SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation

Peng Jiang
The University of Iowa
Iowa City, Iowa, USA
peng-jiang@uiowa.edu

Yihua Wei
The University of Iowa
Iowa City, Iowa, USA
yihua-wei@uiowa.edu

Jiya Su
Illinois Institute of Technology
Chicago, Illinois, USA
jsu18@hawk.iit.edu

Rujia Wang
Illinois Institute of Technology
Chicago, Illinois, USA
rwang67@iit.edu

Bo Wu
Colorado School of Mines
Golden, Colorado, USA
bwu@mines.edu

ABSTRACT

Subgraph Pattern Mining (SPM) is an important class of graph applications that aim to discover structural patterns in a graph. Due to the enormous exploration space, SPM is in general computationally challenging. To accelerate SPM, many random sampling techniques have been proposed. While the existing sampling techniques are effective for conventional SPM tasks such as motif counting and frequent subgraph mining, they cannot be easily adapted for new applications.

In this work, we propose SampleMine, a framework for applying random sampling to any non-listing SPM task. Our main idea is to express subgraph exploration as a nested loop and sample the subgraphs with loop perforation. We first propose a two-vertex exploration technique to accelerate the subgraph exploration procedure. Then, we provide two sampling strategies under the loop perforation framework and show that they can achieve good results for counting and frequent subgraph mining tasks. The experimental results show that our system achieves significant speedups against the state-of-the-art graph mining systems with little accuracy loss.

CCS CONCEPTS

• **Mathematics of computing → Approximation algorithms**; **Graph algorithms**.

KEYWORDS

Subgraph Pattern Mining, Loop Perforation

ACM Reference Format:

Peng Jiang, Yihua Wei, Jiya Su, Rujia Wang, and Bo Wu. 2022. SampleMine: A Framework for Applying Random Sampling to Subgraph Pattern Mining through Loop Perforation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569658

PACT '22, October 8–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569658

1 INTRODUCTION

Subgraph Pattern Mining (SPM) is widely used for retrieving information from graph-structured data in various application domains, including bioinformatics [34, 45], computer vision [16], and social network analysis [44]. An example SPM task is frequent subgraph mining, which is to discover subgraph patterns with supports larger than a threshold. Another example query could be 'counting the subgraphs with at least one vertex of label x'.

Many algorithms and systems have been proposed for different SPM tasks such as motif counting [5, 7, 8, 22, 36, 39, 46, 48] and frequent subgraph mining [3, 6, 18, 20, 27, 37, 38]. These works usually include task-specific optimizations and cannot be easily adopted for new applications. In recent years, there is a growing interest in designing general-purpose SPM systems [17, 24, 31, 43, 47]. These systems aim to cover different SPM tasks with a general computation pattern and provide a flexible API to the users. However, due the enormous exploration space, the existing systems have difficulty in mining large patterns on large graphs.

Random sampling is a commonly used technique to reduce the computational complexity of SPM tasks that do not require an exhaustive listing of subgraphs [3, 9, 11, 19, 37, 38]. Most of the existing sampling methods are task-specific. For instance, graph coloring has proven to be effective for unlabeled motif counting [9], while support estimation is more suitable for frequent subgraph mining [37]. How to enable efficient sampling for an arbitrary mining task is still an open problem. Previous SPM systems have adopted neighbor sampling [23, 32, 35], but they only work for counting tasks and have limited support for user-defined queries.

In this work, we aim to provide a general framework for applying random sampling to various SPM tasks. Our solution is based on the observation that the subgraph exploration procedure can be expressed as a nested loop and the sampling of subgraphs can be achieved by simply perforating the loops. For subgraph exploration, we propose a novel two-vertex approach. The idea is to extend a subgraph by joining it with its neighboring size-3 subgraphs on a common vertex in each exploration step. We prove that two-vertex exploration can discover all subgraphs of any size and show that it has smaller time complexity than the traditional single-vertex exploration.

For subgraph sampling, we provide two loop perforation strategies and show that they can achieve good estimation results for counting and frequent subgraph mining tasks. Compared with the neighbor sampling technique in the existing SPM systems [23, 32], our sampling strategy can quick find the most important patterns and return more accurate counts with the same amount of computation. For frequent subgraph mining, our sampling strategy can discover most of the frequent patterns with only a small portion of the execution time of accurate mining. Besides subgraph counting and frequent subgraph mining, our sampling technique can be easily applied to non-conventional, user-defined mining tasks. Our system also provides a simple API for the users to define their own sampling strategies.

In summary, we make the following contributions in this paper.

- We propose a two-vertex approach for subgraph exploration and show its advantage over single-vertex exploration.
- We propose a loop-perforation-based sampling framework for arbitrary SPM tasks.
- We provide two sampling strategies under the loop perforation framework and show that they can achieve good performance and accuracy for various SPM tasks.

We perform extensive evaluation of our system and compare with five state-of-the-art graph mining systems: AutoMine [31], Peregrine [24], Pangolin [15], ASAP [23], and ScaleMine [3]. For subgraph counting, our system achieves an average (geometric mean) of 16.3x speedup against AutoMine, and 129.7x speedup against Peregrine, with little accuracy loss. For frequent subgraph mining, our system achieves an average of 34.7x speedup against AutMine, 8.3x speedup against Peregrine, 10.7x speedup against Pangolin, and 18.7x speedup against ScaleMine, while returning more than 80% of the frequent patterns. Our system can also return good estimation results for tasks that the existing systems do not support or cannot return in a reasonable amount of time.

2 BACKGROUND

This section gives some background on SPM and the existing systems for SPM.

2.1 Graph Basics

A *graph* G is defined as $G = (V, E, L)$ consisting of a set of vertices V, a set of edges E and a labeling function L that assigns labels to the vertices and edges. A graph $G' = (V', E', L')$ is a *subgraph* of graph $G = (V, E, L)$ if $V' \subseteq V$, $E' \subseteq E$ and $L'(v) = L(v), \forall v \in V'$. A subgraph $G' = (V', E', L')$ is *vertex-induced* if all the edges in E that connect the vertices in V' are included E'. A subgraph is *edge-induced* if it is connected and is not vertex-induced.

Definition 1 (Isomorphism). Two graphs $G_a = (V_a, E_a, L_a)$ and $G_b = (V_b, E_b, L_b)$ are isomorphic if there is a bijective function $f : V_a \Rightarrow V_b$ such that $(v_i, v_j) \in E_a$ if and only if $(f(v_i), f(v_j)) \in E_b$.

We say two (sub)graphs have the same pattern if they are isomorphic. The pattern is a template for the isomorphic subgraphs, and a subgraph is an instance (also called embedding) of its pattern. To determine the pattern of a subgraph, a canonical form of the subgraph can be computed. The subgraphs with the same canonical form are isomorphic. There are various tools and algorithms available for graph isomorphism check [25, 33, 49]. All of these algorithms have exponential complexity. We use *bliss* [25] for isomorphism check in our system as it is fast in practice and is widely used in the existing systems [24, 43, 47]. Isomorphisms from a graph to itself are called *automorphisms*.

2.2 Subgraph Pattern Mining Tasks

Our system supports conventional SPM tasks as well as user-defined tasks. Some examples are:

• *Subgraph Counting (SC).* The task is to count the embeddings of different subgraph patterns and find the patterns with the largest counts.

• *Frequent Subgraph Mining (FSM).* The task is to obtain all frequent subgraph patterns from a labeled input graph. A pattern is considered frequent if it has a *support* above a threshold. Different from the counts, the support of a pattern usually needs to have the *anti-monotone* property, i.e., the support of a pattern can not be larger than the support of it subpatterns. The most commonly used support measure for FSM is the minimum image based (MNI) support [10].

Definition 2 (MNI Support). Given a pattern $P = (V_p, E_p, L_p)$ and an input graph $G = (V, E, L)$, if P has m embeddings $\{f_1, f_2, \dots, f_m\}$ in G, the minimum image based (MNI) support of P in G is defined as

$$\sigma_{MNI}(P, G) = \min_{v \in V_p} |\{f_i(v) : i = 1, 2, \dots, m\}|.$$

The set of nodes in V that are assigned to $v \in V_p$, i.e, $\{f_i(v) : i = 1, 2, \dots, m\}$ is the domain of v on G, denoted as $Dom(v, G)$. With a support measure σ, the frequent subgraph mining problem is defined as finding all patterns $\{P_i = (V_i, E_i, L_i)\}$ in a graph G such that $|V_i| = s$ and $\sigma(P_i, G) \geq t$ where s is the given pattern size and t is the given support threshold. Depending on the applications, the users may require listing the subgraphs of the frequent patterns. We focus on finding the frequent patterns in this work. With the frequent patterns, the frequent subgraphs can be easily obtained with any graph pattern matching procedure.

