Efficient Top-k Frequent Subgraph Mining Using Tight Upper and Lower Bounds

Seonho Lee
Seoul National University
shlee2@theory.snu.ac.kr

Yeunjun Lee
Seoul National University
yjlee@theory.snu.ac.kr

Kunsoo Park
Seoul National University
kpark@theory.snu.ac.kr

ABSTRACT

Frequent subgraph mining is an important and well-studied problem with numerous applications such as the prediction of protein functionalities and graph indexing. Many studies use the minimum-image-based support (MNI) to measure the frequency of subgraphs in single graph mining. Given a graph G and an integer k, top-k frequent subgraph mining is to find top-k frequent subgraphs in the graph G based on MNI. However, there are two main challenges in top-k frequent subgraph mining. (1) Computing MNI is time-consuming. (2) The number of subgraphs for which MNI should be computed is large. In this paper, we propose a novel algorithm `Minting` to address these challenges. We propose a method to significantly reduce the number of subgraphs for which MNI computation is required by using a tight upper bound of the MNI value. We also improve the computation of MNI itself by utilizing both a lower bound and an upper bound of the MNI value. Experiments shows that our algorithm outperforms the state-of-the-art algorithms by up to three orders of magnitude in terms of the elapsed time. Our algorithm is also a feasible solution for this challenging problem, even for large k.

PVLDB Reference Format:
Seonho Lee, Yeunjun Lee, and Kunsoo Park. Efficient Top-k Frequent Subgraph Mining Using Tight Upper and Lower Bounds. PVLDB, 18(3): 557 - 570, 2024.
doi:10.14778/3712221.3712225

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SNUCSE-CTA/Minting.

1 INTRODUCTION

Graphs can model complex relationships between objects and they are widely used in many fields such as bioinformatics, social networks, and chemistry. Mining frequent subgraphs is an important and well-studied problem, which has numerous applications, including the prediction of protein functionalities in computational biology [13, 37, 45], graph indexing [61], classification [16], clustering [21], and recommender systems [6].

Frequent subgraph mining can be categorized into two types: transactional mining, which focuses on mining in a graph database (of typically small graphs), and single graph mining, which is to find frequent subgraphs within a single large graph [28, 30]. Between these two types, single graph mining is a generalized version of transactional mining, as a set of small graphs can be viewed as components of a large graph. Additionally, single graph mining is more challenging because multiple instances of identical subgraphs may overlap [17].

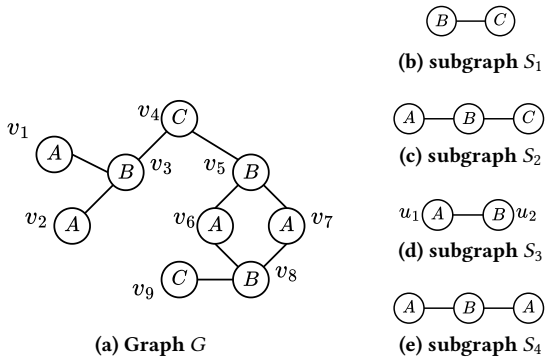


Figure 1: A graph G and its subgraphs S_1, S_2, S_3, S_4. Top-k frequent subgraphs in G when $k = 2$ are S_3 and S_4.

Recently, considerable research has been done on frequent subgraph mining in a single large graph. Many studies [17, 26, 64] use the minimum-image-based support (called MNI) [9] to measure the frequency of subgraphs instead of counting the number of isomorphisms. The image set of a vertex u in a subgraph S is the set of vertices in the given graph G that u is mapped to. The MNI of a subgraph S is the size of the smallest image set among the vertices of S. Our work also uses MNI to measure subgraph frequency. In this paper, we deal with the top-k frequent subgraph mining problem: Given a graph G and an integer k, top-k frequent subgraph mining is to find top-k frequent subgraphs in the graph G based on MNI. Given a data graph G in Figure 1 and $k = 2$, the result of top-2 frequent subgraph mining is the subgraphs S_3 and S_4 in Figure 1.

Existing Works and Challenges. GRAMI [17], Peregrine [26], and FastPat [64, 65] address problems related to top-k frequent subgraph mining. GRAMI finds subgraphs whose frequency is greater than or equal to a user-defined frequency threshold τ. Peregrine takes a threshold τ and an integer m as input, and finds subgraphs with m edges that have an MNI greater than or equal to τ. Thus both of them require some prior knowledge to set the frequency threshold τ. FastPat finds top-k frequent patterns extended from an input core pattern in a knowledge graph. It specifically targets knowledge graphs and needs a core pattern as an input.

The framework used in many frequent subgraph mining algorithms consists of subgraph generation and computing MNI [17, 59,

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712225

64]. Subgraphs of the given graph are generated through an edge-growing method [59]. MNI is computed for the subgraphs to determine whether each subgraph is frequent.

There are two main challenges in top-k frequent subgraph mining. The first challenge is the computation of MNI, which is time-consuming because it is an NP-hard problem [17, 20, 29]. Thus, it is a bottleneck in frequent subgraph mining. Although GRAMI employs constraint satisfaction problems and FastPat utilizes join operations for computing MNI, they still take a considerable amount of time for MNI computation, thus occupying a major portion of the time in finding frequent subgraphs.

The second challenge is the large number of subgraphs for which MNI should be computed. For any given graph, the number of subgraphs can be exponentially large, leading to a huge search space requiring exploration [42, 64]. Since the computation of MNI is a time-consuming process, the large number of subgraphs for which MNI computation is required leads to an exceedingly large amount of time to find frequent subgraphs. Although FastPat employs an upper bound in its process, this bound is not tight, still resulting in the generation of a large number of subgraphs. Therefore, frequent subgraph mining is a doubly hard problem, both in generating subgraphs of the given graph in the form of a lattice [28] and in computing MNI for the subgraphs.

Contributions. In this paper, we propose a novel algorithm `Minting` (**Min**ing **t**op-k patterns **in g**raph) to address the aforementioned challenges. The contributions of our work are as follows.

