shared among all the larger patterns across the pattern space.* If we are able to identify these smaller components, we can generate larger patterns with lower combinatorial complexity than in incremental pattern growth model. An illustrative toy example is given in Figure 2. In the first row are 6 spiders each of size 10, denoted by A to F. In the second row are four larger patterns each of which is composed by three spiders from the first row. We assume a 20% overlapping on average. This means each larger pattern is of size $10 \times 3 \times 80\% = 24$. If we follow the traditional incremental growth paradigm, we need $24 \times 4 = 96$ steps to grow these four large patterns. In SpiderMine, we first mine out the 6 spiders in $10 \times 6 = 60$ steps, then the four large patterns will be generated by assembling these spiders in $3 \times 4 = 12$ steps. As such, we will take $60 + 12 = 72$ steps in total. We save 24 steps, a 30% speed-up. Although this is a much simplified example ignoring all other mining cost such as frequency checking and so on, it is evident that spiders could be very powerful in reducing the inherent cost associated with combinatorial complexity. Moreover, note that the cost for mining the spiders is only a one-time cost. On the other hand, we can run the remaining stages, i.e., the randomized seed selection and iterative spider-growth, multiple times to increase the probability of obtaining the top-K large patterns.

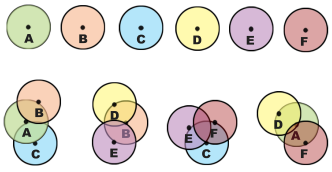

Figure 2: Patterns sharing the same set of spiders

We note that it is a hard problem to decide what is the optimal set of subgraphs such that this set is shared by the most number of larger patterns, thus minimizing the combinatorial complexity. However, as compared against traditional incremental growth, it still pays tremendously to leverage this finding and use subgraphs of small radius and relatively uniform structure to obtain larger patterns. How large should we set the spider radius r? A smaller r means slower growth to large patterns while a larger r means heavier load on the initial spider mining stage. Empirically, we find that a small r, e.g., $r = 1$ or $r = 2$, is a good trade-off choice which gives better overall mining performance because the quality of the top-K mining result is largely unaffected when we increase r further as a result of our probabilistic framework. Experiments on r is given in the Appendix.

4.2.2 Reducing Graph Isomorphism Checking

For a frequent pattern P and a vertex $v \in V(P)$, the r-neighborhood of v is a frequent subgraph, and accordingly an r-spider with head v. We denote an r-spider with head vertex v as $s^r_{h[v]}$ and is written as $s_{h[v]}$ for simplicity when r is fixed. In SpiderMine, each frequent pattern P is associated with a spider-set representation, which is denoted as $\mathbb{S}[P]$ and is defined as a multi-set $\mathbb{S}[P] = \{s_{h[v]} | v \in V(P)\}$. For the example shown in Figure 3 (I), the spider-set representation of the pattern consists of 8 distinct spiders in total, with one of the spiders having two embeddings and therefore 9 spiders in total. This shows the spider-set representation is a multi-set. Here we set the radius of the spider $r = 1$. The node underlined is the head of the spider. The spider's corresponding embeddings are given in the physical vertex ID. Note that one of the spider has two embeddings.

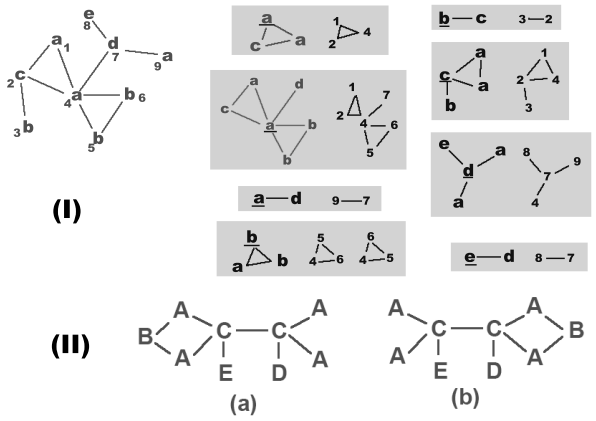

Figure 3: Spider-Set Representation

Note that it is natural to maintain the spider-set representation for a pattern P in SpiderMine. Initially, a frequent pattern P is simply a spider randomly chosen from the complete set of all spiders mined from the input graph G. Set $\mathbb{S}[P] = \{s_{h[v]}\}$ where v is the head of the chosen spider. For each $u \in V(P), u \neq v$, we run BFS for depth at most r to compute $s_{h[u]}$ within P, and add $s_{h[u]}$ into $S[P]$. This completes the initial computation for $\mathbb{S}[P]$. To update $\mathbb{S}[P]$ when P is extended by a new spider s' at boundary vertex u, we merge $\mathbb{S}[P]$ with $\mathbb{S}[s']$ and update those spiders whose heads are within distance r to the common boundary of P and s'. The reason to maintain a spider-set representation for a pattern P is to reduce graph isomorphism test as much as possible. Here is the observation — **If two graphs P and Q have the same spider-set representation, i.e., $\mathbb{S}[P] = \mathbb{S}[Q]$, it is with high probability that $P \cong_L Q$.** Intuitively, as the neighboring spiders overlap each other, the topological constraints imposed upon each other significantly limit the flexibility to construct two non-isomorphic graphs using exactly the same set of spiders. As in Figure 3 (I), it is in fact difficult to construct a graph Q with exactly the same set of 9 spiders such that Q is not isomorphic to the graph in the figure.

It is easy to prove the following theorem which shows that two isomorphic graphs must have the same spider-set representation.

