Efficient Mining of Frequent Subgraphs with Two-Vertex Exploration

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

Rujia Wang  
Illinois Institute of Technology  
rwang67@iit.edu

Bo Wu  
Colorado School of Mines  
bwu@mines.edu

Abstract

Frequent Subgraph Mining (FSM) is the key task in many graph mining and machine learning applications. Numerous systems have been proposed for FSM in the past decade. Although these systems show good performance for small patterns (with no more than four vertices), we found that they have difficulty in mining larger patterns. In this work, we propose a novel two-vertex exploration strategy to accelerate the mining process. Compared with the single-vertex exploration adopted by previous systems, our two-vertex exploration avoids the large memory consumption issue and significantly reduces the memory access overhead. We further enhance the performance through an index-based quick pattern technique that reduces the overhead of isomorphism checks, and a subgraph sampling technique that mitigates the issue of subgraph explosion. The experimental results show that our system achieves significant speedups against the state-of-the-art graph pattern mining systems and supports larger pattern mining tasks that none of the existing systems can handle.

1 Introduction

Frequent Subgraph Mining (FSM) is an important operation on graphs and is widely used in various application domains, including bioinformatics [22, 29], computer vision [11], and social network analysis [28]. The task is to discover frequently occurring subgraph patterns from an input graph. Different from graph pattern matching problems where a query pattern is given, FSM needs to find the important patterns based on a support measure and thus has a much larger exploration space.

Since the patterns of interest are unknown, most systems for FSM take an explore-aggregate-filter approach [10, 12, 27, 30]. The principle is to explore all the subgraphs, aggregate the subgraphs according to their patterns, and filter out the subgraphs that are redundant or are not of interest. The exploration happens in a vertex-by-vertex manner where smaller subgraphs are iteratively extended based on the connections in the graph. There are mainly two ways for exploration: breadth-first and depth-first. Starting from all vertices in the graph, breadth-first exploration stores all subgraphs of size \( l \) and extends them with one more vertex to find subgraphs of size \( l + 1 \). The main problem of breadth-first exploration is that the intermediate data can easily exceed the memory capacity as the subgraph size grows. With depth-first exploration, a subgraph of size \( l \) is immediately extended to a subgraph of size \( l + 1 \) without seeing other subgraphs of size \( l \). It needs not save the intermediate subgraphs and thus can explore larger patterns. However, depth-first exploration cannot exploit the anti-monotone property to prune the search space [12], resulting in a lot of unnecessary computation.

Some recent graph mining systems take a pattern-based approach [15, 17]. The idea is to enumerate the (unlabeled) subgraph patterns and then perform pattern matching on the graph. Because the pre-generated patterns guide the exploration, these systems need not store any intermediate data, and the aggregation overhead can be reduced as the topology of the subgraphs is given. However, this approach only works well for small patterns because when the pattern is larger (more than 6), listing all subgraph patterns itself becomes a hard problem [2, 19, 20]. It is also difficult for the pattern-based systems to exploit the anti-monotone property to prune the search space. Peregrine [15] maintains a list of frequent patterns, extend the patterns with one vertex or edge, and then re-match the extended patterns on the graph. It prunes the search space without storing the intermediate subgraphs, but the re-matching incurs a lot of redundant computation. These issues have impeded the existing graph mining systems from supporting FSM for large patterns. In fact, most of the prior work only reports experimental results for FSM with no more than 4 vertices.

To enable large pattern mining, we propose a novel two-vertex exploration method in this work. Our key observation is that vertex-by-vertex exploration is not necessary for pattern mining. Instead, we can perform two-vertex exploration that joins size-\((n - 2)\) subgraphs with size-3 subgraphs on a common vertex to obtain subgraphs of size-n. The new exploration method significantly accelerates the exploration process and reduces the memory access overhead in the join operation. It also allows us the exploit the anti-monotone property to prune the exploration space without storing the intermediate subgraphs or re-matching the patterns.

To further accelerate the mining process, we propose two new techniques to overcome the performance bottlenecks. One performance bottleneck is due to the expensive isomorphism checks in the aggregation step. To aggregate the subgraphs based on their patterns, we need to generate a canonical form for each subgraph such that the subgraphs with the same canonical form are isomorphic. Unfortunately,
The best known algorithms for generating such canonical forms have exponential complexity [7, 25, 31]. Therefore, we want to perform isomorphism check for as few subgraphs as possible. Previous work has employed a quick pattern technique to reduce the number of isomorphism checks [27, 30]. The main is to first group the subgraphs based on an easily computed pattern (e.g., a list of all edges). Since subgraphs in the same group must be isomorphic, only one isomorphism check is needed for each group. We improve on this idea by proposing an index-based quick pattern technique. It assigns an index to each pattern and uses the indices to compute a quick pattern for the joined subgraph. Compared with the quick pattern technique used in prior work, our quick pattern encodes the information of sub-patterns and achieves more accurate grouping of the subgraphs, leading to a significant reduction of isomorphism checks.