(1) We introduce a new data structure called *marked CS (Candidate Space)* where each candidate vertex is marked as *confirmed*, *invalid*, or *undetermined*. A candidate vertex is called *valid* if it is confirmed or undetermined. Based on this data structure, we propose key concepts *minVL*(S) and *minCF*(S).
   - For a subgraph S of the given graph, the *valid candidate set* $VL(u)$ for each vertex u of S is the set of valid candidates for u in the marked CS. We define *minVL*(S) to be the minimum size among valid candidate sets, and prove that minVL(S) is an upper bound of the MNI value of S, denoted by MNI(S).
   - For a subgraph S of the given graph, the *confirmed candidate set* $CF(u)$ for each vertex u of S is the set of confirmed candidates for u in the marked CS. We define *minCF*(S) to be the minimum size among confirmed candidate sets, and prove that minCF(S) is a lower bound of MNI(S).

(2) We reduce the number of subgraphs for which MNI computation is required. We do this using the upper bound minVL(S) and a filtering algorithm on the marked CS. The filtering algorithm repeatedly selects (using a queue) a candidate vertex which is newly marked as invalid and checks whether the candidates adjacent to it satisfy safety conditions. The safety conditions used here are connectivity-safety and neighbor-safety. If a candidate becomes invalid by violating one of safety conditions, it is inserted into the queue. Since a candidate newly marked as invalid can cause adjacent candidates to become invalid, our method is more effective in filtering candidates than DAG-DP (a filtering technique in DAF and VEQ [22, 29]) which proceeds top-down and bottom-up on a DAG, thus checking unnecessarily all parts of the marked CS repeatedly.
   As candidates in the marked CS are filtered out, minVL(S) decreases. As soon as minVL(S) $\leq \tau$ (smallest MNI value among the current top-k subgraphs), subgraph S cannot be one of top-k results, and so it is pruned out without computing its MNI value. By this method, we significantly reduce the number of subgraphs for which MNI computation is required.

(3) We improve the computation of MNI itself. We do this by a novel algorithm for MNI computation utilizing both lower bound minCF(S) and upper bound minVL(S). If we find an embedding of S in G that maps a vertex u of S to a vertex v of G, $|CF(u)|$ increases, and thus minCF(S) may increase. If there is no such embedding, $v \in C(u)$ becomes invalid. We apply our filtering algorithm from the invalid candidate, which can make other undetermined candidates invalid. Thus minVL(S) can decrease. When the two bounds converge (i.e., are the same), the same value *is* MNI(S). This method significantly reduces the number of subgraph isomorphism checks (i.e., finding an embedding of S in G), making the computation of MNI efficient.
   Additionally, we have improved the process of checking whether a DFScode [59] is canonical for certain subgraphs. This improvement reduces the time required from a potentially exponential to constant.

(4) We conduct extensive experiments with the state-of-the-art algorithms on six real-world datasets. When contributions (2) and (3) are combined together, they produced significant performance improvements in our experiments. Experiments shows that our algorithm outperforms the existing algorithms by up to three orders of magnitude in terms of elapsed time. Our algorithm is also a feasible solution for this challenging problem, top-k frequent subgraph mining, even for large k.

2 PRELIMINARIES

In this paper, we focus on undirected simple graphs with labeled vertices. Our techniques can be extended to directed and edge-labeled graphs. A graph $G = (V(G), E(G), L_G)$ consists of a set of vertices $V(G)$, a set of edges $E(G)$, and a labeling function $L_G : V(G) \rightarrow \Sigma$ that assigns labels to vertices where Σ is a set of labels. A graph is *non-trivial* if a graph has at least one edge.

Definition 2.1. For a graph $S = (V(S), E(S), L_S)$ and a graph $G = (V(G), E(G), L_G)$, an embedding of S in G is an injective function $f : V(S) \rightarrow V(G)$ satisfying (i) $L_S(u) = L_G(f(u))$ for all vertices $u \in V(S)$, and (ii) $(f(u), f(u')) \in E(G)$ for all edges $(u, u') \in E(S)$.

2.1 Problem Statement

Frequent subgraph mining is the problem of finding subgraphs with a large support in a graph, where the support refers to how frequently a subgraph appears in the graph. The most straightforward way to measure the support of a subgraph is to count its embeddings. For instance, in the graph shown in Figure 1, the number of embeddings of the subgraph S_1 and its extension S_2 are 3 and 6, respectively. A support is *anti-monotone* if the support of a graph G is always less than or equal to the support of any subgraph G' of G. However, the number of embeddings does not satisfy the anti-monotone property because the support of subgraph S_2 is greater than the support of its subgraph S_1. The anti-monotone property is crucial in frequent subgraph mining because it allows pruning

of the search space: If a subgraph is infrequent, any subgraph extended from that subgraph will also be infrequent [18]. There are several supports satisfying the anti-monotone property, such as minimum-image-based support [9], minimum instance [35], and maximum independent set [30]. In this paper, we use the minimum-image-based support (called MNI) as the support since MNI is commonly used in many previous studies [17, 64, 65]. The problem of computing MNI is NP-hard because subgraph isomorphism, which is NP-complete [20], can be reduced to it in polynomial time.

Definition 2.2. Let $f_1, f_2, \ldots, f_n$ be the set of embeddings of a subgraph S in a graph G. For a vertex $u \in V(S)$, the *image set* of u, denoted by $F(u)$, is the set that contains the vertices v in G such that a function f_i maps the vertex u to v.

Definition 2.3. The *minimum-image-based support* (MNI) of S in G, denoted by $\text{MNI}(S)$, is defined as $\min_{u \in V(S)} |F(u)|$.

Example 2.4. For a graph G and a subgraph S_3 in Figure 1, there are 6 embeddings of S_3 in G. For the vertices u_1, u_2 of S_3, we have their image sets as $F(u_1) = \{v_1, v_2, v_6, v_7\}$ and $F(u_2) = \{v_3, v_5, v_8\}$. So, $\text{MNI}(S_3)$ in G is $\min\{4, 3\} = 3$. Similarly, $\text{MNI}(S_1)$ is $\min\{3, 2\} = 2$, $\text{MNI}(S_2)$ is $\min\{4, 3, 2\} = 2$, and $\text{MNI}(S_4)$ is $\min\{4, 3, 4\} = 3$.

Definition 2.5. Given a graph G and an integer k, the top-k frequent subgraph mining problem is to find a set $A = \{S_1, S_2, \ldots, S_k\}$ of non-trivial, connected subgraphs of G such that $\text{MNI}(S_i)$ for any $1 \leq i \leq k$ is larger than or equal to $\text{MNI}(S')$ for any other non-trivial, connected subgraph S' of G that is not in A.