THEOREM 2. *For two graphs P and Q, if $P \cong_L Q$, then $\mathbb{S}[P] = \mathbb{S}[Q]$.*

With the theorem, we have our *Spider-set Pruning: For two graphs P and Q, if $\mathbb{S}[P] \neq \mathbb{S}[Q]$, then P and Q cannot be isomorphic. Hence the isomorphic test can be skipped.* However, in general, it is not true for the other direction, i.e., two graphs with the same spider-set representation could be different in terms of graph isomorphism. We have not been able to show theoretical bounds for the pruning power of spider-set as a heuristic to reduce the number of graph isomorphism tests. In practice, spider-set representation has always been able to prune away non-isomorphic graphs. We also note that, as r gets larger, the pruning power will grow stronger by analysis. Intuitively, the radius of the spiders defines the spiders' capacity to constrain topology on the local neighborhood. A larger r corresponds to stronger mutual structural constraints, as illustrated in Figure 3 (II). The two graphs (a) and (b) are different. If we set $r = 1$ then they will have exactly the same spider-set representation, which would elude our heuristic for graph isomorphism test. Yet if we increase r to 2, they will have different spider-set representation.

5. EXPERIMENTAL RESULTS

In this section, we report our performance study on SpiderMine. Due to space limit, we present here some of the most important results on synthetic data. The Appendix contains more extensive experimental results on both synthetic and real data.

All experiments are conducted on an Intel(R) Core (TM) 2 Duo 2.53 GHz CPU, and 2 GB of main memory with Ubuntu 10.04. SpiderMine is implemented in Java (OpenJDK 1.6.0), while the other algorithms (the latest versions of SUBDUE [18] (version 5.2.1), SEuS [21](version 1.0), MoSS [23] (version 5.3) and ORIGAMI [26]) are all obtained from the original authors to whom we are greatly thankful.

We have carefully chosen the set of algorithms to be compared based on both our problem setting and the state-of-the-art in this area. Our goal is to find large frequent patterns from massive networks, especially in large single network setting. As mentioned in Section 2, despite the huge body of literature in graph mining, there are actually not many algorithms capable of the mining task in this setting due to the complication of embedding overlapping and support computation. The algorithms we selected are all representative works each with unique characteristics. SUBDUE is a classic approximate algorithm on single graphs. SEuS is a more recent one with improved heuristics. MoSS is the counterpart of gSpan aiming to mine for the complete pattern set in this setting, which is the state-of-the-art. As we aimed at a harder problem, we can also handle graph transaction setting. gSpan and FFSM cannot run to completion for most of our data sets as a result of the combinatorial complexity even to enumerate all the patterns. The same result is expected for other algorithms based on the comparison given in [31]. ORIGAMI is proposed to solve this problem, which is also the closest to our approach, and we therefore only compare against ORIGAMI for transaction setting.

GID	$\|V\|$	f	d	m	$\|V_L\|$	L_{sup}	n	$\|V_S\|$	S_{sup}
1	400	70	2	5	30	2	5	3	2
2	400	70	4	5	30	2	5	3	2
3	1000	250	2	5	30	2	5	3	20
4	1000	250	4	5	30	2	5	3	20
5	600	130	4	5	30	2	20	3	2

Figure 4: Data Settings

GID vs GID	difference in setting
2 vs 1	GID 2 doubles the average degree
3 vs 1	GID 3 increases the support of small patterns.
4 vs 3	GID 4 doubles the average degree
5 vs 2	GID 5 increases the number of small patterns.

Figure 5: Setting Difference

5.1 Synthetic Data

5.1.1 Single-Graph Setting

SpiderMine is experimented on single graphs generated with two models: (I) the Erdős-Rényi random network model, and (II) the Barabási-Albert scale-free network model.

Random Network.

The Erdős-Rényi model is a well-known model to generate random graphs. Using the $G(n, p)$ variant, our synthetic single graph is constructed by generating a background graph and injecting into it a set of large patterns as well as a set of small patterns. We generate 5 different data sets (labeled GID 1 to 5) with varied parameter settings. The differences among the data sets are described in Figure 5. The detailed description of the data sets is given in Figure 4. The description of the parameters is given as follows. $|V|$ is the number of vertices. f is the number of vertex labels. d is the average degree. $|V_L|$ (or $|V_S|$) is the number of vertices of each injected large (resp. small) pattern. m (or n) is the number of large (resp. small) patterns injected. L_{sup} (or S_{sup}) is the number of embeddings of each large (resp. small) pattern injected.

In this comparison, the scale of the synthetic data, e.g., the number of vertices, the average degree, has been purposely set small so that all the three algorithms (SUBDUE, SEuS and MoSS) are able to return results successfully.

Figures 6 to 10 show the distribution of patterns mined by SpiderMine, SUBDUE, SEuS and MoSS for different parameter settings in Figure 4. Overlapping bars indicate the same pattern size. The minimum support threshold has been set to a very low value of 2 in all these cases. We have following observations.

1. **SpiderMine.** In all 5 cases, SpiderMine successfully returns most of the largest patterns. Note that after 4 patterns of size 30 have been explicitly embedded into the background graph, the interconnections between the patterns and the background graph actually give rise to 10 largest patterns of size 30. Here we set $K = 10$, $D_{max} = 4$.

2. **SUBDUE.** SUBDUE focuses on small patterns that have relatively high frequency. In Figures 8 and 9, when the support of each small patterns increases, the mining result of SUBDUE shifts significantly toward smaller patterns. Interestingly, this is also true when the number of small patterns increases, as shown in Figure 10.