Another more fundamental challenge of mining large patterns on graphs is due to the exponential growth of the exploration space. For example, in a median size graph, MiCo [13], which has $9 \times 10^8$ vertices and $10^6$ edges, there are more than $10^{12}$ size-5 subgraphs. When the pattern size increases to 7, the estimated number of subgraphs is in the order of $10^{17}$ for which exhaustive enumeration becomes infeasible. To mitigate this issue, we propose a subgraph sampling technique. The idea is that we sample a small subset of size-3 subgraphs for exploring larger subgraphs during the joining and/or the matching phase. Since the subgraphs of frequent patterns are more likely to be sampled, we are able to discover frequent patterns with only a small number of sampled subgraphs. Compared with previous works that apply edge or neighbor sampling to FSM [14, 18], we can discover more frequent patterns with the same or less computation. This is because subgraph samples preserve more structural information of the graph than edge samples.

We perform extensive evaluation of our system and compare with three state-of-the-art graph mining systems: AutoMine [17], Peregrine [15], and Pangolin [10]. The results show that without using sampling our system achieves 1.8x to 8.4x speedups on tasks for which the compared systems are refered to [21] for detailed descriptions and computation complexity while achieving a good accuracy in measuring the ‘frequency’ of patterns in a graph. The readers are refered to [21] for detailed descriptions and computation complexity of different support measures. We adopt the MNI support for our experiments, although our proposed techniques are applicable to any support measure with the anti-monotone property.

With a support measure $\sigma$, 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 number of vertices in the pattern and $t$ is a threshold.

### 2.2 Frequent Subgraph Mining

The task of Frequent Subgraph Mining (FSM) 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. While the definition of the support measure can vary across applications, the support usually needs to satisfy an anti-monotone property, i.e., the support of a pattern should be no greater than the support of its sub-patterns [21].

**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 $\{f_1, f_2, \ldots, f_m\}$ in $G$, the minimum image based (MNI) support of $P$ in $G$ is defined as

$$\sigma_{\text{MNI}}(P, G) = \min_{\omega \in V_P} \{ |f_i(\omega) : i = 1, 2, \ldots, m | \}. $$

Other support measures include maximum independent set based (MIS) support, minimum instance based (MI) support, and maximum vertex cover based (MVC) support. All these support measures are anti-monotone. MNI support is the most commonly used one because it has linear computation complexity while achieving a good accuracy in measuring the ‘frequency’ of patterns in a graph. The readers are refered to [21] for detailed descriptions and computation complexity of different support measures. We adopt the MNI support for our experiments, although our proposed techniques are applicable to any support measure with the anti-monotone property.

With a support measure $\sigma$, 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. The support can be calculated with either vertex-induced subgraphs or edge-induced subgraphs. Our proposed techniques work for both cases. We use edge-induced subgraphs for experiments, as it is the common setting in prior work [12, 15, 30].

3 Illustration of the Idea

Before getting into technical details, we describe our idea of two-vertex exploration with an example. We use an unlabeled graph for simple illustration.

Suppose our task is to discover size-5 patterns in an input graph (as shown in Figure 1a). We can first find all size-3 subgraphs and join them on a common vertex to obtain size-5 subgraphs. In this example, we first apply a matching algorithm to obtain all the embeddings of size-3 patterns (i.e., wedge and triangle) as listed in Figure 1b. Each pattern is assigned an index (0 for wedge and 1 for triangle in this example), and the index is stored with each embedding during the pattern matching.

Next, we calculate the MNI support for each size-3 pattern and prune the patterns with support less than the threshold. In this example, the supports for both wedge and triangle is 3. Suppose we set the support threshold to 3. Neither of the patterns will be pruned. After we obtain the pruned size-3 subgraphs, we perform binary join on every pair of the columns (i.e., \( \langle 1,1 \rangle, \langle 1,2 \rangle, \langle 1,3 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle, \langle 2,3 \rangle, \langle 3,1 \rangle, \langle 3,2 \rangle, \langle 3,3 \rangle \)) to explore size-5 subgraphs. Figure 1c shows how we can obtain four size-5 subgraphs by joining column \( \langle 1,1 \rangle \) on key 3. Every pair of subgraphs with key 3 are tested (i.e., \( \langle 342', 342 \rangle \), \( \langle 342, 352 \rangle \), ... \( \langle 387', 387 \rangle \), \( \langle 387, 387 \rangle \)). If two subgraphs have one and only one common vertex, they compose a valid size-5 subgraph. In this example, '342' and '375' make up a valid size-5 subgraph '34275'; '342' and '387' make up '34287'; '352' and '387' make up '35287'; and '342' and '385' make up '34285'. These valid joins are marked with connected arrows in Figure 1c. We can see that, through the join operation, we grow the pattern size from 3 to 5 in one exploration step. We will show that such two-vertex exploration is exhaustive for subgraph exploration in §4.1.

One may notice that the result of joining '342' with '385' ('37485') is not included in Figure 1c. This is because our system performs an automorphism check when generating the join results to remove redundancy. We propose a smallest-vertex-first dissection method that ensures only the results that are obtained by joining the subgraph of the smallest spanning vertex indices are saved. In this case, the '37485' subgraph will be generated when we join the third column of '543' and the first column of '387'. More details on the automorphism check and redundancy removal are explained in §4.3.