2.2 Related Work

Transactional mining focuses on mining frequent subgraphs in a database of many small graphs, typically mining subgraphs whose support is greater than or equal to a user-defined threshold τ. Numerous solutions [24, 53, 59] have been proposed for this problem. gSpan [59] introduced the DFS canonical form, which enables the generation of subgraphs without generating duplicates. For approximate frequent subgraph mining, algorithms such as APGM [27] and REAFUM [33] have been developed. In distributed settings, PrefixFPM [57] and PFSM [55] are proposed for parallel solutions. MARGIN [51] focuses on identifying maximal frequent subgraphs, while CloseGraph [60] targets closed frequent subgraphs. Beyond frequent subgraph mining, various studies conduct on different subgraph mining. LEAP [58] and GraphSig [44] are dedicated to finding significant patterns. RESLING [39] finds representative subgraph patterns. Additionally, TKG [19] returns the top-k frequent subgraphs in a graph database.

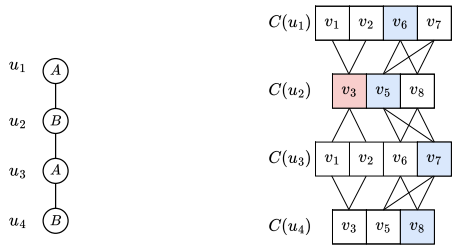
Single graph mining focuses on finding frequent subgraphs in a single large graph. Many works in single graph mining use the MNI as the support metric. Several algorithms [17, 30, 40] have been proposed for mining subgraphs in a single large graph whose support is greater than or equal to a user-defined threshold τ. GRAMI [17] uses a constraint satisfaction problem to compute MNI. For approximate frequent subgraph mining, algorithms such as AGRAMI [17] and MANIACS [43] have been developed. In distributed settings, DISTGRAPH [49] offers solutions for very large graphs that are too large to fit in memory. ScaleMine [4] provides a parallel frequent mining system. For dynamically changing graphs, TIPTAP [38] and IncGM+ [5] have been developed. WeGrami [31] specializes in weighted subgraphs, and fanta [12] finds frequent subgraphs in uncertain graphs. CSM-E [42] targets finding correlated subgraphs in a single graph. FastPat [64, 65] finds top-k frequent patterns extended from an input core graph in knowledge graphs, dealing with directed graphs and requiring a core graph. FastPat employs a combination of meta-indexing and bounds to selectively prune out patterns. It also employs a two-pass join method for the computation of MNI. GsFSM [67] addresses geo-social frequent pattern mining in geosocial networks, considering spatial and label constraints along with core patterns. APRTOPK [56] finds k frequent patterns that maximize an interestingness value.

In system research related to graph mining, several systems have been developed, including RStream [54], Arabesque [50], ASAP [25], G-miner [10], Sandslash [11], and Peregrine [26]. These systems are designed to support a wide range of graph mining problems, such as clique detection, motif finding, and frequent subgraph mining.

Frequent subgraph mining is closely related to subgraph matching because subgraph matching is used to compute MNI. Extensive research has been conducted on subgraph matching [7, 8, 14, 22, 23, 29, 36, 41, 46–48, 62] . Many practical solutions, such as DAF [22], VEQ [29], BICE [14], and GuP [7], are based on Ullmann's backtracking framework [52]. They adopt a filtering-backtracking approach, reducing the candidate set size in the filtering phase and employing techniques to prune the search space during backtracking. Some algorithms [36, 48] utilize a join-based framework.

Table 1: Frequently Used Notations

Symbol	Definition
G, S	Graph and subgraph
$V(G), E(G), L_G$	Vertices, edges, and labels of a graph G
$N_G(u)$	Neighbors of u in a graph G
$C(u)$	Set of candidate vertices for u
$C(u_n \mid u, v)$	Set of candidate neighbors of (u, v) to u_n
$VL(u)$	Set of *valid* candidates in $C(u)$
$CF(u)$	Set of *confirmed* candidates in $C(u)$



(a) subgraph S **(b) Marked CS**

Figure 2: A subgraph S, and the marked CS (blue for confirmed vertices, red for invalid, and white for undetermined) on S and G from Figure 1, after finding an embedding $\{(u_1, v_6), (u_2, v_5), (u_3, v_7), (u_4, v_8)\}$ and determining that there is no embedding mapping u_2 to v_3.

3 OVERVIEW OF OUR ALGORITHM

In this section, we introduce an auxiliary data structure called the marked CS and outline our top-k frequent subgraph mining algorithm.

Algorithm 1: top-k Frequent Subgraph Mining

input : A single graph G, an integer k
output : top-k frequent subgraphs based on MNI

1 $FreqEdge \leftarrow$ Compute frequent edges
2 Initialize H_{ans} with $FreqEdge$
3 **if** $FreqEdge.size() \geq k$ **then**
4 $\tau \leftarrow k$-th largest MNI in $FreqEdge$
5 **else**
6 $\tau \leftarrow 0$
7 **foreach** $e \in FreqEdge$ **do**
8 $Extension(e, e.\text{CS})$
9 **while** $H_{cand} \neq \emptyset$ *and* $H_{cand}.top.ub > \tau$ **do**
10 $S, \text{CS} \leftarrow H_{cand}.pop()$
11 $(\text{MNI}(S), \text{CS}) \leftarrow ComputeMNI(S, \text{CS})$
12 **if** $\text{MNI}(S) > \tau$ **then**
13 Update H_{ans} by $(S, \text{MNI}(S))$
14 Update τ
15 $Extension(S, \text{CS})$
16 **return** H_{ans}

3.1 Marked CS

The candidate space (CS) is an auxiliary data structure used to solve the subgraph matching problem, proposed by [22] and extended by [29]. The candidate space comprises the set of candidates along with edges between them, ensuring that all embeddings are preserved within CS. We extend this definition of CS by adding marking information to compute the MNI of a subgraph quickly.

Definition 3.1. A marked CS on a subgraph S and a graph G consists of the candidate set $C(u)$ with marking information for each vertex $u \in V(S)$ and edges between the candidates as follows:

- For each $u \in V(S)$, there is a candidate set $C(u)$, which is a set of vertices in G that u can be mapped to. A vertex $v \in C(u)$ can have one of the following three states:
  - **confirmed** if v is in the image set $F(u)$;
  - **invalid** if v is not in the image set $F(u)$;
  - **undetermined** if it is not determined whether v is in the image set $F(u)$ or not.
- There is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E(S)$ and $(v, v') \in E(G)$.