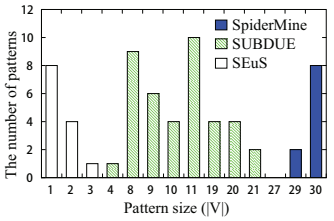

Figure 6: GID 1

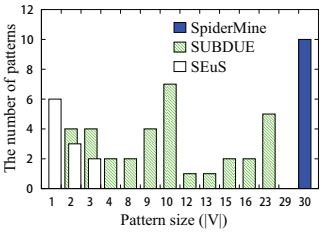

Figure 7: GID 2

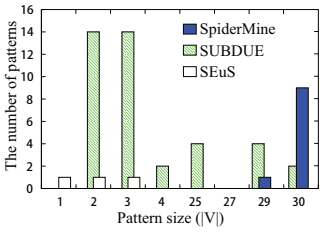

Figure 8: GID 3

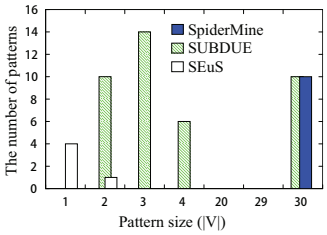

Figure 9: GID 4

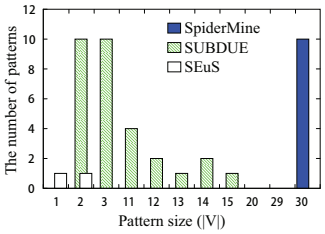

Figure 10: GID 5

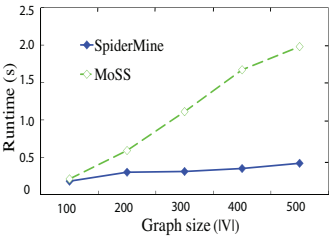

Figure 11: Runtime vs MoSS

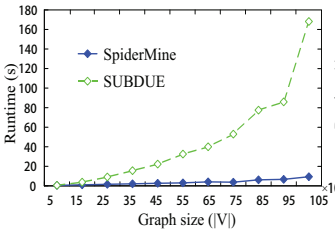

Figure 12: Runtime vs SUBDUE

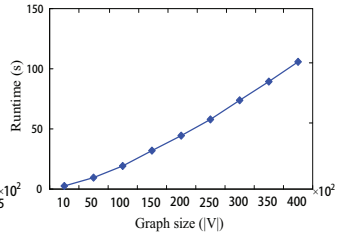

Figure 13: SpiderMine Runtime

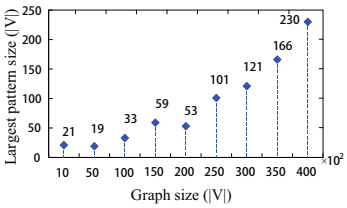

Figure 14: Largest Pattern Size (Random)

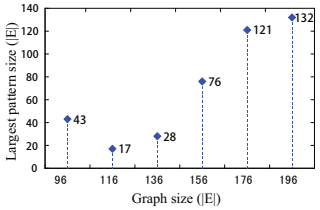

Figure 15: Largest Pattern Size (Power-Law)

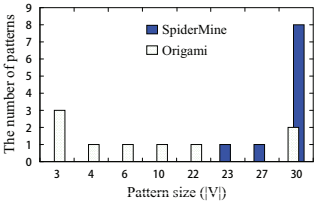

Figure 16: Fewer Small Patterns

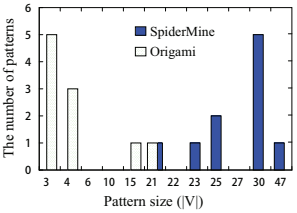

Figure 17: More Small Patterns

Run Time (seconds)				
GID	SpiderMine	SUBDUE	SEuS	MoSS
1	0.345667	0.42	0.929	1.673
2	2.279	7.98	116.982	-
3	0.883	3.18	1.539	2.641
4	13.412	17.87	968.671	-
5	4.914	9.74	2.066	-

Figure 18: Runtime Comparison

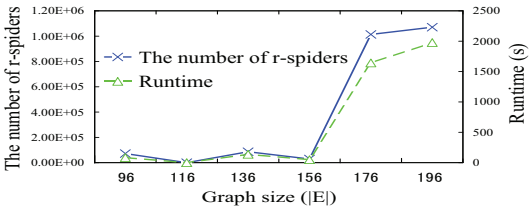

Figure 19: Scale-free Network Patterns

3. **SEuS.** Due to its node collapsing heuristics, which is less powerful in handling a large number of patterns with low frequency, SEuS has mostly generated small ($<= 3$) patterns across the five data sets. Hence we do not consider it further in our runtime comparison in Figure 12.

4. **MoSS.** MoSS aims to mine the complete pattern set, the result of which is a significantly higher runtime complexity than SpiderMine across all data sets. In Figure 18 we show the run time of the four algorithms on the five data sets. Symbol "-" means that MoSS cannot run to completion for data sets with $GID = 2, 4, 5$. We aborted a process if it could no complete within 10 hours. In Figure 11 we decrease the average node degree to 2 ($d = 2, f = 70$) for MoSS to finish execution and further compared the runtime between the two algorithms.

As it is impossible to compare running time against algorithms mining for the complete pattern set on larger data sets, in Figure 12 and Figure 13, we compare against an approximate algorithm and show our own scalability on even larger ones. For these two figures, we generate a random graph with an average degree of 3, D_{max} set to 10, and label number set to 100. The minimum support threshold is set to a low value of 2 to make available as many frequent large patterns as possible with $K = 10$. We show performance comparison with SUBDUE in Figure 12. It can be observed that as the graph size increases, the running time of SUBDUE quickly exhibits exponential growth curve while SpiderMine grows slowly. Figure 13 shows the run time of SpiderMine as the input graph size in increased up to 40000. Figure 14 shows the corresponding sizes of the largest patterns discovered. SpiderMine is able to find patterns of size 230 in data graph of size 40000 in less than two minutes.

Scale-free Network.

We use the Barabási-Albert model to generate graphs with

power-law degree distribution. The difference in degree distribution between a scale-free graph and an random graph could have tremendous impact on the total number of frequent patterns. Even for a scale-free network of a relatively small size, vertices with high degree could give rise to a huge number of small patterns. In our experiments, SUBDUE and SEuS can not run to completion on these scale-free graph data. MoSS on the other hand returns a set of small patterns. The performance of SpiderMine is shown in Figure 19. The number of r-spiders with $r = 1$, mined from such networks, increases sharply as the graph size increases. The sizes of the largest patterns discovered is shown in Figure 15.