The above procedure can be extended to explore larger subgraphs by joining multiple subgraph lists. For example, a 3-way join of two size-3 subgraphs and one size-2 subgraph (i.e. edges) will explore all size-6 subgraphs. A 3-way join of size-3 subgraphs will explore all size-7 subgraphs. Given an input graph \( G \), a pattern size \( s \), and a support threshold \( t \), the workflow of our frequent subgraph mining algorithm is summarized as follows:

Step1: Obtain all size-3 subgraphs by matching.
Step2: Calculate the support for each size-3 and size-2 pattern, and remove patterns with support smaller than \( t \) along with their subgraphs.
Step3: Perform multi-way join of size-3 subgraphs and/or edges to obtain subgraphs of size \( s \): if \( s == 2n + 1 \), join \( n \) size-3 subgraph lists; if \( s == 2n \), join the edge list with \( n - 1 \) size-3 subgraph lists.
Step4: Calculate support for each size-\( s \) pattern and remove patterns with support smaller than \( t \).

For Step1, any matching algorithm will work; we use AutoMine [17] in our implementation. Step2 and Step4 are straightforward based on the definition of the support measure. Step3 is the most important step in the algorithm. We will detail this step in the next section.

4 Subgraph Exploration Process

All the current graph mining systems based on the explore-aggregate-filter approach use single-vertex exploration because it ensures that all the size-\( n \) subgraphs can be found by extending the size-(\( n - 1 \)) subgraphs with an edge. We find
that limiting the step size to 1 is not a must to find all patterns. This section describes our two-vertex exploration idea and explains its advantage over single-vertex exploration.

4.1 Two-Vertex Exploration

We propose to explore the size-\(n\) subgraphs by joining the size-(\(n-2\)) subgraphs with the size-3 subgraphs (i.e., wedges and triangles). The completeness of this two-vertex exploration method is summarized as follows.

**Theorem 1.** 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:** Our goal is 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 we join all size-(\(n-2\)) and size-3 subgraphs 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; 2) if the new vertex is only connected with the size-3 subgraph, it is easy to verify that for either of the two cases (wedge or triangle), we can always pick three connected vertices as the new dissection. 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. \(\square\)

Note that multi-vertex exploration is not complete with more than two vertices. For example, a seven-vertex three-pronged star graph with two vertices in each prong cannot be obtained by joining any two size-4 subgraphs. Therefore, we cannot explore more than two vertices in each step.

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.

4.2 Depth-First Multi-Way Join of Subgraphs

To avoid the large memory consumption, we implement the exploration process as a depth-first multi-way join. Suppose we want to join \(r\) subgraph lists \(SL_1, SL_2, ..., SL_r\) and the subgraph in \(SL_i\) has \(l_i\) vertices. For each subgraph list \(SL_i\), we group the subgraphs by each of its \(l_i\) columns, create \(l_i\) hash tables and store the hash tables in \(H_i\). For example, the size-3 subgraphs in Figure 1c are grouped by the vertex indices in the first column. Once the hash tables are created, the multi-way join operation is simply a nested loop that iterates over all possible combinations of subgraphs in different hash tables, as shown in Figure 2. We first enumerate all possible combinations of columns in different subgraph lists by iterating over all hash tables of each subgraph list. Then, we identify the matching keys \(k_i\) in the first two hash tables and try to combine the subgraphs \((s_1, t_1, k_1)\) on the key. If the two subgraphs make up a valid larger subgraph \(s_2\), we iterate over all the vertices of \(s_2\) and look up each vertex \(k_2\) in the third hash table. For every subgraph \(t_3\) with key \(k_2\) in the third hash table, we combine \(s_2\) with \(t_3\) to obtain a larger subgraph. For joining more subgraph lists, the code simply repeats the for loop.

Depth-first join is also used in Fractal [12] for single-vertex exploration. The main issue is that it incurs a huge amount of redundant memory accesses. Our two-vertex exploration mitigates this issue as it requires fewer join steps to enumerate subgraphs of a certain size. To see this, let us consider the exploration of size-5 subgraphs. With single-vertex exploration, it requires a 4-way join of the edges in the graph. The first join operation of the edge list does not incur any redundant memory accesses as each neighbor list is accessed only once in the two hash tables. However, when we join the intermediate size-3 subgraphs with the edge list, we need to query the edge list for each intermediate subgraph. For non-consecutive size-3 subgraphs of the same key, each neighbor list will be accessed multiple times during the join process. The same problem exists when joining size-4 subgraphs with the edge list. In contrast, two-vertex exploration obtains size-5 subgraphs by performing a binary join of size-3 subgraphs which incurs no redundant memory accesses. Our experimental results also validate this point.

4.3 Redundancy Removal through Smallest-Vertex-First Dissection

Combing small subgraphs in different ways can lead to identical results. As we briefly mentioned in Figure 1c, joining subgraph ‘342’ and ‘375’ generates the same subgraph as