• *User-Defined Queries.* In addition to the conventional SPM tasks, we may be interested in finding subgraphs that meet certain constraints. For example, one might be interested in 'the number of size-7 subgraphs that have at least two vertices with label x'. Another query example is 'finding all the frequent subgraphs that have at least one vertex with label x or label y'.

2.3 Systems for Subgraph Pattern Mining

There are mainly two approaches to generic subgraph mining taken by the existing systems. Some SPM systems are *subgraph-centric* [15, 17, 43, 47]. They enumerate all the subgraphs and filter out the unwanted subgraphs. The enumeration is performed in a vertex-by-vertex manner. The subgraphs of size l are extended with one vertex in each step to obtain subgraphs of size l+1. The other systems take a *pattern-based* approach [3, 18, 24, 31]. The idea is to enumerate the patterns, filter out the unwanted patterns, and match the remaining patterns on input graph. The pattern-based systems are efficient for small-pattern queries because there are not many small patterns and they can exploit the well-optimized pattern matching techniques. However, when the query pattern is large, enumerating the patterns becomes expensive as the number of patterns grows exponentially. To accelerate generic subgraph mining, many task-independent optimizations have been proposed.

For example, Fractal [17] proposes a hierarchical work-stealing mechanism to achieve better load balance for parallel subgraph enumeration. AutoMine [31] searches the matching orders of vertices to achieve the most pruning of exploration space.

3 NESTED LOOP FOR SUBGRAPH EXPLORATION

Our system takes the subgraph-centric approach. The most important task of a subgraph-centric system is to enumerate all subgraphs of size n without knowing the exact patterns. All the existing SPM systems use single-vertex exploration for this task. We find that limiting the step size to one is not necessary. In this section, we propose a two-vertex exploration technique and show its advantage over single-vertex exploration.

3.1 Two-Vertex Exploration

Figure 1a shows the procedure of single-vertex exploration adopted by the existing SPM systems. The exploration starts from all edges in the graph. In each loop level, the subgraphs are extended by their neighboring edges. The extension continues until the desired size is reached. Single-vertex exploration ensures that all the size-n subgraphs can be discovered in the innermost loop.

Figure 1b illustrates the idea of our two-vertex exploration. We first obtain all the size-3 subgraphs in the input graph. Then, the exploration starts from the size-3 subgraphs. It extends a subgraph by joining it with its neighboring size-3 subgraphs in each loop level. We call it two-vertex exploration because the subgraph size is increased by two in each join step. We find that two-vertex exploration also ensures the exhaustive enumeration of subgraphs.

Theorem 1. *For any $n > 3$, all of the size-n subgraphs can be discovered by joining the size-$(n-2)$ subgraphs with the size-3 subgraphs on a common vertex.*

Proof. To prove Theorem 1, we only need to show that any size-n subgraph can be dissected into a connected size-$(n-2)$ subgraph and a connected size-3 subgraph on one vertex. Because the size-$(n-2)$ and size-3 subgraphs are joined in all possible ways, if a dissection exists for a size-n subgraph, it will be discovered by the join operation. Suppose any size-n subgraph can be dissected into a size-$(n-2)$ and a size-3 subgraph. There are only two way a size-$(n+1)$ subgraph can be constructed from a size-n subgraph: 1) the new vertex is connected with the size-$(n-2)$ subgraph, and in this case, the size-$(n+1)$ subgraph can be dissected in the same way as the size-n subgraph (Figure 2a); 2) if the new vertex is only connected with the size-3 subgraph, we can always pick three connected vertices as the new dissection (Figure 2bcd). As the base case, all the six size-4 patterns can be dissected into a size-3 subgraph and an edge. The proof finishes by induction. □

Note that subgraph exploration is different from subgraph enumeration for a given pattern considered in previous works [28, 29]. These works propose to decompose the pattern and join the embeddings of the smaller subpatterns to find the embeddings of the original pattern. These pattern decomposition methods do not work for subgraph exploration because the patterns are unknown. In fact, for subgraph exploration, the step size cannot be larger than two. As an example, the graph in Figure 3 cannot be discovered by three-vertex exploration because it cannot be obtained by joining two connected size-4 subgraphs on a common vertex. For subgraph exploration, previous works have also proposed to merge smaller patterns to explore larger ones [26, 27]; however, their step size is one (e.g., FSG [26] joins two size-k subgraphs to obtain a size-$(k+1)$ subgraph).

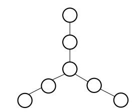


Figure 3: A graph that cannot be discovered by three-vertex exploration.

Two-vertex exploration can be either vertex-induced or edges induced. For vertex-induced exploration, we add all the connecting edges between the two joining subgraphs to the resulting subgraph. For edge-induced exploration, we enumerate all possible combinations of the connecting edges between the joining subgraphs and generate a resulting subgraph for each combination.

Compared to single-vertex exploration, two-vertex exploration has smaller time complexity for the same subgraph mining task. Intuitively, because the combination of two edges is precomputed and stored in the size-3 subgraphs, two-vertex exploration needs fewer join operations to obtain subgraphs of a certain size. More rigorously, suppose the maximum degree of the input graph is d, the maximum number of size-3 subgraphs associated with a vertex is D, the number of vertices is N, and the subgraph size is n. The time complexity of the nested loop in Figure 1a is $O((1\cdot 2\cdot 3\cdots(n-1))Nd^{(n-1)}))$. The time complexity of the nested loop for two-vertex exploration in Figure 1b is $O((1\cdot 3\cdot 5\cdots(n-2))\cdot ND^{(n-1)/2})$ if n is odd and $O((1\cdot 3\cdot 5\cdots(n-1))\cdot ND^{(n-2)/2}d)$ if n is even. Because $D < d^2$ and $2i-1 < i(i+1), \forall i \geq 1$, the time complexity of two-vertex exploration is smaller than that of single-vertex exploration.

3.2 Avoiding Redundant Subgraphs

The exploration procedure in Figure 1 can produce redundant subgraphs. As a simple example, a size-3 subgraph composed of two edges (a, b) and (b, c) can be discovered twice. It is discovered when e_1 is assigned to (a, b) and e_2 is assigned to (b, c). It is discovered again when e_1 is assigned to (b, c) and e_2 is assigned to (a, b). In many cases, we want to eliminate the redundant subgraphs. Previous systems have adopted a canonicality checking technique for redundancy removal [43]. This canonicality check, however, does not work for two-vertex exploration. We propose a *smallest-vertex-first* dissection method to achieve redundancy removal for two-vertex exploration.

Our method is based on the following observation: for any subgraph, there is only one way to divide it into two smaller subgraphs with both subgraphs being connected and one of them having the smallest spanning vertex indices. Thus, we can eliminate redundancy by finding this unique dissection of $\{\{s\}, t\}$ and checking if the dissected subgraphs are the same as s and t. The checking is performed each time we extend s with its neighboring size-3 subgraph t (i.e., in the *is_valid* function in Figure 1b). The procedure is shown in Algorithm 1. For a pair of subgraphs $\{\{s\}, t\}$, we first check if there are any other identical vertices except for the joining vertex. If yes, s and t cannot form a valid subgraph, and

```
// iterate over all edges in the graph
for e_1 ∈ E
  s_2 = {e_1};
  if (is_valid(s_2) == false) continue;
  // iterate over all neighboring nodes of s_1
  for e_2 ∈ N_2(s_2)
    // join s_2 with its neighbor edge
    s_3 = {{s_2}, e_2};
    // check if the intermediate subgraph is valid
    if (is_valid(s_3) == false) continue;
      ⋮
       for e_{n-1} ∈ N_2(s_{n-1})
         s_n = {{s_{n-1}}, e_{n-1}};
         if (is_valid(s_n) == false) continue;
```

(a) Single-vertex exploration. The first loop iterates over all edges. The following loops iterate over neighboring edges of intermediate subgraph. $N_2(s)$ represents the edges that have one and only one common vertex with s.

```
// iterate over all size-3 subgraphs
for t_1 ∈ S_3
  s_3 = {t_1};
  if (is_valid(s_3) == false) continue;
  // iterate over all neighboring size-3 subgraphs of s_3
  for t_2 ∈ N_3(s_3)
    s_5 = {{s_3}, t_2};
    // check if the intermediate subgraph is valid
    if (is_valid(s_5) == false) continue;
      ⋮
   // if n is odd, last loop iterates over size-3 subgraphs
   // if n is even, last loop iterates over edges
        for t_{(n-1)/2} ∈ N_3(s_{n-2})
          s_n = {{s_{n-2}}, t_{n-1/2}};
          if (is_valid(s_n) == false) continue;
```

(b) Two-vertex exploration. The first loop iterates over all size-3 subgraphs. The following loops iterate over neighboring size-3 subgraphs of intermediate subgraph. $N_3(s)$ represents size-3 subgraphs that have one and only one common vertex with s.