5.1.2 Graph-Transaction Setting

As we have discussed in Section 2, the algorithm that is closest to our mining task in graph transaction setting is ORIGAMI [26] as both algorithms compute certain subset of the complete pattern set. Hence we compare with ORIGAMI in this setting. We construct a graph transaction database as follows. We use the same Erdős-Rényi model to generate 10 graphs each with 500 vertices and an average degree of 5. The number of labels is set to 65. Five *distinctive* large patterns each of 30 vertices are then injected into the graph database. Figure 16 shows the pattern distribution of the results by ORIGAMI and SpiderMine. ORIGAMI does capture some of the large patterns. However, if the data contains more small patterns as shown in Figure 17 where we injected 100 small patterns each of size 5, ORIGAMI's result leans significantly towards smaller ones, missing all the large yet equally distinctive ones. In fact, the authors of ORIGAMI mentioned in [26] that, in general, their approach favors a maximal pattern of smaller size over a maximal pattern of larger size. This shows that ORIGAMI is not designed to accomplish our task of finding the top-K largest patterns.

6. CONCLUSIONS

In this paper, we propose a novel and efficient mining framework SpiderMine for single-graph top-k large pattern mining problem based on a new concept of r-spider. We also propose a new graph pattern representation based on spiders to reduce the cost of graph isomorphism test. By assembling these spiders, we are able to efficiently discover top-K large frequent patterns with any user-specified probability. Experiments demonstrate the efficiency as well as scalability of our algorithm.

7. REFERENCES

[1] B. Berendt, A. Hotho, and G. Stumme, "Towards semantic web mining," in *International Semantic Web Conference*, 2002, pp. 264–278.

[2] J. K. J. Leskovec and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *KDD'05*, pp. 177–187.

[3] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *SIGMOD'04*, pp. 335–346.

[4] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases," in *SIGMOD '07*, pp. 857–872.

[5] C. Borgelt and M. Berthold, "Mining molecular fragments: Finding relevant substructures of molecules," in *ICDM'02*, pp. 211–218.

[6] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *ICDM'02*, pp. 721–724.

[7] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *PKDD'00*, pp. 13–23.

[8] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraph in the presence of isomorphism," in *ICDM'03*, pp. 549–552.

[9] X. Yan and J. Han, "CloseGraph: Mining closed frequent graph patterns," in *KDD'03*, pp. 286–295.

[10] J. Prins, J. Yang, J. Huan, and W. Wang, "Spin: Mining maximal frequent subgraphs from graph databases," in *KDD'04*, pp. 581–586.

[11] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," in *TKDE*, 2004, pp. 1038–1051.

[12] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008.

[13] G. Canfora and A. Cimitile, *Software Maintenance*, ser. Handbook of Software Engineering and Knowledge Engineering. World Scientific, 2002, vol. 1, pp. 91–120.

[14] E. Erlikh, "Leveraging legacy system dollars for e-business." *IEEE IT Pro*, pp. 17–23, 2000.

[15] T. Standish, "An essay on software reuse." *IEEE TSE*, pp. 494–497, 1984.

[16] C. Olston, S. Chopra, and U. Srivastava, "Generating example data for dataflow programs," in *SIGMOD '09*, pp. 245–256.

[17] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *KDD '07*, pp. 460–469.

[18] L. Holder, D. Cook, and S. Djoko, "Substructure discovery in the subdue system," in *KDD'94*, pp. 169–180.

[19] M. Kuramochi and G. Karypis, "Grew-a scalable frequent subgraph discovery algorithm," in *ICDM '04*, pp. 439–442.

[20] N. Vanetik, E. Gudes, and S. Shimony, "Computing frequent graph patterns from semistructured data," in *ICDM'02*, pp. 458–465.

[21] S. Ghazizadeh and S. S. Chawathe, "Seus: Structure extraction using summaries," in *In Proc. of the 5th International Conference on Discovery Science*, 2002, pp. 71–85.

[22] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *Data Mining and Knowledge Discovery*, no. 11, pp. 243–271, 2005.

[23] M. Fiedler and C. Borgelt, "Support computation for mining frequent subgraphs in a single graph," in *The 5th International Workshop on Mining and Learning with Graphs*, 2007.

[24] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from graphs:mining graph data," *Machine Learning*, vol. 50, no. 3, pp. 321–354, 2003.

[25] L. Thomas, S. Valluri, and K. Karlapalem, "Margin: Maximal frequent subgraph mining," in *ICDM '06*, pp. 1097–1101.

[26] M. A. Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki, "Origami: Mining representative orthogonal graph patterns," in *ICDM '07*, pp. 153–162.

[27] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *SIGMOD '08*, pp. 433–444.

[28] F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng, "Mining colossal frequent patterns by core pattern fusion," in *ICDE07*, pp. 706–715.

[29] P. Crescenzi, "On the analysis of graphs evolving over time," *http://cost295.net/docs/dagstuhl/files/crescenzi.pdf*, 2009.

[30] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, "Hadi fast diameter estimation and mining in massive graphs with hadoop," *Technical Report CMU-ML-08-117*, December 2008.

[31] I. F. M. Worlein, T. Meinl and M. Philippsen, "A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston," in *PKDD '05*, pp. 392–403.

[32] "Jeti. Version 0.7.6(Oct. 2006)." http://jeti.sourceforge.net/.

[33] M.Fowler, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[34] W. Stevens, G. Myers, and L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, pp. 115–139, 1974.

[35] F. Harary, "On the notion of balance of a signed graph." *Michigan Math. Journal*, vol. 2, no. 2, pp. 143–146, 1953.