---

\[\text{Figure 2. Code of multi-way join.}\]
Algorithm 1: Combine two subgraphs and check for automorphism.

```
func dissect(s', n):
    foreach v in s' in ascending order do
        l = the first n vertices visited by starting from v
        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 identical vertices other than k then return Ø;
    // s' is a valid subgraph joined by s and t
    // find the smallest dissection of s'
    l, r = dissect(s', t.size);
    // if the two joining subgraphs correspond to the smallest
dissection, return s'
    if l == t and r == s then return s';
else return Ø;
```

joining ‘352’ and ‘274’. These redundant subgraphs incur redundant computation, and the redundancy can accumulate over the exploration steps. To eliminate the redundant subgraphs, we perform an automorphism check when a subgraph is generated. The previous automorphism check technique for single-vertex exploration is based on the concept of the canonicality of the subgraphs [27]. This canonicality check does not work for multi-vertex exploration because the small subgraphs are generated by a matching algorithm and may not have the canonicality property. We propose a smallest-vertex-first dissection method that enables the redundancy removal for multi-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 producing a subgraph s’ only if the two joining subgraphs correspond to this unique dissection of s’. The automorphism check is performed each time we combine two subgraphs (i.e., in the combine function in Figure 2). Algorithm 1 shows the procedure of the combine function. For a pair of input subgraphs s and t (t is usually a size-3 subgraph), we first check if there are any other identical vertices except for the joining vertex k. If yes, s and t cannot form a valid subgraph, and the function return an empty set. 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 the smallest n vertices and store them in l where n is the size of t. 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 s and t. If yes, the combine function returns the combined subgraph; otherwise, it returns an empty set.

Example: The smallest-vertex-first dissection of the subgraph ‘34257’ in Figure 1a can be obtained by spanning from vertex 2. The two adjacent vertices of 2 are 3 and 8. Because 3 is smaller, we take 3 in the first step, and the visited set contains vertex 2 and 3. The vertices that are adjacent to the two visited vertices are 4, 5, 7, 8. Because 4 is the smallest, we take 4 in the next step, and we have three vertices 2, 3, 4 in l. The unvisited vertices are 5 and 7. We check if any of 2, 3, 4 can form a connected graph with 5, 7, and we find 3 is the smallest vertex that connects 5 and 7. The algorithm stops and returns l = {2, 3, 4} and r = {3, 5, 7}. When joining the two subgraph lists in Figure 1c, our system generates ‘34257’ (by combining ‘342’ and ‘375’) instead of ‘35274’ (by combining ‘352’ and ‘274’). For the same reason, ‘37485’ is not generated by combining ‘374’ and ‘385’ as the smallest dissection of ‘37485’ is ‘345’ and ‘387’.

The worst cases complexity of the algorithm is $O(|s'|^3)$. Although it is higher than the linear complexity of the automorphism check for single-vertex exploration [27, 30], the actual number of instructions does not increase much because s’ is small and the algorithm usually returns early at line 7.

4.4 Pattern Aggregation with Index-based Quick Pattern

Next, we need to aggregate the subgraphs according to their patterns. This is done by computing the canonical form of each subgraph. The subgraphs with the same canonical form are isomorphic and will be put in the same group. As pointed out in §2, computing the canonical form is expensive, especially for large patterns. Previous work has used a quick pattern technique to reduce the canonical form computation. However, their quick patterns encode little topological information of the subgraphs, resulting in a lot of quick pattern groups of isomorphic subgraphs.

We propose an index-based quick pattern technique that can achieve more accurate grouping of subgraphs and reduce the overhead of canonical form computation. The idea is to assign an index to each pattern in a subgraph list and use the indices for computing the quick pattern of the combined subgraph. If a subgraph list is generated by the matching algorithm, we simply index the input patterns and store the indices with each subgraph. As shown in Figure 1b, the size-3 subgraphs are obtained by matching the two size-3
patterns. We store the pattern_idx with each of its embeddings. When two subgraphs are combined, we construct a 4-tuple as the quick pattern for the combined subgraph. The first two elements in the 4-tuple are the pattern indices of the two joining subgraphs. The third element represents the position of the joining vertex in the two subgraphs. Suppose the two joining subgraphs $s_1$ and $s_2$ are of size $n_1$ and $n_2$. If the joining vertex is the $i$th vertex in $s_1$ and the $j$th vertex in $s_2$, then the value of the third element is $(i \times n_2 + j)$. The last element is a bitarray representing connections between the two subgraphs. If the $i$th vertex in $s_1$ is connected with the $j$th vertex in $s_2$, then the $(i \times n_2 + j)$th bit in the bitarray is set.

Example: In Figure 1c, the resulting subgraph '34275' is obtained by joining $s_1 = '342'$ and $s_2 = '375'$, and its quick pattern is $(0, 0, 32)$. The first two elements are the pattern index of '342' and '375'. The third element is 0 because the joining vertex is at position 0 in both subgraphs. The last element is 32 because the $s_1[1] = 4$ is connected with $s_2[2] = 5$ in the graph and the $(1 \times 3 + 2)$th bit is set in the bitarray. Similarly, the quick pattern of both '34278' and '35287' is $(0, 0, 128)$, and the quick pattern of '34285' is $(0, 0, 0, 272)$.

By encoding the sub-pattern information, our quick pattern achieves more accurate grouping of the subgraphs and thus reduces the canonical form computation. The computation is further reduced by multi-vertex exploration as larger subgraphs contain more accurate sub-pattern information. To see this point, let us consider the number of possible size-4 4-tuple as the quick pattern for the combined subgraph. The number of possible values of the last element in the 4-tuple is the number of possible joining positions, and 4 is the number of possible joining positions, and 4 is the number of possible values of the last element in the 4-tuple. In comparison, if we use the edge list as the quick pattern as in previous work [27, 30], the fully-connected size-4 graph alone has 624 (6! − 4 × 4!) possible quick patterns where 6! represents all possible permutations of the six edges and 4 × 4! represents the permutations that do not have adjacent edges. This indicates that our index-based technique has much fewer possible patterns compared with the technique used in previous work, leading to fewer groups for isomorphism check.