Figure 1: Subgraph exploration implemented as a nested loop.

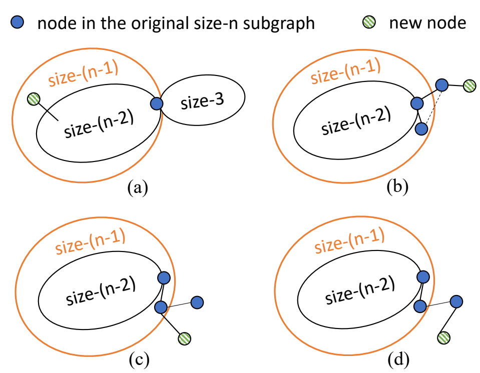


Figure 2: Two-vertex exploration can discover all subgraphs. Suppose all size-n subgraphs can be obtained by joining size-$(n-2)$ with size-3 subgraphs. For any size-$(n+1)$ subgraph, the new node is either connected to the size-$(n-2)$ part as in (a), or connected to the size-3 part in three different ways as in (b,c,d). Each case has a valid dissection into a size-$(n-1)$ and a size-3 subgraph. Thus, all size-$(n+1)$ subgraphs can be discovered by two-vertex exploration.

Algorithm 1: Automorphism check for two-vertex exploration.

```
Input  : subgraph s; subgraph t; joining vertex k
Output : combined subgraph s'
func dissect(s'):
    foreach v in s' in ascending order do
        l = the first three vertices visited by starting from v and
          spanning to the smallest vertex at each step;
        r' = the unvisited vertices in s';
        foreach v' in l in ascending order do
            r = r' ∪ v';
            if r is connected then return l, r ;
if s and t have same vertices other than k then return false;
// s' is a valid subgraph joined by s and t
s' = s ∪ t;
// find the smallest dissection of s'
l, r = dissect(s');
// if the two joining subgraphs correspond to the smallest
   dissection, return s'
if l == t and r == s then return true;
else return false;
```

the function returns false. If no, we give the combined subgraph to a dissection procedure that divides the subgraph into two small subgraphs l and r. From the vertex with the smallest index, the dissection procedure finds three connected vertices with the smallest indices and stores them in l. Next, the algorithm checks if the remaining vertices can constitute a connected subgraph r with any of the vertices in l. If yes, the dissection procedure stops and returns l and r. The algorithm returns as soon as the first dissection is found, and it will always return because of Theorem 1. Once we have the smallest dissection l and r, we check if they are the same as t and s. If yes, the function returns the true, indicating that the combined subgraph $\{\{s\}, t\}$ is valid and is not automorphic to any previously discovered subgraph.

A Running Example: Figure 4 shows a graph with all of its size-3 subgraphs (including wedges and triangles). Let us consider the size-5 subgraph '34257'. Without redundancy removal, the nested loop in Figure 1b will discover the subgraph multiple times (when $t_1 = (3, 7, 2)$ and $t_2 = (5, 4, 3)$, $t_1 = (3, 4, 2)$ and $t_2 = (3, 7, 5)$, and $t_1 = (3, 5, 2)$ and $t_2 = (3, 7, 4)$). A straightforward way to avoid redundancy is to store all the subgraphs in a no-duplicate set. This method, however, is inefficient due to the large lookup overhead, especially when the loop is executed in parallel. With our smallest-vertex-first dissection method, we can eliminate redundancy without explicitly storing the subgraphs. In this example, the smallest vertex of subgraph '34257' is 2, so the dissection procedure starts

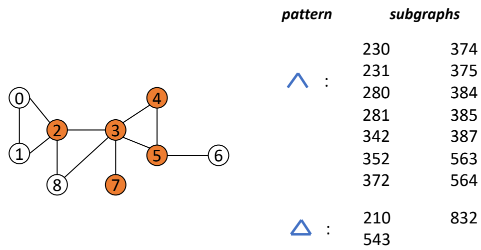


Figure 4: An example graph with its size-3 subgraphs. The size-5 subgraph '34257' can be discovered multiple times by the nested loop in Figure 1b with different t_1 and t_2.

from vertex 2 (line 2 in Algorithm 1). Then, It finds the neighbor of 2 in '34257' with the smallest index, which is 3. Then, it spans vertex 2 and 3 to their smallest index neighbors. The vertices adjacent to 2 and 3 in the subgraph are $4, 5, 7$. Because 4 is the smallest, 4 is added to l in the next step. This gives us three vertices $2, 3, 4$ in l (line 3). The unvisited vertices are 5 and 7 (line 4). We check if any of $2, 3, 4$ can form a connected graph with $5, 7$ (line 5). Since 3 is the smallest vertex that connects 5 and 7, the dissection procedure stops and returns $l = \{2, 3, 4\}$ and $r = \{3, 5, 7\}$ (line 7). By checking if the joining subgraphs match the dissection (line 11), the algorithm ensures that the subgraph '34257' is generated only once.

4 APPROXIMATE SUBGRAPH MINING WITH LOOP PERFORATION

To further accelerate subgraph pattern mining, we apply random sampling to the subgraph exploration procedure. The idea is simply executing a subset of the iterations of each loop in Figure 1. This technique is also known as loop perforation and is widely used for approximate computing [30, 41]. Compared with the existing task-specific sampling techniques, loop perforation is general – it can be applied to the nested loop for any arbitrary SPM task. The main challenge is how to perforate the loops in an efficient way so that important subgraphs can be obtained and small error bounds can be achieved. We now present two sampling strategies under this framework and show that they can achieve good performance for counting and frequent subgraph mining tasks.

4.1 Sampling for Counting Tasks

In many applications, we are interested in counting the subgraphs that meet certain constraints. For such counting tasks, a straightforward sampling strategy is to sample a fixed proportion of the iterations in each loop level. This *proportional sampling* strategy achieves a uniform sampling of the entire outcome space. To see this, we can consider the probability of executing one iteration of the innermost loop in Figure 1a. Suppose the loop has $n-1$ levels for exploring subgraphs of n vertices and the explored edges from the outermost to the innermost loop are $e_1, e_2, \ldots, e_{n-1}$. The probability of executing one innermost iteration is

$$\Pr[e_1, e_2, \ldots, e_{n-1}] = \Pr[e_1] \Pr[e_2|e_1] \ldots \Pr[e_n|e_1, e_2, \ldots, e_{n-2}]$$

where $\Pr[e_l|e_1, e_2, \ldots, e_{l-1}]$ is the probability of e_l being sampled in the lth loop. Proportional sampling ensures that this probability is a fixed number in each loop level, and thus, $\Pr[e_1, e_2, \ldots, e_{n-1}]$ is the same for all execution paths. The argument also applies to two-vertex exploration in Figure 1b. This sampling strategy is most suitable for large motif counting tasks where exhaustive enumeration of the patterns is infeasible. Since all subgraphs in the outcome space have the same sampling probability, we can quickly obtain subgraphs of the patterns that have the most embeddings.