Example 3.2. Consider the graph G in Figure 1 and the subgraph S in Figure 2a. After finding an embedding $\{(u_1, v_6), (u_2, v_5), (u_3, v_7), (u_4, v_8)\}$ and finding that there is no embedding in which u_2 is mapped to v_3, the marked CS on G and S is shown in Figure 2b. Four candidates v_1, v_2, v_6, v_7 are in $C(u_1)$, and there is an edge between v_6 in $C(u_1)$ and v_5 in $C(u_2)$. The vertex v_3 in $C(u_2)$ is marked red (invalid) because there is no embedding that maps u_2 to v_3. The vertices v_6 in $C(u_1)$, v_5 in $C(u_2)$, v_7 in $C(u_3)$ and v_8 in $C(u_4)$ are marked blue (confirmed) because there is an embedding $\{(u_1, v_6), (u_2, v_5), (u_3, v_7), (u_4, v_8)\}$. All remaining candidates are marked white (undetermined).

Definition 3.3. For a vertex u in $V(S)$ and candidate v in $C(u)$, the candidate v in $C(u)$ is *valid* if the state of the candidate v is confirmed or undetermined.

Algorithm 2: Extension

input : A subgraph S, a marked CS of S

1 **foreach** $u \in V(S)$ *and* $e = (u, u') \in FreqEdge$ **do**
2 **if** $\text{MNI}(e) > \tau$ **then**
3 Let S' be the extension of S with the new edge e
4 Build CS′ from CS
5 $(isCandGraph, \text{CS}') \leftarrow CSnodeFiltering(S', \text{CS}', e)$
6 **if** $isCandGraph$ **then**
7 $UB' \leftarrow \text{minVL}(S')$
8 $H_{cand}.\text{insert}(S', \text{CS}', UB')$

Definition 3.4. For $v \in C(u)$ and $u_n \in N_S(u)$, the *candidate neighbors* of (u, v) to u_n, denoted by $C(u_n \mid u, v)$, is defined as the set of candidate $v_n \in C(u_n)$ that is adjacent to $v \in C(u)$.

Example 3.5. For the marked CS in Figure 2b, $C(u_3 \mid u_2, v_5)$ is $\{v_6, v_7\}$ and $C(u_2 \mid u_3, v_2)$ is $\{v_3\}$.

Definition 3.6. For a subgraph S of G and a marked CS on S and G, the *valid candidate set* $VL(u)$ is the set of valid candidates in $C(u)$ for each vertex $u \in V(S)$.

Definition 3.7. For a subgraph S of G, $minVL(S)$ is the minimum size of *valid candidates set* $VL(u)$ among $u \in V(S)$, i.e., $\min_{u \in V(S)} |VL(u)|$.

Theorem 3.8. *For a subgraph S of G, $minVL(S)$ is an upper bound of* $\text{MNI}(S)$.

Proof. By definition of the marked CS, all embeddings of S in G are in the marked CS. Furthermore, each embedding consists of only valid candidates. It means that for every $u \in V(S)$, the image set $F(u)$ is a subset of $VL(u)$, i.e., $|F(u)| \leq |VL(u)|$. Hence $\min_{u \in V(S)} |F(u)| \leq \min_{u \in V(S)} |VL(u)| = \text{minVL}(S)$. Since $\text{MNI}(S)$ is $\min_{u \in V(S)} |F(u)|$, $\text{minVL}(S)$ is an upper bound of $\text{MNI}(S)$. □

3.2 Top-k frequent subgraph mining

Algorithm 1 shows the overview of our algorithm that outputs the top-k frequent subgraphs in a given graph G. It follows the general framework to solve top-k frequent subgraph mining [19, 64, 65]. On top of this framework, our algorithm reduces the number of subgraphs for which MNI computation is required by a filtering process, and uses both lower and upper bounds of MNI to compute the MNI of a subgraph efficiently. The framework uses two data structures: a min-heap H_{ans}, and a max-heap H_{cand}. The min-heap H_{ans} keeps the current top-k subgraphs along with their MNIs, while the max-heap H_{cand} holds subgraphs for which MNI computation is needed, along with the marked CS and the MNI upper bound (i.e., minVL). The algorithm starts by computing the top-k frequent edges based on their MNIs to form the set of frequent edges, $FreqEdge$ (line 1). The top-k frequent edges become the current top-k results, and the subgraphs extended from these edges can become subgraphs for which MNI computation is needed. So, H_{ans} is initialized using the top-k frequent edges from $FreqEdge$ (line 2). τ is the smallest MNI value among the current top-k results, and it is 0 if the size of $FreqEdge$ is less than k (lines 3-6). H_{cand} is initialized by adding subgraphs extended from the edges in $FreqEdge$ via *Extension*. *Extension* takes a subgraph and its marked CS as inputs,

Algorithm 3: Filtering

input : A subgraph S, a marked CS of S, a new edge $e = (u, u')$
output : (whether S is a subgraph for which MNI computation is needed, refined CS)

1 $I \leftarrow ComputeInitialSet(S, \text{CS}, e)$
2 $(isCandGraph, \text{CS}) \leftarrow CSnodeFiltering(S, \text{CS}, I)$
3 **return** $(isCandGraph, \text{CS})$

generates extended subgraphs, and adds to the max-heap H_{cand} the subgraphs for MNI computation is needed (lines 7-8).

We pop a subgraph S with the largest MNI upper bound (i.e., minVL(S)) from H_{cand} (line 10). We find the embeddings of S in G and mark the candidates in the marked CS. Using this marking information, we compute MNI(S) (line 11). If this MNI(S) is greater than τ, then the subgraph S becomes one of the current top-k results. Thus, the subgraph S is added to H_{ans}, and the value of τ is updated. Additionally, new subgraphs are generated by adding an edge to the subgraph S, and subgraphs for which MNI computation is needed are added to H_{cand} via *Extension* (lines 12-15). This iteration repeats until either H_{cand} is empty or the largest MNI upper bound of subgraphs in H_{cand} is less or equal to τ.