The quick pattern is computed after every combine function in Figure 2. If the combine function returns a valid subgraph, we compute its quick pattern and look for the quick pattern in a global dictionary. The dictionary keeps a mapping from quick patterns to their indices. If the quick pattern exists, we store its index with the subgraph. If a quick pattern is not found in the dictionary, we increase the global index number and insert a new pair of quick pattern and its index. In our implementation, we parallelize the for-loop that iterates over all keys in the first joining hash table. To avoid synchronization among vertices of the two joining subgraphs. For any size-3 subgraph 'abc' with 'a' being the joining vertex and 'b', 'c' from different subgraphs, if the subgraph is not in the list of frequent size-3 patterns, the combine function returns an empty set immediately.

4.5 Exploration Space Pruning

An optimization that most graph mining systems adopt for frequent subgraph mining is to filter out the subgraphs of infrequent patterns so as to reduce the subgraph exploration space [10, 15, 27, 30]. All of the existing systems achieve this optimization with breadth-first exploration. They either store all intermediate subgraphs (e.g., RStream [30], Pangolin [10]) or maintain a list of frequent patterns and re-match these patterns (e.g., Peregrine [15]) in each exploration step. The problem with the first approach is that it takes a lot of memory and needs to aggregate the subgraphs in each step. The problem with the second approach is that it needs to perform redundant matching in each step, and it only works for support measures that can be computed without storing all the embeddings (e.g., MNI). If the user wants to use more memory support measures (e.g., MIS, MVC [21]), the second approach will not work. An advantage of two-vertex exploration is that it enables exploration space pruning without storing intermediate results or re-matching.

Our main idea is that, instead of checking the support of the combined pattern, we check whether the vertices around the joining point form any subgraphs of smaller infrequent patterns. If an infrequent subgraph is found, then the combined subgraph must be infrequent and should be discarded. Figure 3 shows an example of this method. When the system tries to join two subgraph '012' and '234' at vertex 2, it finds that there is an edge connecting vertex 0 and 3 and an edge connecting vertex 1 and 4. This forms two triangles '023' and '124'. While triangle '124' is frequent, triangle '023' is not, according to the list of frequent size-3 patterns. Due to the anti-monotone property of the support measure, a frequent pattern cannot contain infrequent subpatterns. Thus, the combined subgraph '01234' must be infrequent and should not be used for further exploration.

The above pruning procedure is done in the combine function (line 9 in Algorithm 1) when we check the connectivity among vertices of the two joining subgraphs. For any size-3 subgraph 'abc' with 'a' being the joining vertex and 'b', 'c' from different subgraphs, if the subgraph is not in the list of frequent size-3 patterns, the combine function returns an empty set immediately.
5 Subgraph Sampling for Faster Exploration

In real applications, we may not need to find all frequent patterns, and exhaustive exploration is unnecessary [14]. Thus, we propose a subgraph sampling technique to accelerate the exploration process.

Sampling during Joining: The idea is to add a sampling operation each time we iterate over the joining subgraphs, i.e., before each for-loop in the dotted boxes in Figure 2. Because the MNI support measures the frequency of a pattern as the number of distinct matching vertices, we sample a fixed number of iterations in each of the boxed for-loops in Figure 2, in order to achieve a more even distribution of subgraphs over all vertices. If a loop has fewer iterations than the sampling threshold, we execute all of them; if a loop has more iterations than the threshold, we sample the iterations uniformly to the threshold number. This subgraph sampling during the joining phase can be considered as a generalization of the neighbor sampling technique in ASAP [14]. ASAP samples a subset of the edges when it extends the matched subgraph from one vertex to its neighbors. We sample the neighboring size-3 subgraphs instead. Intuitively, our subgraph sampling is more accurate than neighbor sampling because size-3 subgraphs preserve more graph structures than edges.

Sampling during Matching: For very large graphs, we may not be able to store all size-3 subgraphs in memory or even on disk. To achieve fast mining, we can sample the subgraphs during the matching phase and only store the sampled subgraphs. Similar to the sampling in the joining phase, we sample a fixed number of subgraphs around each vertex in order to have subgraphs evenly distributed over all vertices. More specifically, we permute the vertex list at each inner loop of the nested for-loop generated by AutoMine [17]. The execution continues to the next iteration of the outermost loop if $t$ subgraphs have been matched in the current iteration. This will give us $t$ subgraphs sampled from each vertex. We set $t$ to a number such that all the sampled subgraphs can be stored in memory. These sampled size-3 subgraphs are then given to the join procedure to explore larger subgraphs. This subgraph sampling during the matching phase can be considered as a generalization of the edge sampling technique for approximate graph processing [5, 33]. Previous work has shown that edge sampling does not work well for graph mining tasks [14]. Our subgraph sampling is much more robust than edge sampling for graph pattern mining as it preserves more structures of the graph. Our experiments also validate this point.