Figure 5c shows an example of proportional sampling. For simple illustration, we consider the discovery of size-3 subgraphs by single-vertex exploration. The nested loop has two levels. The first loop iterates over all edges in the graph, and the second loop iterates over the neighboring edges of the first edge. Suppose we set the sampling probability of the first edge to $1/3$ and the probability of the second edge to 1. Let us consider the sampling of subgraphs of the two patterns in Figure 5a. For $Pattern1$, node-0 has three edges and one of them (edge $(0, 1)$) is sampled in the first loop. In the second loop, because the sampling probability is 1, both the neighbors of node-1 are sampled. This gives us two sampled subgraphs ('014' and '015'). For $Pattern2$, node-4,5,6 have six edges in total and one third of them (edge $(4, 2)$ and $(5, 3)$) are sampled in the first loop. In the second loop, $(2, 6)$ is the neighboring edge of $(4, 2)$, and $(3, 6)$ is the neighboring edge of $(5, 3)$. Both of them are sampled, and we obtain two sampled subgraphs ('426' and '536'). We can see that the number of sampled subgraphs of a pattern is in expectation proportional to its subgraph count as shown in Figure 5b.

For any user-defined counting task, suppose there are m sampled subgraphs that meet the query constraints and the sampling probability of each subgraph is p. We can obtain an unbiased estimation of the total number of inquired subgraphs as m/p. For the example in Figure 5c, the estimated number of subgraphs of both $Pattern1$ and $Pattern2$ is $2/(1/3) = 6$, which is the same as the actual count. More generally, let us denote all subgraphs that meet the query constraints as Q, all subgraphs in the outcome space as O, and the sampled subgraphs that meet the query constraints as S. With any sampling method, the size of Q can be estimated

$$\hat{c} = \sum_{s \in S} \frac{1}{p_s} \tag{1}$$

where $p_s = \Pr[s_1, t_2, t_3, \ldots, t_n]$ is the sampling probability of subgraph s.

Theorem 2. *Suppose $c = |Q|$ is the number of subgraphs that meet the query constraints, and $\hat{c}$ is an estimation of c from (1), we have*

$$\mathbb{E}[\hat{c}] = c. \tag{2}$$

If we further assume that the subgraphs are sampled independently, then we have

$$\mathrm{Var}[\hat{c}] = \sum_{s \in Q} \frac{1 - p_s}{p_s}. \tag{3}$$

The proof can be found in the supplementary material. When proportional sampling is used, i.e., $p_s = p, \forall s \in O$, Theorem 2 leads to the following error bounds.

Corollary 2.1. *If we define the estimation error as*

$$err = \frac{|\hat{c} - c|}{c}, \tag{4}$$

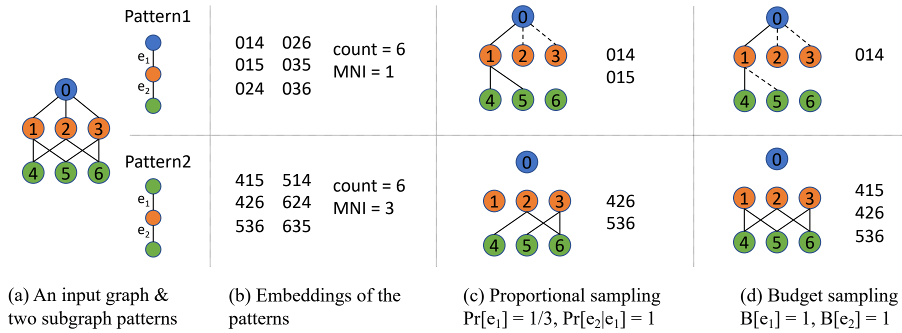


Figure 5: An example of subgraph sampling with loop perforation. We consider the input graph and the sampling of subgraphs of two patterns in (a). All the embeddings of the two patterns are listed in (b). The count is the number of embeddings. With proportional sampling (c), two subgraphs are sampled for both *Pattern*1 and *Pattern*2, the number of sampled subgraphs is proportional to the count of the pattern. With budget sampling (d), one subgraph is sampled for *Pattern*1 and three subgraphs are sampled for *Pattern*2, more subgraphs are sampled for the pattern with larger MNI support.

then we have

$$\mathbb{E}[err] \leq \frac{\sqrt{\text{Var}[\hat{c}]}}{c} = \sqrt{\frac{(1-p)}{pc}}, \tag{5}$$

$$\Pr\left[err \geq \epsilon\right] \leq \exp\left(\frac{-\epsilon^2 cp}{3}\right) \tag{6}$$

for any $0 < \epsilon < 1$ *and* $0 < \mu < 1$.

Formula (5) is obtained by setting $p_s = p$ in (3) and applying the inequality $\mathbb{E}[\sqrt{X}] \leq \sqrt{\mathbb{E}[X]}$. The formula suggests that 1) for a particular query, the larger sampling probability p we use, the smaller estimation error we achieve; 2) for queries with larger c, we can apply more aggressive sampling (i.e., smaller p) to preserve the same estimation error.

Formula (6) is a direct application of Chernoff bound [21]. The formula indicates that the chance of the estimation error exceeding a threshold is exponentially small. While the loop perforation idea seems simple, we find that the error bound matches the bound of the state-of-the-art graph-coloring-based technique for motif counting (See Theorem 3 in [7]). This increases our confidence that loop perforation can be used as a general and efficient sampling technique for subgraph mining if appropriate perforation strategies are used.

In practice, because c is unknown, it is hard to configure p and achieve good estimation in one execution. We can use a small p to obtain a quick estimation and run the sampling procedure multiple times until the average of the estimated counts converges to certain accuracy.

4.2 Sampling for Finding Frequent Subgraphs

In some applications, we are interested in finding the most frequent subgraph patterns that meet certain constraints. The frequency is defined based on certain support measures. The support measures are different from the counts because they need to have the anti-monotone property (explained in §2.2). A pattern with a large count may have a small support if most of its embeddings overlap. For example, *Pattern*1 in Figure 5 has six embeddings, but its MNI support is 1 because all the embeddings have the first vertex mapped to node-0 in the input graph. In contrast, *Pattern*2 has the same number of embeddings, but its MNI support is 3. The proportional sampling method described above cannot find the frequent patterns efficiently. For such queries, we propose a *budget sampling* strategy. The basic idea is to set a limit on the number of sampled subgraphs associated with each vertex so that the number of overlapping samples is limited. Specifically, for single-vertex exploration, we group the edges of each vertex according to their patterns, and we sample a fixed number of edges from each group in each loop level. For two-vertex exploration, we group the neighboring size-3 subgraphs of each vertex according to their patterns and sample a fix number of size-3 subgraphs from each group.

Figure 5d shows an example of budget sampling. For the two subgraph patterns in Figure 5a, we set the sampling budgets for both e_1 and e_2 to 1. For *Pattern*1, suppose edge $(0, 1)$ is sampled in the first loop and edge $(1, 4)$ is sampled in the second loop, we obtain one sampled subgraph '014'. For *Pattern*2, one edge is sampled for each of node-4,5,6 in the first loop. Suppose the sampled edges are $(4, 1)$, $(5, 3)$ and $(6, 2)$. In the second loop, edge $(1, 5)$ is sampled as a neighbor of $(4, 1)$, edge $(3, 6)$ is sampled as a neighbor of $(5, 3)$, and edge $(2, 4)$ is sampled as a neighbor of $(6, 2)$. We obtain three subgraphs '415', '536' and '624'. Compared to proportional sampling, budget sampling returns more subgraphs of *Pattern*2 which has larger MNI support.

4.3 Sampling for Large Graphs

When the input graph is large, there may be a large number of size-3 subgraphs which cannot be entirely stored in memory. In such cases, we can perform sampling when obtaining the size-3 subgraphs. For counting tasks, we use single-vertex exploration with proportional sampling to obtain the size-3 subgraphs. Suppose the sampling probability of each size-3 subgraph is q. The estimation of the subgraph counts is similar to the procedure described in §4.1, only with the p replaced by $p \cdot q^{(n-1)/2}$ if n is odd and replaced by $p \cdot q^{(n-2)/2}$ if n is even. For FSM, we can sample a fixed number of size-3 subgraphs around each vertex in order to have subgraphs evenly distributed over all vertices. This can be done by

```
struct SmpRes {
 double prob;  // sampling probability of current iteration
 bool skip; // skip the current iteration or not };

struct Sampler {
 virtual SmpRes smp_prob(Subgraph s, Subgraph t, int x, int l) {
    return { 1.0;  false; } // no sampling by default };
```

Figure 6: The interface for defining a sampler: s is the intermediate subgraph, t is the joining size-3 subgraph (or edge if single-vertex exploration is used), x is the joining vertex that s and t have in common, and l is the loop level. The sampler returns two values for each loop iteration: $prob$ is the sampling probability, and $skip$ is the sampling outcome.

```
struct BudgetSampler: Sampler {
   // W[i][j] stores the number of subgraphs containing
   //  node i and of pattern j
   map<int, map<int, double>> W;
  // B[l] is the sampling budget in loop level l
   vector<double> B;
   SmpRes smp_prob(Subgraph s, Subgraph t, int x, int l) {
      double p = B[l] / W[x][t.pat_id];
      double r = uniform(0, 1);
      return {p, r >= p} };
```

Figure 7: Implementation of budget sampling.

incorporating sampling into a subgraph matching procedure. We adapted AutoMine [31] for this task.