Algorithm 2 shows *Extension*, which is the process of generating extensions of a subgraph S using the edge-growing method and adding to the max-heap H_{cand} the subgraphs for which MNI computation is needed. The edge-growing method, frequently used in prior studies [17, 59, 64, 65], extends a subgraph by adding an edge. Instead of every edge, the edges added to the subgraph S are from $FreqEdge$ and have an MNI value greater than τ (lines 1-2). This is due to the anti-monotone property of MNI. If the MNI of the edge is not greater than τ, then the MNI of the extended subgraph is also not greater than τ, which is the smallest MNI value of the current top-k subgraphs. We employ gSpan's DFScode canonical form [59] to prevent generating duplicate subgraphs.

For each subgraph S' extended from the subgraph S, we build a marked CS on the extended subgraph S' and G. To build the marked CS on S' and G, the candidates are copied from the marked CS on the subgraph S and G excluding invalid candidates. Subsequently, all marking information is initialized to undetermined. If a new vertex u' is created, the candidate set $C(u')$ is initialized as the set of vertices in graph G that have the same label as u. In addition, the neighborhood label frequency (NLF) filter [23] is applied (line 4). After the marked CS on S' and G is constructed, a filtering process computes a tight upper bound of MNI(S') and outputs whether the subgraph is a subgraph for which MNI computation is needed, along with refined marked CS (line 5). If the subgraph S' needs MNI computation, then S' is added to the max heap H_{cand} (lines 6-8).

4 REDUCING THE NUMBER OF SUBGRAPHS

In this section, we propose a method to reduce the number of subgraphs for which MNI computation is required. We present a filtering algorithm on the marked CS to make minVL(S) a tight upper bound. Additionally, we describe two conditions, connectivity-safety and neighbor-safety, which are used in the filtering process.

Computing a tight MNI upper bound reduces the number of subgraphs for which MNI should be computed. For example, consider

Algorithm 4: CSnodeFiltering

input : A subgraph S, a marked CS of S, an initial set I
output : (**false**, refined marked CS) **if** $\exists\, u$ s.t. $|VL(u)| \leq \tau$
(**true**, refined marked CS) **otherwise**

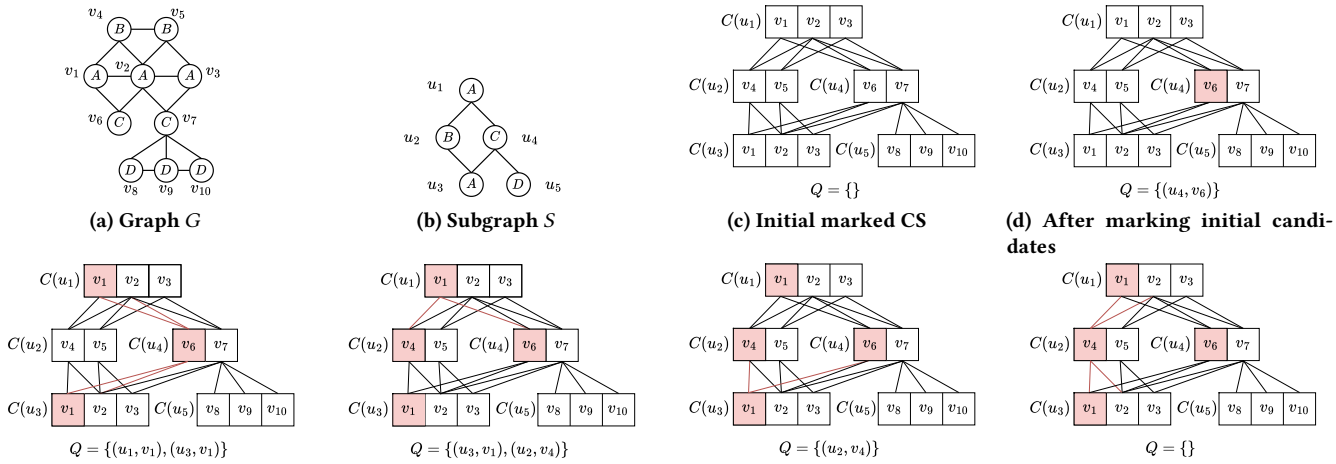
1 **for** $(u, v) \in I$ **do**
2 mark $v \in C(u)$ as invalid
3 **if** $|VL(u)| \leq \tau$ **then**
4 **return** (false, CS)
5 $Q.insert(u, v)$
6 **while** Q *is not empty* **do**
7 $(u, v) \leftarrow Q.pop$
8 **for** *each pair* (u_n, v_n) *adjacent to* (u, v) **do**
9 **if** $v_n \in C(u_n)$ *is marked as* invalid **then**
10 **continue**
11 $violated \leftarrow false$
 `// update and check connectivity-safety`
12 $NbrCnt(u_n, v_n, u)$−−
13 **if** $NbrCnt(u_n, v_n, u) = 0$ **then**
14 $violated \leftarrow true$
15 **else**
 `// update and check neighbor-safety`
16 Update $Nbr_{CS}(u_n, v_n, L_S(u))$
17 **if** neighbor-safety is violated **then**
18 $violated \leftarrow true$
19 **if** $violated$ *is true* **then**
20 Mark $v_n \in C(u_n)$ as invalid
21 **if** $|VL(u_n)| \leq \tau$ **then**
22 **return** (false, CS)
23 $Q.insert(u_n, v_n)$
24 **return** (true, CS)

the graph G in Figure 3a and the subgraph S in Figure 3b, with τ set to 1. Without the filtering process, the marked CS on S and G is shown in Figure 3c. The MNI upper bound, minVL(S), is computed as 2, which exceeds τ. Thus, we compute MNI(S). However, since MNI(S) is 1, subgraph S cannot be added to the top-k results. In contrast, when we apply the filtering process, the resulting marked CS is shown in Figure 3h and the subgraph has an MNI upper bound of 1. Since this is equal to τ, we can ignore the subgraph without computing its MNI.

To obtain a tight MNI upper bound, we use a filtering process to get a compact valid candidate set. Following are the two conditions used in the filtering process. One is connectivity-safety condition and the other is neighbor-safety condition which is proposed in VEQ [29].

Definition 4.1. For vertex $u \in V(S)$, $v \in C(u)$, and its neighbor $u_n \in N_S(u)$, $NbrCnt(u, v, u_n)$ is the number of valid candidates in the candidate neighbor set $C(u_n|u, v)$, i.e., $|\{v_n | v_n \in C(u_n|u, v)$ and $v_n \in C(u_n)$ is $valid\}|$.