[36] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010.

8. APPENDIX

8.1 Proof Sketch of Lemma 2

Given a pattern P, if a spider s is chosen in a random spider draw such that s is a subgraph of P, we say P is *hit*. We now analyze how to estimate the probability of a pattern P being hit by a random spider draw, which we denote as $P_{hit}(P)$. For a spider s to be a subgraph of P, s must have one of the vertices of P as its head. Thus the total number of spiders which are subgraphs of P is the number of spiders with $v \in V(P)$ as its head. Given a spider s and a vertex $v \in V(P)$, if there is at least one embedding $\hat{s}$ of s such that v is the head of $\hat{s}$, we say s is *adjacent* to v. All the spiders that are adjacent to v is denoted as $Spider(v)$. As such, we have

$$P_{hit}(P) = \frac{\sum_{v \in V(P)} |Spider(v)|}{|S_{all}|}$$

where S_{all} is the set of all spiders mined in the first stage of the algorithm. We use the average value for all the vertices to lower-bound the size of $Spider(v)$ for vertices in a large pattern P. This is based on the following argument — Large graph patterns usually contain some " hub" or " significant" vertex, which, in most cases, has a much higher degree than other average vertices in the graph. The number of spiders adjacent to these kind of vertices is accordingly much greater than that of others. As the average size of $|Spider(v)|$ for a vertex $v \in V(G)$ can be computed as $\frac{|S_{all}|}{|V(G)|}$, we have

$$P_{hit}(P) \geq \frac{\sum_{v \in V(P)} \frac{|S_{all}|}{|V(G)|}}{|S_{all}|} = \frac{|V(P)|}{|V(G)|}$$

We define $P_{fail}(P)$ as the probability that at most one spider within P is chosen in the random drawing process. As our algorithm shows, if at least two spiders within P are picked initially, we will be able to identify them at the end of the growth iterations since they must merge at some iteration. Consequently, if none or at most one spider within P is chosen, we will miss the pattern P and that is why we term it as the probability of failing to discover P.

Let M be the total number of spiders we randomly pick. When $P_{hit}(P) \leq 1/2$, we have

$$\begin{aligned} P_{fail}(P) &= (1 - P_{hit}(P))^M + M \cdot P_{hit}(P)(1 - P_{hit}(P))^{M-1} \\ &\leq (1 - P_{hit}(P))^M + M \cdot (1 - P_{hit}(P))^M \\ &\leq (M+1)(1 - P_{hit}(P))^M \end{aligned}$$

When $P_{hit}(P) > 1/2$, we have similar results as

$$P_{fail}(P) \leq M \cdot P_{hit}(P)(1 - P_{hit}(P))^{M-1} \leq M \cdot P_{hit}(P)^M$$

If more than one spider within a pattern P are chosen in the random drawing process, we say that P is successfully identified. We denote as $P_{success}$ the probability that all the top-K largest patterns are successfully identified. Let P_i be the top i-th pattern in descending order. Since the events that P_i is successfully identified are not independent as one spider could be a frequent subgraph of multiple patterns, assuming M is much larger than K and $P_{hit}(P_i) \leq 1/2, 1 \leq i \leq K$, we have

$$\begin{aligned} P_{success} &= Prob[\bigcap P_i \text{ is successfully identified}] \\ &\geq \prod_{i=1}^{K} Prob[P_i \text{ is successfully identified}] \\ &= \prod_{i=1}^{K} (1 - P_{fail}(P_i)) \\ &= \prod_{i=1}^{K} \left(1 - (M+1)(1 - P_{hit}(P_i))^M\right) \\ &\geq \prod_{i=1}^{K} \left(1 - (M+1)(1 - \frac{V_{min}}{|V(G)|})^M\right) \\ &= \left(1 - (M+1)(1 - \frac{V_{min}}{|V(G)|})^M\right)^K \end{aligned}$$

8.2 Algorithm Details

This section presents the details of our algorithm. we focus on the case for $r = 1$ for simplicity of presentation and implementation. First we define the boundary vertices for a spider. Given an r-spider s with head v, the set of *boundary vertices* of s is denoted as $B[s]$ and is defined as $B[s] = \{u | Dist(u, v) = r, u \in V(s)\}$ where $Dist(u, v)$ is the shortest distance between u and the head v. In practice, $B[s]$ is implemented as a queue in which all the boundary vertices are sorted lexicographically. In several steps of our algorithm, we need to examine each member of $B[s]$ once. As such, a spider s also has an index, $s._{pointer}$, pointing to the current boundary vertex to be examined. $s._{next}$ links to the next boundary vertex to be examined, and $s._{next}$ is $NULL$ if s is the last member in the queue $B[s]$.

Detailed pseudo-code of SpiderGrow() is given in Algorithm 2. T is a working set to store all the frequent supergraphs grown out of one pattern. In general, a pattern P would extends to multiple frequent patterns of larger sizes. The "Continue" statement, same as that in C programming, forces an immediate jump to the loop control statement. Lines 4 to 13 are to initialize T with a set of frequent patterns each obtained by growing the current pattern P with a spider extended at one of its boundary vertices. SpiderExtend() is the routine which, given a boundary vertex v and a spider s headed at v, decides if P can be extended with s at v to form a larger frequent pattern Q. If so, Q is generated with corresponding embedding list $E[Q]$, and the list of boundary vertices $B[Q]$ is adjusted accordingly. In particular, the proper portion of $B[s]$ will be inserted into $B[P]$ to form the correct $B[Q]$. Details are shown in Algorithm 3. Line 7 is to check if the new pattern Q is a redundant generation of an existing pattern. Whenever a current pattern P is extended with a new spider s, it will be checked for possible merging with other patterns by CheckMerge(). If so, the newly merged pattern will be added to the output result set. Line 10 is to initialize P's index pointer to indicate which boundary vertex to examine next. At lines 14 to 29, all frequent patterns that can be grown from P are generated by spider extension. Line 25 drops the pattern if it is found to be non-closed, i.e., Q has exactly the same embeddings as P and $P \subset Q$.