6 Experimental Results

This section presents our experimental setup and performance comparison with the existing graph mining systems and methods.
on MI) compared with the joining time, and 2) Step1 can be considered as preprocessing. We find that Peregrine and Pangolin abort for most tasks. In fact, Peregrine paper [15] only reports results of 3-FSM. Pangolin [10] 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 is 1.8x to 5.6x faster. 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 property (i.e., it does not run faster for larger support thresholds). Our system is 1.9x to 8.4x faster than AutoMine for these tasks.

**Advantage over Single-Vertex Exploration:** As discussed in Section 4.2, one advantage of two-vertex exploration over single-vertex exploration is that it reduces the memory access overhead in depth-first multi-way join. To show the advantage, we configure our system to run single-vertex exploration. The single-vertex version still uses our index-based quick patterns, but it does not support exploration pruning since the size-3 subgraphs are not computed. The execution times of single-vertex exploration are shown in Figure 4. We can see that single-vertex exploration is 1.02x to 1.52x slower than two-vertex exploration. We also collect the total memory access sizes to the hash tables with two-vertex exploration and single-vertex exploration (assuming every query to the hash tables is a cache miss). As shown in Figure 5, two-vertex exploration reduces the memory access overhead by 5x to 189x. The results are collected with support threshold 0.001n. Other support thresholds show a similar pattern.

**Benefit of Index-Based Quick Pattern:** To show the benefit of our index-based quick pattern technique, we disable our index-based quick pattern and use the quick pattern technique in previous work instead (i.e., a list of edges with labels of adjacent nodes). Figure 4 shows the execution times of two-vertex exploration without our index-based quick pattern. We can see that it leads to 1.75x to 2.78x slowdown. To further verify the advantage, we collect the number of invocations to the bliss function [16] for computing the canonical forms of subgraphs. As shown in Figure 6, our index-based quick pattern reduces the number of isomorphism checks by 31x to 564x for different tasks, which explains the speedups.

**6.3 Performance with Sampling**

Next, we evaluate the effectiveness of our sampling methods. Since all the size-3 subgraphs of CI and MI can be stored in memory, we only perform sampling during the joining phase for these two graphs.

Figure 7 shows the number of size-4 frequent patterns we can find on MI graph with different support thresholds and different sampling thresholds. The execution time of different runs are labeled on top of the bars. When the support threshold is set to 0.001n, there are 215025 frequent patterns in total, and to discover all these patterns precisely our system needs to run for 41645 seconds (as shown in Table 2). If we sample 10 edges and 100 size-3 subgraphs in each key group when we join the edge list and the size-3 subgraphs (ST10 in the figure), our two-vertex exploration returns 49% of the frequent patterns in 671 seconds. The execution time is reduced by 62x. When the support threshold is set to 0.005n, we can find 41% of the total frequent patterns within 524 seconds, which is 1/63 of the total execution time. When the support threshold is set to 0.05n, there are only 8 frequent patterns, and our two-vertex exploration with ST6 sampling can find 7 of them in 58 seconds, which leads to a 443x speedup compared with the accurate execution. The figure also shows that the larger sampling thresholds we use the more frequent patterns we can find.

Figure 8 shows the number of size-7 frequent patterns found on CI graph with different support thresholds and different sampling thresholds. When the support threshold is 0.001n, our two-vertex exploration can find 86% of the frequent patterns in 1284 seconds with ST10 sampling. Compared with the time of accurate execution in Table 2, sampling achieves a 21x speedup. When the support threshold is set to 0.005n and 0.01n, there are fewer frequent patterns, and our two-vertex exploration with ST10 sampling can find more than 90% of the frequent patterns with less than 1/21 of the total execution time. When the support threshold is 0.05n, there are only 22 frequent patterns, and our two-vertex exploration with ST6 sampling can find all of them within 170 seconds, which is 1/96 of the accurate execution time.
Figure 4. Performance of two-vertex exploration, single-vertex exploration, and two-vertex exploration without our index-based quick pattern. The Execution times are normalized for each task with the execution time of two-vertex exploration in Table 2.

Figure 5. Total memory access size in multi-way join with two-vertex and single-vertex exploration for different FSM tasks (MNI support threshold 0.001).

Figure 6. Number of isomorphism checks for different FSM tasks (MNI support threshold 0.001) with and without our index-based quick pattern technique.

Advantage over Single-Vertex Exploration: As discussed in Section 5, another advantage of two-vertex exploration over single-vertex exploration is that it leads to more accurate sampling. To verify this, we configure our system to run sampled single-vertex exploration. Since single-vertex exploration needs twice join steps as two-vertex exploration, we set its sampling threshold to the square root of the threshold for two-vertex exploration in order to achieve a similar size of overall exploration space. As shown in Figure 7 and 8, if we do not include the matching time, two-vertex exploration has a slightly shorter execution time than single-vertex exploration when they use the corresponding sampling thresholds. Even if we add the time for matching size-3 subgraphs (102 seconds on MI and 0.08 seconds on CI), the total execution time is close to that of single-vertex exploration. For 4-FSM on MI, two-vertex exploration finds 6% to 57% more frequent patterns than single-vertex exploration. For 7-FSM on CI, the number of frequent patterns found by two-vertex exploration is 1.18x to 4x that of single-vertex exploration.

Figure 7. Number of discovered size-4 frequent patterns on MI graph with different support thresholds and different sampling thresholds. For single-vertex exploration, STx means in every join step only x edges are sampled in each neighbor list. For two-vertex exploration, it means that x edges and $x^2$ size-3 subgraphs are sampled in each key group when we join the edge list with the size-3 subgraphs.