Definition 4.2. Given a subgraph S of G and a marked CS on S and G, $v \in C(u)$ is *connectivity-safe* if $NbrCnt(u, v, u_n) > 0$ for every neighbor vertex $u_n \in N_S(u)$.



(a) Graph G **(b) Subgraph S** **(c) Initial marked CS** **(d) After marking initial candidates**

(e) After the first iteration **(f) After the second iteration** **(g) After the third iteration** **(h) Upon completion of filtering process**

Figure 3: Filtering process for a given graph G and a subgraph S created by adding a new edge (u_4, u_5). Among the candidates for u_4 and u_5, candidate $v_6 \in C(u_4)$ is marked as invalid for violating connectivity-safety. Then, $v_1 \in C(u_1)$, $v_1 \in C(u_3)$, and $v_4 \in C(u_2)$ are marked as invalid in this order.

Definition 4.3. For each vertex $u \in V(S)$ and a label $l \in \Sigma$, a neighbor set $Nbr_S(u, l)$ is the set of neighbors of u labeled with l. For each vertex $v \in C(u)$ and a label $l \in \Sigma$, a neighbor set $Nbr_{CS}(u, v, l)$ is defined as $\cup_{u_n \in Nbr_S(u,l)} \{v_n | v_n \in C(u_n | u, v)$ and $v_n \in C(u_n)$ is $valid\}$.

Definition 4.4. Given a subgraph S of G and a marked CS on S and G, $v \in C(u)$ is *neighbor-safe* if $|Nbr_S(u, l)| \leq |Nbr_{CS}(u, v, l)|$ for every label $l \in \Sigma$.

These conditions are necessary conditions for the existence of an embedding that maps u to v. During the filtering process, a candidate $v \in C(u)$ violating any of two conditions is marked as invalid. Thus, after the filtering process, all embeddings are still preserved in the marked CS and consist of only valid candidates.

For each vertex u in $V(S)$ and each candidate v in $C(u)$, the "(u, v) pair" represents a node in the marked CS. For a (u, v) pair and a (u', v') pair, we say (u, v) is adjacent to (u', v') if and only if there is an edge between $v \in C(u)$ and $v' \in C(u')$ in the marked CS. Our algorithm runs over these (u, v) pairs.

In *Extension*, a new subgraph is created by adding an edge $e = (u, u')$. This new edge e may cause certain candidates v of vertex u (also certain candidates v' of vertex u') to violate connectivity-safe or neighbor-safe conditions. The (u, v) pairs for such v (also (u', v') pairs for such v') form the initial set I for the filtering process, and the filtering begins with the pairs in the set I.

Algorithm 3 shows the overview of this filtering process. It computes the initial set I for filtering (line 1). Then, *CSnodeFiltering* computes a compact valid candidate set, starting by marking candidates $v \in C(u)$ as invalid for each (u, v) pair in the initial set I. This returns whether subgraph S is a subgraph for which MNI computation is needed along with the refined CS (line 2).

Algorithm 4 shows *CSnodeFiltering*, which is the process of computing a compact valid candidate set by marking the candidates that violate any of the connectivity-safety or neighbor-safety conditions as invalid from the (u, v) pairs in the initial set I. During the process, the algorithm checks whether the size of $VL(u)$ for a vertex u is less than or equal to τ. If this is the case, indicating that the upper bound of MNI, minVL(S), cannot exceed τ, the algorithm terminates early and returns false, along with the refined marked CS. Conversely, if the filtering process completes without such early termination, indicating that the upper bound of MNI exceeds τ, it returns true, along with the refined marked CS.

The algorithm uses a queue Q to store (u, v) pairs, where the state of candidate v in $C(u)$ has recently transitioned from valid to invalid due to violating any condition.

Initially, for each (u, v) pair in the initial set I, the candidates $v \in C(u)$ are marked as invalid, and the algorithm checks the size of $VL(u)$. After checking the size of $VL(u)$, the (u, v) pair is added into the Q (lines 1-5). A (u, v) pair is popped from Q (line 7). The changed state of the candidate $v \in C(u)$ to invalid triggers update in the $NbrCnt$ and Nbr_{CS} for candidates $v_n \in C(u_n)$ adjacent to $v \in C(u)$. If these updates cause $v_n \in C(u_n)$ to violate any of the connectivity-safety or neighbor-safety condition, then $v_n \in C(u_n)$ is marked as invalid and the algorithm updates the size of $VL(u_n)$. After checking the size of $VL(u_n)$ against τ, this candidate $v_n \in C(u_n)$ is added into the queue Q (lines 8-23). The algorithm continues this process until the queue Q is empty.

Example 4.5. Figures 3(c)-(h) show filtering process when considering the new edge (u_4, u_5) for a graph G and a subgraph S in Figure 3. Among the candidates of u_4, u_5, which are the endpoints of the new edge (u_4, u_5), the candidate $v_6 \in C(u_4)$ violates the connectivity-safety. Thus, the initial set I is $\{(u_4, v_6)\}$. Then, $v_6 \in C(u_4)$ is marked as invalid and the pair (u_4, v_6) is inserted into the queue Q. Subsequently, the pair (u_4, v_6) is popped from Q and for candidates adjacent to $v_6 \in C(u_4)$, their $Nbrcnt$ and Nbr_{CS} are updated. The candidates $v_1 \in C(u_1)$ and $v_1 \in C(u_3)$ violate connectivity-safety and thus are marked as invalid and added into Q. Subsequently, $v_4 \in C(u_2)$ is also marked as invalid.

In dense graphs, marking (u, v) pairs as invalid tends to affect more pairs. In our implementation, we apply the filtering process when the average degree of the graph G is greater than or equal to 3 and we erase the invalid candidates in the marked CS.