SpiderExtend() is shown in Algorithm 3. To decide if a current pattern P can be extended with a spider s at v is to check every embedding of P with s headed at v and see if

Algorithm 2 SpiderGrow

Input: freq. subgraph set S, sup. threshold σ
spider buffers Buf_{pre}, Buf_{cur}
Output: S'
1: $S' \leftarrow \emptyset$
2: **For each** $P \in S$
3: $T \leftarrow \emptyset$;
4: **For each** $v \in B[P]$
5: **For each** $s \in Spider(v)$
6: $Q \leftarrow \mathsf{SpiderExtend}(P, v, s, \sigma)$;
7: **If** $\mathsf{SpiderSetCheck}(Q)$
8: **Continue** ;/*Q is redundant.*/
9: $S' \leftarrow S' \bigcup \mathsf{CheckMerge}(Q, s, Buf_{pre}, Buf_{cur})$;
10: $P_{.pointer} \leftarrow v_{.next}$;
11: $T \leftarrow T \bigcup \{Q\}$;
12: **Do Until** no new pattern is generated
13: $P \leftarrow$ the next pattern in T;
14: **If** $P_{.ponter} = NULL$
15: **Continue**;/*$B[P]$ have been all checked*/
16: $v \leftarrow P_{.pointer}$;
17: **For each** $s \in Spider(v)$
18: $Q \leftarrow \mathsf{SpiderExtend}(P, v, s, \sigma)$;
19: **If** $\mathsf{SpiderSetCheck}(Q)$
20: **Continue** ;/*Q is redundant.*/
21: $S' \leftarrow S' \bigcup \mathsf{CheckMerge}(Q, s, Buf_{pre}, Buf_{cur})$;
22: **If** $Q_{sup} = P_{sup}$
23: Remove P from T; /*P is not closed*/
24: $T \leftarrow T \bigcup \{Q\}$;
25: $P_{.pointer} \leftarrow v_{.next}$;
26: $S' \leftarrow S' \bigcup T$;
27: $Buf_{pre} \leftarrow Buf_{cur}; Buf_{cur} \leftarrow \emptyset$;
28: **Return** S';

both the following conditions are satisfied. Only those satisfying embeddings are counted in the support. (I) *Maximal Overlap.* s contains all edges of P that are within distance 1 to v. This is to make sure that no other spider overlaps more with P than s, and (II) *Internal Integrity.* s contains no new edge e connecting two vertices of P. This is to establish the iteration invariant that each iteration we only expand the current pattern outward, leaving the internal part of P intact. Line 6 checks condition (I), and condition (II) is checked at Lines 8 to 14. The notation $e_s \bigcap e_P$ at Line 6 denotes the overlapping part of two embeddings e_s and e_P. Line 15 collects all the valid embeddings of Q.

CheckMerge() detects merging whenever a pattern P is extended with a spider s. In Algorithm 4, we keep two buffers of spider, Buf_{pre} and Buf_{cur}, to store spiders which have been used for extension in the previous iteration and the current iteration respectively. CheckMerge() returns a set of merged patterns if merging is detected and is valid, otherwise it returns $NULL$. Note that when P is extended with s, it can potentially merge with more than one other pattern. For each spider s registered in the two buffers, we also store pointers to the patterns which have used s for extension for the previous and current iterations. We denote the list of such patterns as $Buf_{pre}[s]$ and $Buf_{cur}[s]$.

8.3 Extended Experimental Results

8.3.1 Synthetic Data

Algorithm 3 SpiderExtend

Input: pattern P, boundary v, spider s, sup. threshold σ
Output: Q
1: Compute $N_v^P = \{e | e$ is an edge of e_P adjacent to $v\}$;
2: **If** $N_v^P \not\subset E[s]$
3: **Return** P;
4: **For each** $e_p \in E[P]$
5: **For each** $e_s \in E[s]$
6: **If** $N_v^P \neq E(e_s \bigcap e_P)$
7: **Continue**;
8: $T \leftarrow E(e_s) \setminus E(e_p)$;
9: **For each** $(u_1, u_2) \in T$
10: **If** $u_1 \in V(P)$ **or** $u_2 \in V(P)$
11: $FAIL \leftarrow TRUE$;
12: **If** $FAIL = TRUE$
13: **Continue** ;
14: $Q \leftarrow P \bigcup (E(s) \setminus N_v^P)$;
15: $E[Q] \leftarrow E[Q] \bigcup \{e_p \bigcup e_s\}$;
16: **If** $|E[Q]| < \sigma$
17: **Return** P;
18: Update $B[Q]$;
19: **Return** Q;

Algorithm 4 CheckMerge

Input: pattern P, spider s, sup. threshold σ
Buf_{pre} and Buf_{cur}
Output: S
1: $S \leftarrow \emptyset$;
2: **If** $s \in Buf_{pre}$ **or** $s \in Buf_{cur}$
3: $T \leftarrow Buf_{pre}[s] \bigcup Buf_{cur}[s]$;
5: **For each** $P' \in T$
6: **If** P and P' can be merged
7: $Q \leftarrow P \bigcup P'$
8: Update $B[Q]$;
9: $S \leftarrow S \bigcup \{Q\}$;
10: **Else**
11: Register s and P in Buf_{cur};
12: **Return** S;

(1) Varied Pattern Distributions. As shown in Figure 26, 5 datasets (GID 6 to 10) are designed with increasing proportion of small patterns. We show in Figure 20 that SpiderMine is fairly robust against varied pattern distributions. The results returned by SpiderMine, the top 5 largest patterns mined sorted in size-decreasing order, is quite consistent. The outlier of GID 9 is due to the incidental overlapping of two injected large patterns resulting in one double-sized, which in fact demonstrates our mining effectiveness. Here D_{max} is set to 6 with $\sigma = 10$ and $K = 5$.