4.4 Programming Interface

Our system provides a simple interface for defining different sampling strategies, as shown in Figure 6. The *smp_prob* function accepts as input the intermediate subgraph s, the joining size-3 subgraph (or edge) t, the joining vertex x, and the loop level l. It returns two values for each iteration in the lth loop level. The first value *prob* is the sampling probability of the current iteration, and the second value *skip* is the sampling outcome, indicating whether the current iteration should be skipped or not. The users can inherit the `Sampler` class and define their own *smp_prob* function. By default, the function returns $prob = 1$ and $skip = false$, meaning that no sampling is performed.

Figure 7 shows the implementation of budget sampling with our API. Before the exploration procedure, we aggregate the size-3 subgraphs (or edges if single-vertex exploration is used) according to their patterns and store a pattern index for each size-3 subgraph. We then group the neighboring size-3 subgraphs of each vertex according to their patterns and store the number of size-3 subgraphs in each group. For each vertex i, the number of its neighboring size-3 subgraphs of pattern j is stored in $W[i][j]$. The `BudgetSampler` computes the sampling probability of a joining size-3 subgraph t as $B[l]/W[x][t.pat\_id]$ where $B[l]$ is the sampling budget in loop level l and $t.pat\_id$ is the pattern index of t. The sampling probability is compared with a random number between 0 and 1. If the random number is greater than the sampling probability, we skip the current iteration.

Due to space limit, we leave the implementation of proportional sampling in the supplementary material.

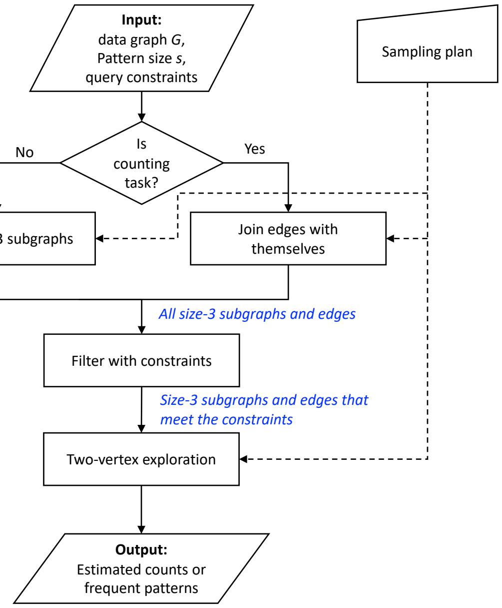


Figure 8: Workflow of SampleMine.

4.5 Putting It Together

We summarize the workflow of SampleMine in Figure 8. Given a data graph, a target pattern size, and query constraints, it starts by collecting the size-3 subgraphs. If the task is to count subgraphs, we obtain the size-3 subgraphs by joining the edges with themselves; if the task is to find frequent patterns, we use AutoMine to match the size-3 subgraphs. Both approaches return the same set of size-3 subgraphs if no sampling is used. For large graphs, we apply random sampling to the joining or matching procedure as described in §4.3. After obtaining all the size-3 subgraphs, we filter out the unwanted ones based on the query constraints. The constraints can be a support threshold for frequent subgraph mining tasks, or they can be any user-defined constraints. The filtered size-3 subgraphs are given to the two-vertex exploration procedure in Figure 1b to find subgraphs of target size. As discussed in this section, we can accelerate the exploration procedure by randomly sampling the loop iterations. Finally, the subgraph counts are estimated or the frequent patterns are returned for the query.

5 EXPERIMENTAL RESULTS

This section presents our experimental setup and performance comparison with the existing graph mining systems.

5.1 Experimental Setup

Platform: We run all the experiments on a workstation with an Intel Xeon W-3225 CPU containing 8 physical cores (16 logical cores) and 192GB memory. We use GCC 7.3.1 for compilation with optimization level O2 enabled.

Datasets: Table 1 lists the graphs used in our experiments. CI and MI are labeled; the other four are unlabeled. We randomly assign 20 labels to the vertices in OK graph and 30 labels to UK and FR.

Table 1: Graph datasets.

Graph	#vertices	#edges	Max_degree
CiteSeer (CI) [18]	3,264	4,536	99
MiCo (MI) [18]	100K	1.1M	1,359
Orkut (OK) [2]	3.1M	117.2M	33,313
UK-2005 (UK) [1]	39M	936M	1,776,858
Friendster (FR) [50]	65M	1.8B	5,214

Tasks: We evaluate our system with subgraph counting (SC), frequent subgraph mining (FSM), and user-defined queries. FSM and SC are two standard SPM tasks and have been described in §2.2. We consider labeled subgraphs for SC which is more challenging than unlabeled motif counting. We also test with five user-defined queries:

- Q1: count subgraphs with at least one vertex of label 1 and one vertex of label 2;
- Q2: count subgraphs with at least two vertices of label 1;
- Q3: count subgraphs that contain triangles with label 1;
- Q4: count subgraphs that do not contain squares with label 1;
- Q5: find subgraphs with label 1 or 2 and return the frequent patterns among them with MNI support greater than a threshold.

For most of the user-defined queries, we check the query constraints and filter out the unwanted subgraphs in the innermost loop. For example, Q1 computes the number of vertices with label 1 and 2 in the subgraph. If both numbers are 0, the subgraph is discarded. Q2 performs filtering in both the innermost loop and the second last loop. If the subgraph does not have label 1 in the second last loop, it cannot have at least two label 1's, so we can filter it out early. Q5 checks the query constraints in the innermost loop before assigning the vertices to the domains, so the MNI support is calculated only with the subgraphs that meet the constraints. We consider vertex-induced subgraphs for SC, Q1, Q2, Q4, Q5, and edge-induced subgraphs for FSM and Q3.

Baselines: We compare our system with three state-of-the-art systems for general-purpose subgraph mining: Peregrine (PR) [24], AutoMine (AM) [31] and Pangolin (PG) [15], an approximate system specialized for subgraph counting: ASAP [23], and a sampling-based system specialized for frequent subgraph mining: ScaleMine (SA) [3]. Pangolin supports unlabeled vertex-based extension or labeled edge-based extension. It cannot enumerate labeled vertex-induced subgraphs for SC, so we only compare with Pangolin for FSM. The source code of ASAP is not available. We implement its neighbor sampling method into AutoMine. ASAP samples one subgraph at a time. When sampling the subgraph, it starts from a random edge in the graph and gradually extends the subgraph by randomly selecting a neighbor of the previous node. If the sampled subgraph belongs to a pattern, ASAP estimates the total number of embeddings of that pattern as the reciprocal of the sampling probability. ASAP runs this sampling procedure for a sufficient number of times and uses the average over executions as the final estimation. The main difference between this neighbor sampling method and our proportional sampling method is that neighbor sampling always samples one edge from a neighbor list. It cannot ensure a higher sampling probability for patterns with more embeddings. ASAP does not support MNI-based FSM. It has limited support for user-defined queries. The users can perform "all" or

Table 2: Sampling ratios for counting tasks.

Graph	CI	MI	OK	UK	FR
$1/sr_1$	1	1	8	1024	32
$\{1/sr_2, 1/sr_3\}$	{2, 4}	{8, 64}	{32, 1024}	{512, 1024}	{8, 32}

Table 3: Execution time (in seconds) of subgraph counting. Systems: SampleMine using two-vertex exploration with sampling (TV-smpl) and without sampling (TV-acc), SampleMine using single-vertex exploration with sampling (SV-smpl) and without sampling (SV-acc), AutoMine (AM), and Peregrine (PR). 'T' represents timeout after 24 hours of execution. 'F' execution failure due to insufficient memory.