Algorithm 5: ComputeMNI(basic)

input : A subgraph S, a marked CS of S
output: max(MNI(S), τ), refined marked CS

```
1  mni ← +∞
2  for u ∈ V(S) do
3      count ← 0
4      for v ∈ C(u) do
5          if v ∈ C(u) is already marked as confirmed then
6              count++
7              continue
8          Find an embedding that maps u to v
9          if there is such an embedding f then
10             mark v' ∈ C(u') as confirmed for all mappings
                 (u', v') in f
11             count++
12         else
13             mark v ∈ C(u) as invalid
14     if count ≤ τ then
15         return (τ, CS)
16     mni ← min(mni, count)
17 return (mni, CS)
```

5 IMPROVING MNI COMPUTATION

In the previous section, we showed that minVL serves as an upper bound of MNI. In this section, we will define a lower bound of MNI and propose an algorithm to compute the exact MNI using both the upper and lower bounds.

Algorithm 5 shows the basic process of computing MNI of the subgraph S. For each vertex $u \in V(S)$ and each candidate $v \in C(u)$, the algorithm checks if there exists an embedding in which u is mapped to v. If such an embedding exists, for every mapping (u', v') in that embedding, $v' \in C(u')$ is marked as confirmed (lines 9-11). These confirmed candidates $v' \in C(u')$ are then skipped in subsequent checks for the existence of embeddings (lines 5-7). Conversely, if there is no such embedding, then $v \in C(u)$ is marked as invalid (lines 12-13). After checking all candidates for u, the size of the image set $F(u)$ becomes equal to the number of candidates marked as confirmed, so if this size is less than or equal to τ, the algorithm terminates early, as the MNI of the subgraph cannot exceed τ (lines 14-15). After candidates for all vertices in $V(S)$ are marked, the algorithm returns the MNI of the subgraph (line 17).

In the basic algorithm, every candidate is marked as confirmed or invalid to compute the MNI. In contrast, our algorithm use lower bound and upper bound of the MNI, enabling the computation of the MNI without the need to mark all candidates as confirmed or invalid.

Definition 5.1. For a subgraph S of G and a marked CS on S and G, the *confirmed candidate set* $CF(u)$ is the set of confirmed candidates in $C(u)$ for each vertex $u \in V(S)$.

Definition 5.2. For a subgraph S of G, $minCF(S)$ is the minimum size of *confirmed candidate set* $CF(u)$ for each $u \in V(S)$, which is $min_{u \in V(S)} |CF(u)|$.

Algorithm 6: ComputeMNI

input : A subgraph S, a marked CS of S
output: max(MNI(S), τ), refined marked CS

```
1  foreach u ∈ V(S) do
2      candIdx[u] ← 0
3  while minCF < minVL do
4      choose vertex u in V(S) based on |VL(u)| and |CF(u)|
5      while candIdx[u] < |C(u)| do
6          if (u, v) is marked as confirmed or invalid then
7              candIdx[u]++
8              continue
9          v ← C(u)[candIdx[u]]
10         Find an embedding that maps u to v
11         candIdx[u]++
12         if there is such an embedding f then
13             mark v' ∈ C(u') as confirmed for all mappings
                 (u', v') in f
14         else
15             (success, CS) ← CSnodeFiltering(S, CS, (u, v))
16             if success is false then
17                 return (τ, CS)
18         break
19 return (minCF, CS)
```

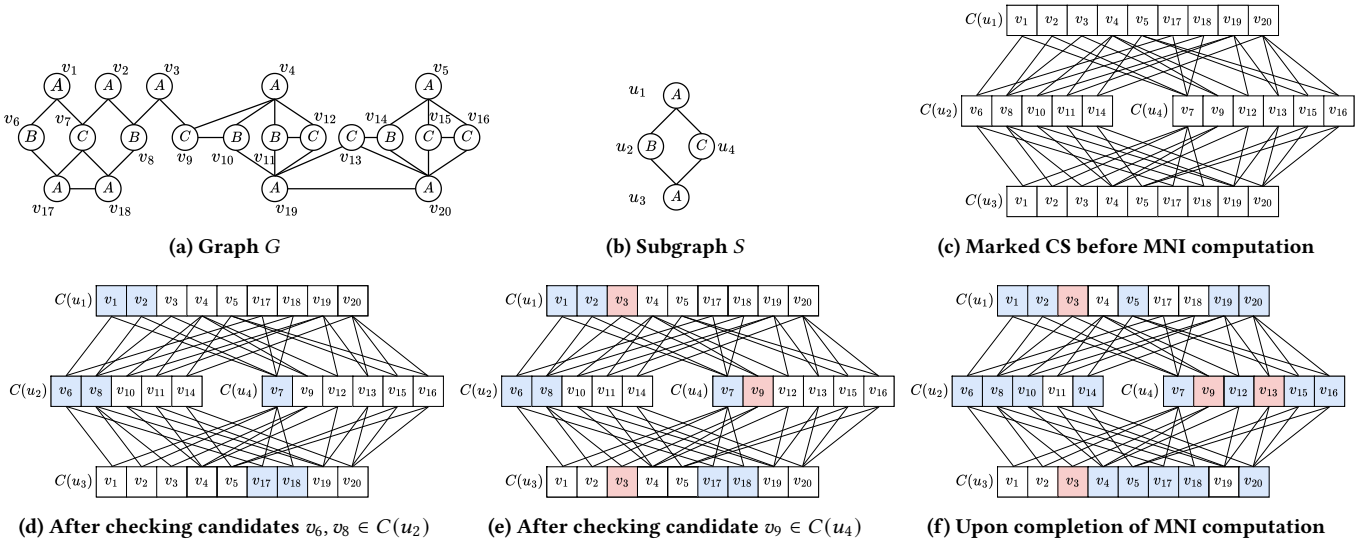
Theorem 5.3. *For a subgraph S of G, $minCF(S)$ is a lower bound of $MNI(S)$.*

Proof. For every $u \in V(S)$, the *confirmed candidate set* $CF(u)$ is the subset of the image set $F(u)$. So minCF(S) = $min_{u \in V(S)} |CF(u)|$ ≤ $min_{u \in V(S)} |F(u)|$ = MNI(S). Thus, minCF(S) is the lower bound of MNI(S). □

By Theorems 3.8 and 5.3, the MNI lies between minCF and minVL. Our algorithm employs a lower bound (minCF) and an upper bound (minVL) to optimize the computation of the MNI. For $v \in C(u)$, if an embedding in which u is mapped to v exists, then the size of $CF(u)$ increases. Consequently, minCF increases over time. Conversely, if no such embedding is found, then the size of $VL(u)$ decreases. Consequently, minVL decreases over time. As the algorithm progresses, the gap between the two bounds narrows. When these two bounds become equal, the converged value is the MNI of the subgraph, as the MNI of the subgraph lies between the lower and upper bounds. Therefore, our algorithm continues until the two bounds converge, and then it outputs the converged value as the MNI of the subgraph.