Figure 8. Number of discovered size-7 frequent patterns on CI graph with different support thresholds and different sampling thresholds. For single-vertex exploration, STx means in every join step only $x^2$ edges are sampled in each neighbor list. For two-vertex exploration, it means that in every join step only $x^2$ size-3 subgraphs are sampled in each key group.

Table 3 shows the number of size-9 frequent patterns found on CI graph with different support thresholds. We can see that with a similar execution time two-vertex exploration discovers much more frequent patterns than single-vertex
Table 3. Results of 9-FSM on CI graph with sampling threshold 0.001n and different sampling thresholds. 'M. ST' stands for Matching Sampling Threshold, 'M. Time' stands for Matching Time, 'J. ST' stands for Joining Sampling Threshold, 'J. Time' stands for Joining Time.

|                | M. ST | M. Time (sec) | J. ST | J. Time (sec) | # of Patterns |
|----------------|-------|---------------|-------|---------------|---------------|
| SV 0.001       | 119   | 4             | 175   | 18            | 1             |
| SV 0.005       | 119   | 4             | 207   | 20            | 2             |
| SV 0.01        | 120   | 4             | 268   | 20            | 2             |
| SV 0.05        | 122   | 4             | 299   | 20            | 2             |
| SV 0.1         | 1184  | 16            | 1651  | 18            | 1             |
| SV 0.5         | 600   | 16            | 105   | 20            | 1             |
| SV 1.0         | 601   | 16            | 16    | 20            | 2             |
| SV 5.0         | 602   | 16            | 16    | 20            | 2             |
| SV 10.0        | 603   | 16            | 16    | 20            | 2             |
| SV 100.0       | 604   | 16            | 16    | 20            | 2             |

Table 4. Results of 5-FSM on OK graph with support threshold 0.001n and different sampling thresholds. 'M. ST' stands for Matching Sampling Threshold, 'M. Time' stands for Matching Time, 'J. ST' stands for Joining Sampling Threshold, 'J. Time' stands for Joining Time.

|                | M. ST | M. Time (sec) | J. ST | J. Time (sec) | # of Patterns |
|----------------|-------|---------------|-------|---------------|---------------|
| SV 0.001       | 4     | 175           | 18    |               | 1             |
| SV 0.005       | 4     | 207           | 20    |               | 2             |
| SV 0.01        | 4     | 268           | 20    |               | 2             |
| SV 0.05        | 4     | 299           | 20    |               | 2             |
| SV 0.1         | 1184  | 16            | 1651  | 18            | 1             |
| SV 0.5         | 600   | 16            | 105   | 20            | 1             |
| SV 1.0         | 601   | 16            | 16    | 20            | 2             |
| SV 5.0         | 602   | 16            | 16    | 20            | 2             |
| SV 10.0        | 603   | 16            | 16    | 20            | 2             |
| SV 100.0       | 604   | 16            | 16    | 20            | 2             |

Table 5. Results of 5-FSM on UK and FR graph with different sampling thresholds and support thresholds. 'M. ST' stands for Matching Sampling Threshold, 'M. Time' stands for Matching Time, 'J. ST' stands for Joining Sampling Threshold, 'J. Time' stands for Joining Time.

|                | M. ST | M. Time (sec) | J. ST | J. Time (sec) | # of Patterns |
|----------------|-------|---------------|-------|---------------|---------------|
| UK 0.001       | 4     | 175           | 18    |               | 1             |
| UK 0.005       | 4     | 207           | 20    |               | 2             |
| UK 0.01        | 4     | 268           | 20    |               | 2             |
| UK 0.05        | 4     | 299           | 20    |               | 2             |
| UK 0.1         | 1184  | 16            | 1651  | 18            | 1             |
| UK 0.5         | 600   | 16            | 105   | 20            | 1             |
| UK 1.0         | 601   | 16            | 16    | 20            | 2             |
| UK 5.0         | 602   | 16            | 16    | 20            | 2             |
| UK 10.0        | 603   | 16            | 16    | 20            | 2             |
| UK 100.0       | 604   | 16            | 16    | 20            | 2             |

The results of 5-FSM on UK and FR graph are shown in Table 5. We can see that matching takes a large proportion of the total execution time. This is because there are a lot of size-3 subgraphs and patterns in these two large graphs. However, if we consider matching as preprocessing and store the sampled size-3 subgraphs in memory, the joining procedure is fast. As shown in Table 5a, our system can find frequent patterns on UK within a few minutes, and more patterns can be found by using larger joining sampling thresholds. Table 5b shows the results of 5-FSM on FR graph. Again, the matching procedure is expensive. Once the size-3 subgraph are sampled, we can find size-5 frequent patterns in a relatively short time with sampled join. For comparison, we also run single-vertex exploration with sampling threshold 2 on these two graphs. It cannot finish execution within 24 hours, so we print out the found patterns after 24 hours of execution. For UK, it returns 5 frequent patterns when support threshold is set to 0.0001n, and 0 frequent pattern when support threshold is 0.0005n. For FR, single-vertex exploration with sampling threshold 2 cannot return any frequent pattern within 24 hours.

7 Related Work

This section summarizes the graph pattern mining systems that are most related to our work.