Size	Gr.	TV-smpl	SV-smpl	TV-acc	SV-acc	AM	PR
4	CI	0.45	0.78	0.89	1.03	0.90	4.6
5		1.8	3.0	19	26	20	332
6		23	38	530	688	525	26,605
7		470	585	17,808	21,523	17,994	T
4	MI	321	447	19,931	24,856	19,270	F

"atleast-one" predicate subgraph matching, but it does not support the more general queries used in our experiments.

Sampling Ratios: Table 2 lists the sampling ratios for counting tasks in our experiments. Here, sr_1 the sampling ratio of edges for obtaining size-3 subgraphs, sr_2 and sr_3 are the sampling ratios of edges and size-3 subgraphs for two-vertex exploration. The sampling ratios are determined by the following procedure. First, sr_1 is set to ensure that the size-3 subgraphs can be stored in memory. We use Nd^2 (N is the number of vertices, d is the maximum degree) as an upper bound of the number of size-3 subgraphs. Since we know all size-3 subgraph of MI can be stored in memory on our machine (i.e., sr_1 can be set to 1 on MI), we calculate sr_1 for larger graphs with $N_{MI}d_{MI}^2 = N_G(d_G/sr_1)^2$ and round it to the closest power of 2. Next, we determine sr_3 based on an upper bound of the number of size-5 subgraphs (i.e., ND^2 where D is the maximum degree of size-3 subgraphs). Since we know $sr_3 = 1/64$ can obtain good results for MI, we calculate sr_3 for larger graphs with $N_{MI}(D_{MI}/64)^2 = N_G(D_G/sr_3)^2$. Finally, given sr_3, we use NdD as an upper bound of the number of size-4 subgraphs and calculate sr_2 with $N_{MI}(d_{MI}/8)(D_{MI}/64) = N_G(d_G/sr_2)(D_G/sr_3)$.

Parallelization: We use 16 threads for parallel execution for all systems. For our system, the outermost loop is parallelized with OpenMP using dynamic scheduling. For ASAP, since the sampling of subgraphs are independent, we use 16 threads to sample subgraphs at the same time.

5.2 Results for Subgraph Counting

Table 3 shows the execution time of SC with different systems. We list the results of tasks for which at least one of the systems can return accurate results in 24 hours. The execution time of two-vertex exploration includes both the time of the nested loop and the time for obtaining size-3 subgraphs. Without sampling, our system has almost the same execution time as AutoMine and is 5x to 50x faster than Peregrine. This is mainly because Peregrine needs to maintain all the labeled patterns and it is expensive when the number of patterns is large. To show the benefit of two-vertex exploration, we configure our system to run single-vertex exploration. Two-vertex exploration is 1.2x to 1.4x faster than single-vertex exploration. We

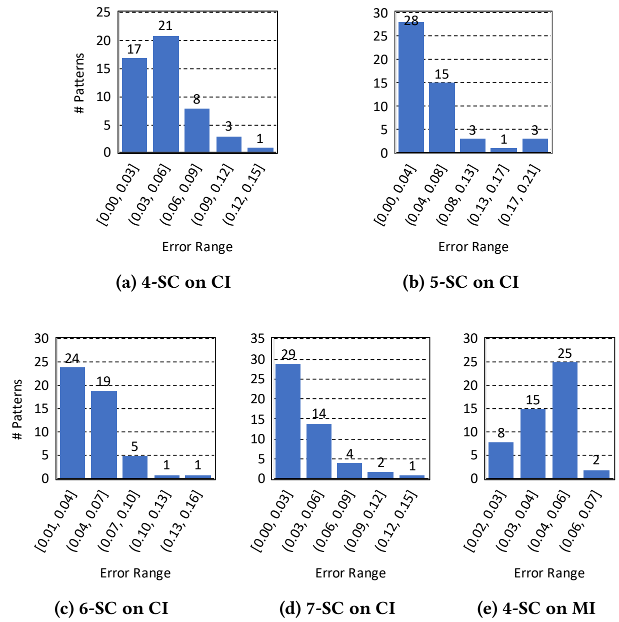


Figure 9: Histogram of estimation errors of SampleMine for the top-50 patterns with most embeddings.

Table 4: Number of patterns returned by SampleMine (SM) and ASAP with the same execution time. Tot# is the total number of patterns. SM/ASAP-tot is the number of patterns returned by SampleMine/ASAP. SM/ASAP-50 is the number of top-50 patterns returned by SampleMine/ASAP.

Size	Gr.	Tot#	SM-tot	SM-50	ASAP-tot	ASAP-50
4	CI	1,141	614	50	225	0
5		7,048	3,028	50	945	37
6		45,917	15,572	50	2,900	0
7		323,794	93,781	50	5,646	0
4	MI	855,010	752,561	50	165,327	0

then apply proportional sampling to these tasks with the sampling ratios listed in Table 2. The sampling brings 2x to 37.9x speedups on CI and a 62.1x speedup on MI. The average (geometric mean) speedup of TV-smpl over AutoMine is 16.3, and the average speedup over Peregrine is 129.7.

Figure 9 shows the histogram of estimation errors for the top-50 patterns with most embeddings for the above tasks. We use the definition of estimation error in (4). The results show that, with the above sampling configuration, our system returns estimation of small errors – for most patterns the estimation error is smaller than 0.06, and the average error over the 50 patterns is smaller than 0.05. Comparing 4-SC on CI and MI graph, we can see that the estimation error on MI graph is smaller than on CI graph even with a smaller sampling ratio. This is because MI has much more size-4 subgraphs than CI. The top-50 labeled size-4 patterns in MI have 1.57×10^7 to 2.36×10^9 embeddings, while the top-50 size-4 patterns in CI have only 434 to 1.47×10^5 embeddings. According to Corollary 2.1, the more subgraphs a query returns, the more aggressive sampling we can use to preserve the same accuracy.

Table 5: Execution times of subgraph counting in hours with SampleMine.

Size	Gr.	Time per exec	Max_err
5	MI	1.9	0.02
5	OK	5.4	0.04
4	UK	4.3	0.05
4	FR	5.9	0.03

Table 6: Execution times of frequent subgraph mining in seconds with different systems.

Size	Sup.	Gr.	SM	TV-acc	SV-acc	AM	PR	PG	SA
4	0.001	CI	0.45	0.82	1.3	1.8	5.4	5.5	1.5
	0.005		0.45	0.82	1.3		4.8	4.8	1.5
	0.01		0.38	0.77	1.2		3.4	3.7	1.4
	0.05		0.27	0.69	0.93		1.1	2.8	1.1
4	0.001	MI	505	54,903	62,062	78,244	F	F	57,754
	0.005		489	42,695	49,179				45,820
	0.01		387	31,115	38,989				30,475
	0.05		252	25,719	30,063				25,223
5	0.001	CI	3.2	35	46	68	F	F	40
	0.005		2.6	27	40				37
	0.01		2.2	25	39				27
	0.05		1.7	23	32				23
6	0.001	CI	54	1,135	1,482	1,924	F	F	1,096
	0.005		39	1,047	1,443				1,030
	0.01		37	1,052	1,420				1,025
	0.05		22	749	1,076				804

We run ASAP for the same amount of time as our system and compare the estimation accuracy. Table 4 lists the number of patterns returned by SampleMine and ASAP. The total number of patterns returned by SampleMine (SM-tot) is 2.7x to 16.6x that of ASAP (ASAP-tot). For the top-50 patterns, our system returns all the 50 patterns (SM-50), while ASAP returns none for most tasks and returns 37 patterns for 5-SC on CI graph. For the 37 patterns that ASAP finds, the average error is 15.7, the maximum error is 430, the minimum error is 0.05, and the median error is 0.9. The results show that our system is more effective in finding significant patterns and obtains more accurate estimations than ASAP.