(2) Varied D_{max}. In Figure 21 we show the top 5 largest patterns mined sorted in size-decreasing order with varied D_{max}. In the figure, d stands for $\frac{D_{max}}{2}$. As illustrated, in general, SpiderMine is robust with respect to varied D_{max} unless D_{max} is too small, which happens because even if multiple spiders are picked within a large pattern, the number of iterations is then too small for them to grow closer to merge if they are fairly apart initially. The data setting here is the same as GID 7.

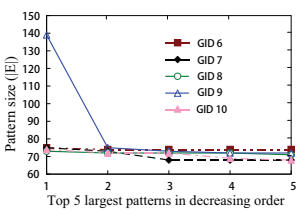

Figure 20: Robustness Comparison

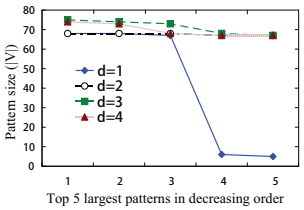

Figure 21: D_{max} Estimation

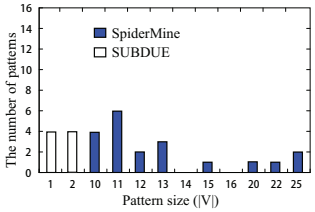

Figure 22: Pattern Distribution For DBLP

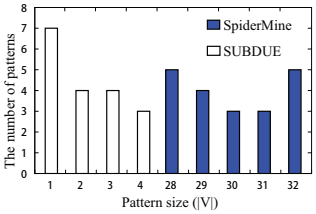

Figure 23: Pattern Distribution For Jeti

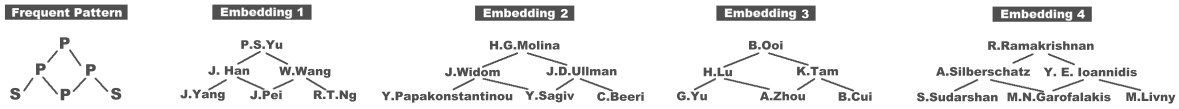

Figure 24: DBLP Example (partial): A Collaborative Pattern Common To Different Research Groups

(3) Varied r. The value of r mainly affects the running time of the first stage of mining all the r-spiders, which grows exponentially with the increase of r. For example, on one data graph of 600 edges and 30 labels, the runtime is 610 milliseconds when r=1, 2696 milliseconds when r=2, 86696 milliseconds when r=3 and when r=4, the memory ran out before termination. On the other hand, larger spiders give faster generation of large patterns in the second stage. Empirically, we find that a small r, e.g., $r = 1$ or $r = 2$, is a good trade-off choice which gives better overall mining performance because the quality of the top-K mining result is largely unaffected as a result of our probabilistic framework.

(4) Varied ϵ. In general, smaller ϵ leads to longer runtime as more seed spiders are drawn for growth. However, the exact rate of efficiency degradation depends on a number of factors of the input data including the distribution and the level of prominence of the large patterns, the overall distribution of closed patterns of various sizes and so on. For example, on Jeti data, when minimum support = 10, the runtime is 7.198s when $\epsilon = 0.45$, 7.725s when $\epsilon = 0.25$ and 9.103s when $\epsilon = 0.05$.

8.3.2 Real Data

We demonstrate with two real data sets the application of large patterns in social network analysis and software engineering.

DBLP.

DBLP (http://www.informatik. uni-trier.de/~ley/db/) provides bibliographic information on major computer science journals and proceedings. DBLP data contains more than 955000 papers from 418139 distinct authors and 2687 conferences. We selected 600 top conferences covering nine major computer science areas. Each author in DBLP will be assigned to exactly one of the nine areas, which is the one in which he or she publishes the most papers. We then pick all the 15071 authors assigned to the "Database and Data Mining" area. We constructed from DBLP data a co-author relationship graph in which each vertex is a distinct author. We give labels to the authors as follows: An author is assigned a label "Prolific(P)" if the author published at least 50 papers in the area of Database and Data Mining. The "Senior(S)" label is assigned to authors with 20 to 49 papers; "Junior(J)" with 10 to 19 papers and "Beginner(B)" with 5 to 9 papers. The authors with less than 5 papers are not considered. We are left with 6762 authors, each represented by a vertex in the co-author relationship graph G. There is an edge between two authors v and u if the number of papers they co-authored exceeds λ% of one author's total number of publications where λ is determined by the type of the authors. Details are omitted here due to space limit. With these definitions, we obtain an co-author relationship graph G with 6508 vertices and 24402 edges.

With the minimum frequency support set to be 4 and $K = 20$, SpiderMine returned 20 large patterns with the largest of size 25. Comparison with SUBDUE is shown in Figure 22. A frequent pattern in the co-author relationship graph describes pair-wise co-author relationship. It does not necessarily imply that all authors in a particular embedding have co-authored the same paper at one time. Rather, it reveals a collective behaviorial model of a community of authors. Due to the small number of node attribute values, small patterns are almost ubiquitous, therefore offering little analytical power. On the other hand, large patterns provide insight into (I) finding collaborative patterns common to different research groups, as shown in Figure 24; and (II) identifying clusters of researchers based on the discriminative collaborative patterns unique to a particular group as shown in Figure 25. In Figure 25, the main pattern, which is present in all embeddings, is shown in solid lines while the pattern variant, extra edges each appearing in some embeddings, is indicated by dotted blue lines. As we found out, a discriminative large pattern, together with its variants which only differ slightly, have all their embeddings (indicated in Figure 25 by the total number of embeddings) clustered on a similar group of researchers, thus distinguishing clusters of researchers with different collaborative patterns. Note that in Figure 25, not all the author names are shown due to space limit.