Exploration-based Systems: Arabesque [27] is a distributed graph pattern mining system. It enumerates all possible embeddings in multiple rounds and uses a filter-process model to generate the results. It first propose the quick pattern technique for reducing isomorphism checks. RStream [30] is...
Approximate Pattern Mining:

which leads to redundant computation. DwarvesGraph [9] is
step, they ensure that patterns of higher supports are more
it calculates a probability distribution of all candidate pat-
all patterns. Every time it walks from one pattern to another,
it exists in more than a certain amount of graphs. The main
idea of these works is to perform random walk in the space of
patterns based on the subgraphs it has explored and main-
of enumerating all the patterns before matching, it discovers
cult. Peregrine [15] is another pattern-based system. Instead
size is more than 7, enumerating the patterns becomes diffi-
cides patterns at each node, they can be applied to the exploration-based systems. We fill this gap and show that FSM can be accelerated by sampling the subgraphs in each key group of the join operation.

8 Conclusion

In this work, we propose a novel two-vertex exploration method to accelerate frequent subgraph mining. Based on two-vertex exploration, we further improve the performance through an index-based quick pattern technique and sub-
graph sampling. The experiments show that our method outperforms other state-of-the-art graph mining systems for FSM on various input graphs and pattern sizes.

References

[1] [n.d.]. Dataset for “Statistics and Social Network of YouTube Videos”. http://netsg.cs.sfu.ca/youtubedata/
[2] [n.d.]. Number of Graphs on n unlabelled vertices. http://garsia.math.yorku.ca/~zabrocki/math3260w03/nall.html
[3] [n.d.]. Orkut social network. http://snap.stanford.edu/data/com-Orkut.html
[4] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Faud Jamour. 2016. 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. IEEE, 716–727.
[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In Proceedings of the 8th ACM European Conference on Computer Systems. 29–42.
[6] Mohammad Al Hasan and Mohammed J Zaki. 2009. Output space sampling for graph patterns. Proceedings of the VLDB Endowment 2, 1 (2009), 730–741.
[7] László Babai, William M Kantor, and Eugene M Luks. 1983. Computational complexity and the classification of finite simple groups. In 24th Annual Symposium on Foundations of Computer Science (Sfcs 1983). IEEE, 162–171.
[8] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In Proceedings of the Thirteenth EuroSys Conference. 1–12.
[9] Jingji Chen and Xuehai Qian. 2020. DwarvesGraph: A High-Performance Graph Mining System with Pattern Decomposition. arXiv:2008.09682 [cs.DC]
[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. Proc. VLDB Endow. 13, 8 (April 2020), 1190–1205. https://doi.org/10.14778/3389113.3389137
[11] Wei-Ta Chu and Ming-Hung Tsai. 2012. Visual Pattern Discovery for Architecture Image Classification and Product Image Search. In Proceedings of the 2nd ACM International Conference on Multimedia Retrieval (Hong Kong, China) (ICMR ’12). Association for Computing Machinery, New York, NY, USA, Article 27, 8 pages. https://doi.org/10.1145/2324796.2324831
[12] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In Proceedings of the 2019 International Conference on Management of Data (Amsterdam, Netherlands) (SIGMOD ’19). Association for Computing Machinery, New York, NY, USA, 1357–1374. https://doi.org/10.1145/3299869.3319875

[13] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panois Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. Proc. VLDB Endow. 7, 7 (March 2014), 517–528. https://doi.org/10.14778/2732286.2732289

[14] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA, 745–761. https://www.usenix.org/conference/osdi18/presentation/iyer

[15] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A system for distributed graph mining. In Proceedings of the 25th Symposium on Operating Systems Principles. 425–440.

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

[21] Jinghan Meng and Yi-cheng Tu. 2017. Flexible and Feasible Support Measures for Mining Frequent Patterns in Large Labeled Graphs. In Proceedings of the 2017 ACM International Conference on Management of Data (Chicago, Illinois, USA) (SIGMOD ’17). Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/3039183.3039366

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

[23] Aduri Pavan, Srikantha Tirthapura, et al. 2013. Counting and sampling triangles from a graph stream. (2013).

[24] Tanay Kumar Saha and Mohammad Al Hasan. 2015. FS3: A sampling based method for top-k frequent subgraph mining. Statistical Analysis and Data Mining: The ASA Data Science Journal 8, 4 (2015), 245–261.

[25] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Veriﬁcation Hardness: An Efﬁcient Algorithm for Testing Subgraph Isomorphism. Proc. VLDB Endow. 1, 1 (Aug, 2008), 364–375. https://doi.org/10.14778/1453856.1453899

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

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

[28] Johan Ugander, Lars Backstrom, and Jon Kleinberg. 2013. Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections. In Proceedings of the 22nd International Conference on World Wide Web (Rio de Janeiro, Brazil) (WWW ’13). Association for Computing Machinery, New York, NY, USA, 1307–1318. https://doi.org/10.1145/2488388.2488502

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

[30] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. 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). 763–782.

[31] XiFeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In 2002 IEEE International Conference on Data Mining, 2002. Proceedings. 721–724.

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

[33] Ruoyu Zou and Lawrence B Holder. 2010. Frequent subgraph mining on a single large graph using sampling techniques. In Proceedings of the eighth workshop on mining and learning with graphs. 171–178.