Table 5 shows the execution time of tasks on larger graphs for which we cannot obtain accurate results in 24 hours. We apply proportional sampling with ratios as listed in Table 2. Since the actual counts are unknown, we run the experiment for 10 times and calculate the empirical error by replacing c in (4) with the average count of the 10 runs. The maximum error of the 10 runs for the top-50 patterns is listed in the last column of Table 5. For all the testcases, the errors are smaller than 0.05. Again, we are able to apply aggressive sampling to these large graphs because they have a large number of subgraphs.

5.3 Results for Frequent Subgraph Mining

Table 6 lists the execution times of FSM for which at least one of the systems can return result within 24 hours. We find that Peregrine and Pangolin abort for most tasks. Peregrine paper [24] only reports results of 3-FSM. Pangolin [15] reports results mostly for 3-FSM. It reports 4-FSM for only one graph using large support thresholds, but it fails to give result for MI. For the only one testcase (4-FSM on CI) that Peregrine and Pangolin do return, our system (TV-acc) is 1.6x to 6.8x faster without any sampling. AutoMine is able to return results for these tasks. However, because it matches the patterns in a depth-first order, it cannot benefit from the anti-monotone

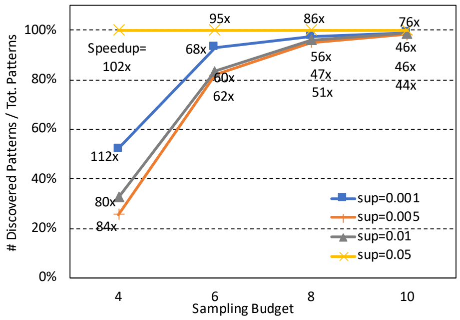


Figure 10: Number of discovered size-4 frequent patterns on MI graph with different support thresholds and different sampling budgets. Sampling budget x means that the size-3 and size-2 subgraphs are sampled with budget x^2 and x during the exploration procedure.

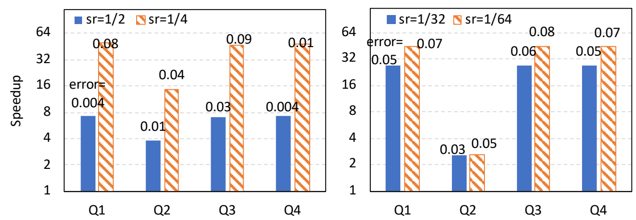
property (i.e., it does not run faster for larger support thresholds). Our system prunes the infrequent size-3 subgraph before the exploration procedure, and it runs 1.6x to 3.2x faster than AutoMine without sampling. Compared with ScaleMine (SA) [3] which uses node sampling for support estimation, our system (TV-acc) achieves almost the same performance without using any sampling.

We then configure our system to run budget sampling with budget 4^2 and 4 for size-3 and size-2 subgraphs on CI, and budget 6^2 and 6 for size-3 and size-2 subgraphs on MI. The execution times are listed in column 'SM' in Table 6. Our system runs 4.1x to 310x faster than the compared systems while returning more than 80% of the frequent patterns for all these tasks. The average (geometric mean) speedup is 34.7 against AutoMine, 8.3 against Peregrine, 10.7 against Pangolin, and 18.7 against ScaleMine. Figure 10 shows the number of size-4 frequent patterns found by our system with different support thresholds and different sampling budgets on MI graph. The speedups over non-sampling execution (TV-acc) are labeled on the lines. The total number size-4 frequent patterns on this graph is 249140, 54164, 12241 and 9 for support threshold of $0.001N$, $0.005N$, $0.01N$ and $0.05N$ where N is the number of vertices. When the sampling budget is set to 4, our system returns all the 9 patterns with support $0.05N$ using only $1/102$ of the total execution time. When the sampling budget increases to 6, our system returns more than 80% of the frequent patterns for all support thresholds with about $1/60$ of the total execution time. When the sampling budget increases to 10, our system still achieves more than 40x speedups while obtaining more than 98% of the frequent patterns for all support thresholds.

Table 7 lists the results of 5-FSM on UK graph with our system. Since the size-3 subgraphs cannot be entirely stored in memory, we perform sampling during the matching phase as described in §4.3. The matching procedure takes a large proportion of the total execution time. Once the size-3 subgraph are sampled, we can find size-5 frequent patterns in a relatively short time. None of the compared systems (including the sampling-based ScaleMine [3]) can return results for this task within 24 hours. This shows the main advantage of our system against previous sampling-based systems: while our system does not guarantee to find all the frequent patterns, it is able to return the most frequent patterns quickly, which are of most interest in real-world applications.

Table 7: Results of 5-FSM on UK graph with different sampling budgets and support thresholds. 'M. sb' is the number of size-3 subgraphs sampled from each vertex by the match procedure. 'M. time' is the time for obtaining size-3 subgraphs. 'J. sb' is the sampling budget for loops that iterate over size-3 subgraphs in two-vertex exploration. 'J. time' is the execution time of the nested loop.

Sup.	M. sb	M. time (sec)	J. sb	J. time (sec)	# patterns
0.0001	2	2,254	4	140	76
			16	288	143
	4	2,530	4	185	105
			16	505	188
0.0005	2	2,254	4	105	1
			16	235	3
	4	2,530	4	138	8
			16	421	14



(a) size-7 on CI **(b) size-4 on MI**

Figure 11: Speedups for user-defined counting tasks with different sampling ratios: '$sr = 1/x$' means that two-vertex exploration uses sampling ratio $1/x$ for loops that iterate over size-3 subgraphs and $1/\sqrt{x}$ for loops that iterate over edges.

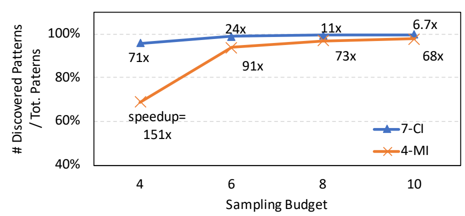


Figure 12: Speedups for Q5 with different sampling budgets.

5.4 Results for User-Defined Queries

We first run Q1~Q5 on CI and MI graph where accurate results are available. Figure 11 shows speedups for Q1~Q4 with different sampling ratios. We can see that the queries run 3.8x to 7.3x faster for size-7 subgraphs on CI with $sr = 1/2$ and 14.5x to 49.2x faster with $sr = 1/4$. For size-4 subgraphs on MI, the queries run 2.6x to 26.9x faster with $sr = 1/32$ and 2.7x to 44.4x faster with $sr = 1/64$. The speedups for Q2 are smaller than for other queries because Q2 has a relatively small exploration space as the subgraphs without label 1 are pruned in the second last loop. The estimation errors are labeled in the figure. The errors are smaller than 0.1 for all tasks.

Table 8: Execution times of user-defined queries in hours with SampleMine.

Task	Gr.	Time per exec	Err.
5-Q1	MI	3.5	0.02
5-Q1	OK	10.1	0.03
4-Q1	UK	4.4	0.05
4-Q1	FR	5.8	0.04
5-Q2	OK	8.3	0.04
4-Q2	UK	2.3	0.05
4-Q2	FR	3.1	0.05

Figure 12 shows the speedups for Q5 with different sampling budgets. As expected, the number of discovered patterns increases with the sampling budget. When the budget is set to 6, our system can obtain 98.7% of the frequent patterns with only 1/24 of the total execution for size-7 subgraphs on CI, and obtain 93.7% of the frequent patterns with only 1/91 of the total execution time for size-4 subgraphs on MI.

We then run Q1 and Q2 on larger graphs with the sampling ratios listed in Table 2. Table 8 shows the execution time of different tasks on different graphs. Because the accurate results are unknown, we run the experiments for 10 times and compute the empirical estimation error. For all the tasks, our system returns estimation of high accuracy. The same as for subgraph counting, we are able to apply aggressive sampling to these user-defined queries on large graphs because the subgraph counts are large. (The estimated counts are from 10^{12} to 10^{20}.) It is possible that for some user-defined queries with stricter constraints there are not many output subgraphs. For such queries, we cannot use a small p with proportional sampling. Formula (3) suggests that the estimation error can be reduced by assigning higher sampling probabilities to subgraphs in Q. For example, if the task is to count the number of subgraphs with label 1, a better sampling strategy might be assigning higher probabilities to the edges or size-3 subgraphs that contain label 1. We leave it for future work to investigate more sophisticated sampling strategies (e.g. adaptive sampling) under the loop-perforation framework.