Jeti.

We analyze Jeti [32], a popular full featured open source instant messaging application based on the Jabber (XMPP) open standard for Instant Messaging and Presence technology. Jeti has an open plug-in architecture and supports many chat features including file transfer, group chat, picture chat (whiteboard group drawing), buddy lists, dynamic presence indicators, etc. The application has about 49,000

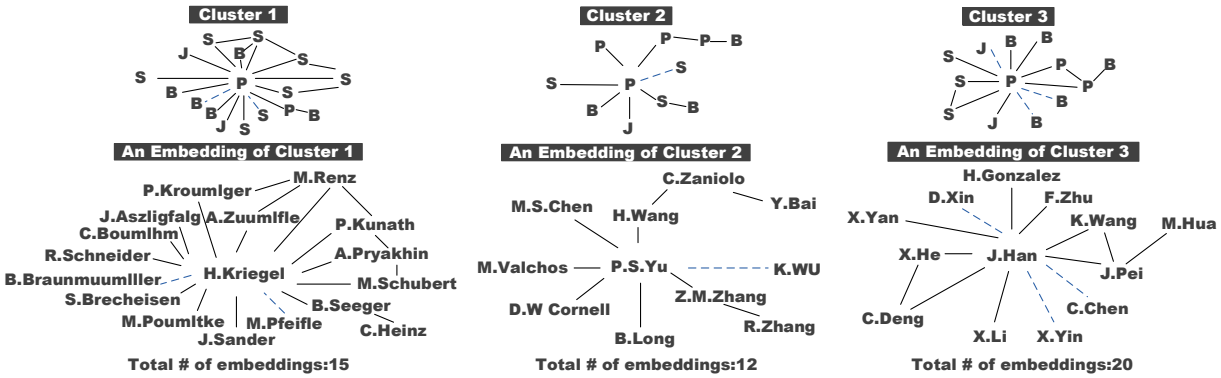

Figure 25: DBLP Example: Discriminative Collaborative Patterns Distinguishing Different Research Groups

GID	GID=6	GID=7	GID=8	GID=9	GID=10
$(\|V\|, \|E\|, f)$	(20490,31255,1064)	(31110,47446,1658)	(37595,57262,2062)	(47410,72149,2610)	(56740,86330,3138)
large patterns injected	5 large patterns injected each with 50 vertices, support is varied between 10 to 15				
small patterns injected	50 small patterns injected each with 5 vertices				
	support 5-15	support 10-20	support 15-25	support 20-30	support 25-35

Figure 26: GID 6 to 10

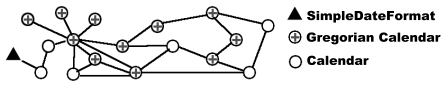

Figure 27: Jeti Pattern Example (partial pattern)

lines of code, comments, and blank lines. We extracted a graph from Jeti, where the nodes correspond to methods and the edges correspond to calling relationships between the methods. Each node is labeled with the class the corresponding method belongs to. There is an edge from node A to node B, if method A potentially calls method B in the Jeti's and relevant Java API's source code. The graph has in total 835 nodes, 1,764 edges and 267 labels. The nodes have an average degree of 2.13 and a maximum degree of 69. In Figure 23, we show the patterns mined by SpiderMine and SUBDUE. Minimum support is set to 10. MoSS and SEuS can not return result with hours of running on this data.

A portion of one of the patterns with the highest support is shown in Figure 27. There are 3 node labels corresponding to 3 classes namely: java.util.GregorianCalendar, java.util.Calendar, and java.text.SimpleDateFormat. Each node in the pattern corresponds to a method from either GregorianCalendar, Calendar, or SimpleDateFormat class. Links between the nodes correspond to the call relationships between the methods (i.e., one method calls another). The pattern shows a tight communication among the various methods in the GregorianCalendar class and Calendar class. Mined patterns like the above could help in identifying design smells (see [33]). Some classes should have a high cohesion (i.e., a measure of how strongly-related classes are), while others should have low coupling (i.e., a measure of how loosely-related classes are) [34]. A class that is a subclass of another class should have high cohesion, e.g., the GregorianCalendar class and the Calendar class. On the other hand two unrelated classes should not have too much cohesion. Indeed it should have low coupling.

8.4 Discussion & Future Work

In this section, we illustrate some potential use of large network patterns in software engineering and social network study, as well as some future work.

Software Engineering. In software engineering, large patterns could be used to capture the "backbone" of a family of systems. Many software houses release various variants of the same product to customize based on individual client needs. Many of them share many similarities which are the backbone of the systems. These backbones are typically quite large and involve invocations of a series of methods spread across various components.

Mining software backbone would be very useful especially for understanding legacy system which is usually termed as program comprehension. Program comprehension has been estimated to cost up to 50% of software maintenance cost. Software maintenance cost has been estimated to go as high as 90% of the total software cost [13, 14, 15]. When the original developers leave the team, new developers would need to learn the existing system often without sufficient documentation, c.f. [16, 17].

Social Network. Most large patterns are infrequent. Therefore, it is particularly interesting when a frequent one is indeed found in social networks. These patterns represent massive interactive patterns that could shed light into internal dynamics of social networks. These patterns in turn could be used to "groom" existing networks by finding "stable" patterns that could inspire and guide social network service providers to promote activities that would encourage more collaborations among "citizens" of the networks and reduce the churn rate of the network (c.f., balance theorem in social network [35, 36]).

In our future work, we would seek to lift the constraint on the pattern diameter by designing new algorithms tailored for patterns with long diameters. A more rigorous complexity analysis of SpiderMine is also on the agenda. We would also explore further applications based on large patterns mined from a single large network.