6 RELATED WORK

There is a growing interest in supporting general-purpose subgraph pattern mining in recent year. Many systems and task-independent optimizations have been proposed.

Subgraph-Centric Systems: Arabesque [43] is a distributed system that enumerates all possible embeddings in multiple rounds and uses a filter-process model to generate the results. RStream [47] is the first single-machine, out-of-core graph mining system. It supports a rich programming model that exposes relational algebra for developers to express various mining tasks and a runtime engine that can efficiently compute the relational operations. Pangolin [15] also targets single-machine but provides GPU acceleration. DistGraph [42] and G-miner [12] are distributed graph mining systems that adopt breadth-first exploration. DistGraph focuses on reducing the communication of distributed computing when each node can only have a portion of the graph. G-miner proposes a block-based graph partitioning technique and uses work stealing to achieve good load balance. These systems use breadth-first exploration and need to store all intermediate results. The large memory consumption prevents them from efficiently mining for large graphs. Fractal [17] addresses the memory consumption issue by implementing depth-first exploration. All of the existing subgraph-centric systems are based on single-vertex exploration. Our system is the first to use two-vertex subgraph exploration.

Pattern-Based Systems: AutoMine [31] is a single-machine graph mining system that features compiler-based optimizations. Their main idea is to enumerate all unlabeled patterns of a particular size and match them one-by-one on a graph. Because the patterns are given, AutoMine is able to search an optimal matching strategy and combine matching procedures of multiple patterns. Peregrine [24] is another pattern-based system. Instead of enumerating all patterns before matching, it discovers patterns based on the subgraphs it has explored and maintains a list of the patterns. DwavesGraph [13] lists all the unlabeled patterns and uses pattern decomposition for fast matching of the patterns. Similar to AutoMine, it needs to know all the unlabeled patterns in advance. Sandslash [14] supports both subgraph-centric and pattern-based exploration. To achieve both high productivity and high efficiency, it provides a high-level API for specifying the graph mining problems and performing subgraph enumeration automatically, and a low-level API that allows users to express algorithm-specific optimizations.

Approximate Subgraph Pattern Mining: Random sampling has been widely used to reduce the computational complexity of SPM [3, 9, 11, 19, 37, 38]. Motivo uses graph coloring and adaptive sampling to accelerate motif counting [9]. ScaleMine proposes a sampling technique to accelerate FSM by estimating the MNI support of a pattern without enumerating all of its embeddings [3]. Sampling has also been used for accelerating FSM in a database of graphs [4, 40]. The main idea of these works is to perform random walk in the space of all patterns. By carefully setting the sampling probability at each step, they ensure that patterns of higher supports are more likely to be sampled [4]. These sampling techniques are task-specific and cannot be easily adopted for new applications. Sampling has been adopted in pattern-based graph mining systems [23, 32]. The idea is to sample edges in the graph based on the given patterns and estimate the actual results with the sampled results. These systems are good at counting subgraphs of a given pattern. However, as we show in the experiments, when the pattern is unknown, such neighbor sampling technique is not effective in finding the patterns with the most embeddings.

7 CONCLUSION

In this work, we propose a framework for applying random sampling to general-purpose subgraph pattern mining. Our system is designed with two novel techniques: two-vertex subgraph exploration and loop-perforation-based subgraph sampling. We show that two-vertex exploration accelerates subgraph exploration procedure by extending the subgraph by two vertices in each step. We also show that our loop-perforation-based sampling technique is flexible and can be used for designing efficient sampling strategies for different SPM tasks. The experiments show that our system significantly outperforms other state-of-the-art subgraph pattern mining systems for different tasks on various input graphs.

ACKNOWLEDGEMENTS

This work was supported by NSF award CCF-2028825.

REFERENCES

[1] Dataset for "Statistics and Social Network of YouTube Videos" . http://netsg.cs.sfu.ca/youtubedata/.

[2] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[3] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 716–727. IEEE, 2016.

[4] Mohammad Al Hasan and Mohammed J Zaki. Output space sampling for graph patterns. *Proceedings of the VLDB Endowment*, 2(1):730–741, 2009.

[5] Suman K Bera and C Seshadhri. How to count triangles, without seeing the whole graph. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 306–316, 2020.

[6] Vandana Bhatia and Rinkle Rani. Ap-fsm: A parallel algorithm for approximate frequent subgraph mining using pregel. *Expert Systems with Applications*, 106:217–232, 2018.

[7] M. Bressan, Stefano Leucci, and A. Panconesi. Motivo: Fast motif counting via succinct color coding and adaptive sampling. *Proc. VLDB Endow.*, 12:1651–1663, 2019.

[8] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–25, 2018.

[9] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. Motivo: fast motif counting via succinct color coding and adaptive sampling. *arXiv preprint arXiv:1906.01599*, 2019.

[10] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 858–863. Springer, 2008.

[11] Mostafa Haghir Chehreghani, Talel Abdessalem, Albert Bifet, and Meriem Bouzbila. Sampling informative patterns from large single networks. *Future Generation Computer Systems*, 106:653–658, 2020.

[12] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[13] Jingji Chen and Xuehai Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition, 2020.

[14] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, pages 378–391, 2021.

[15] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205, April 2020.

[16] Wei-Ta Chu and Ming-Hung Tsai. Visual pattern discovery for architecture image classification and product image search. In *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, ICMR '12, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1357–1374, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014.

[19] Ruohan Gao, Huanle Xu, Pili Hu, and Wing Cheong Lau. Accelerating graph mining algorithms via uniform random edge sampling. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[20] Shayan Ghazizadeh and Sudarshan S Chawathe. Seus: Structure extraction using summaries. In *International Conference on Discovery Science*, pages 71–85. Springer, 2002.

[21] Michel Goemans. Chernoff bounds, and some applications. https://math.mit.edu/~goemans/18310S15/chernoff-notes.pdf.

[22] Guyue Han and Harish Sethu. Waddling random walk: Fast and accurate mining of motif statistics in large graphs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 181–190. IEEE, 2016.

[23] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[24] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007.

[26] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *Proceedings 2001 IEEE international conference on data mining*, pages 313–320. IEEE, 2001.

[27] Michihiro Kuramochi and George Karypis. Grew-a scalable frequent subgraph discovery algorithm. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 439–442. IEEE, 2004.

[28] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *The VLDB Journal*, 26(3):421–446, 2017.

[29] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[30] Shikai Li, Sunghyun Park, and Scott Mahlke. Sculptor: Flexible approximation with selective dynamic loop perforation. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 341–351, 2018.

[31] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 509–523, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. Approxg: Fast approximate parallel graphlet counting through accuracy control. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 533–542. IEEE, 2018.

[33] Brendan D McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, USA, 1981.

[34] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[35] Aduri Pavan, Srikanta Tirthapura, et al. Counting and sampling triangles from a graph stream. 2013.

[36] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*, pages 1431–1440, 2017.

[37] Giulia Preti, Gianmarco De Francisci Morales, and Matteo Riondato. Maniacs: Approximate mining of frequent subgraph patterns through sampling. KDD '21, page 1348–1358, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Sumit Purohit, Sutanay Choudhury, and Lawrence B Holder. Application-specific graph sampling for frequent subgraph mining and community detection. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1000–1005. IEEE, 2017.

[39] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.

[40] Tanay Kumar Saha and Mohammad Al Hasan. Fs3: A sampling based method for top-k frequent subgraph mining. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 8(4):245–261, 2015.

[41] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134, 2011.

[42] Nilothpal Talukder and Mohammed J Zaki. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery*, 30(5):1024–1052, 2016.

[43] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440, 2015.

[44] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 1307–1318, New York, NY, USA, 2013. Association for Computing Machinery.

[45] A Vazquez, R Dobrin, D Sergi, J-P Eckmann, Zoltan N Oltvai, and A-L Barabási. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *Proceedings of the National Academy of Sciences*, 101(52):17940–17945, 2004.

[46] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 1505–1514, 2020.

[47] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 763–782, 2018.

[48] Pinghui Wang, John CS Lui, Don Towsley, and Junzhou Zhao. Minfer: A method of inferring motif statistics from sampled edges. In *2016 IEEE 32nd international conference on data engineering (ICDE)*, pages 1050–1061. IEEE, 2016.

[49] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724, 2002.

[50] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.