The basic algorithm sequentially selects a vertex u and check all candidates in $C(u)$ before proceeding to the next vertex. Our algorithm selects each (u, v) pair and checks the candidate $v \in C(u)$ at each iteration. Our algorithm selects a vertex u such that $|VL(u)| < |CF(u)|$, as the equal sizes of the two sets indicate that all candidates in $C(u)$ are checked.

The selection of vertex u depends on the ratio of the current number of candidates for which no embedding exists to the total number of candidates checked. If the ratio is high, implying that



(a) Graph G **(b) Subgraph S** **(c) Marked CS before MNI computation**

(d) After checking candidates $v_6, v_8 \in C(u_2)$ **(e) After checking candidate $v_9 \in C(u_4)$** **(f) Upon completion of MNI computation**

Figure 4: Process of computing MNI for subgraph S in graph G. After finding two embeddings $\{(u_1, v_1), (u_2, v_6), (u_3, v_{17}), (u_4, v_7)\}$ and $\{(u_1, v_2), (u_2, v_8), (u_3, v_{18}), (u_4, v_7)\}$, the marked CS is shown in Figure (d). Subsequently, upon finding that there is no embedding that maps u_4 to v_9, the filtering process marks $v_3 \in C(u_1)$ and $v_3 \in C(u_3)$ as invalid (Figure (e)). This process continues until the lower bound and upper bound converge, with the final value of MNI being 4 (Figure (f)).

the graph might have fewer embeddings, we select u with the smallest $|VL(u)|$ to quickly reduce the upper bound. If multiple vertices tie with the same minimum $|VL(u)|$, we further break the tie by choosing the vertex among these with the smallest $|CF(u)|$. Conversely, if the ratio is low, we select u by prioritizing the smallest $|CF(u)|$ to increase the lower bound, and use $|VL(u)|$ to break ties in a similar manner.

Algorithm 6 shows the process of computing MNI(S). Since a vertex u is selected in each iteration, we maintain $candIdx[u]$, which stores the index of the next candidate to be checked in the candidate set $C(u)$. Initially, for every vertex $u \in V(S)$, $candIdx[u]$ is set to 0 (lines 1-2). The vertex u is chosen based on the method previously described (line 4). We then check the first undetermined candidate $v \in C(u)$ (lines 5-10). If an embedding in which u is mapped to v exists, each candidate in the embedding is marked as confirmed (lines 12-13). Otherwise, *CSnodeFiltering* is invoked, with the initial set I consisting of the pair (u, v) (lines 14-17). The algorithm proceeds until the lower and upper bounds converge (lines 3-18). If the upper bound is less than or equal to τ, indicating the subgraph is infrequent, the algorithm terminates early (lines 16-17).

Example 5.4. Consider a graph and a subgraph S in Figure 4. First, vertex u_2 is selected because all vertices $u \in V(S)$ have the same $|CF(u)|$, and u_2 has the smallest $|VL(u)|$. We find an embedding $\{(u_1, v_1), (u_2, v_6), (u_3, v_{17}), (u_4, v_7)\}$, and then select u_2 again and find an embedding $\{(u_1, v_2), (u_2, v_8), (u_3, v_{18}), (u_4, v_7)\}$. Next, u_4 is selected because u_4 has the smallest $|CF(u)|$. We find that no embedding exists for the candidate $v_9 \in C(u_4)$. Consequently, CSnodeFiltering is invoked, and candidates $v_3 \in C(u_1)$ and $v_3 \in C(u_3)$ are also marked as invalid. This process continues until the lower bound and upper bound converge, with the final value of MNI(S) being 4.

We employ the searching process from VEQ [29] to find an embedding. During this searching process, each vertex $u \in V(S)$ is first mapped to an undetermined candidate in $C(u)$, prioritizing undetermined ones over confirmed ones to mark undetermined candidates as quickly as possible. For MNI computation, graph automorphism is used. We compute the automorphism of graph S using nauty & Traces [34]. As the implementation disregards labeled edges, automorphism is only applied to graph without labeled edge.

DFScode. We use gSpan's *canonical DFScode* [59] to avoid generating duplicate subgraphs. A DFScode is constructed from a depth-first search (DFS) traversal [15] of a graph. Among DFScodes, the canonical DFScode is the smallest one in lexicographical order.

After extending a subgraph S to create a new subgraph S', we construct the DFScode of S' by adding the new edge to the end of the canonical DFScode of S. If this DFS code is not canonical, S' is disregarded, and we continue to generate other subgraphs.

If a vertex of a subgraph is adjacent to only leaf vertices with an identical label during a DFS traversal, these vertices will generate the same DFScodes, even though there are an exponential number of DFS traversal orders. In such cases, therefore, we produce just one DFScode instead of generating the DFScode from every possible traversal order.

6 THEORETICAL ANALYSIS

Table 2 shows the time and space complexities of CSnodeFiltering (Algorithm 4) and MNI computation (Algorithm 6) for each subgraph S. Here, the time complexity of CSnodeFiltering is the sum of the complexity of filtering (line 2 in Algorithm 3) to obtain the tight upper bound minVL(S) (line 7 of Algorithm 2) just after creating the subgraph S and the complexity of filtering during MNI computation (line 15 in Algorithm 6).

By Theorem 6.1, the time complexity of CSnodeFiltering is $O(|E(S)|$ $|E(G)|)$. MNI computation is an NP-hard problem, thus requiring exponential time to compute [17, 20, 29]. Since both CSnodeFiltering and MNI computation use the marked CS, the space complexity

Table 2: Time and space complexities of CSnodeFiltering and MNI computation of `Minting`

	Time	Space
CSnodeFiltering	$O(\|E(S)\| \cdot \|E(G)\|)$	$O(\|E(S)\| \cdot \|E(G)\|)$
MNI computation	$O(\|V(S)\| \cdot \|V(G)\|^{\|V(S)\|})$	$O(\|E(S)\| \cdot \|E(G)\|)$

is proportional to the size of the marked CS, which is bounded by $O(|E(S)| \cdot |E(G)|)$.

Theorem 6.1. *For a subgraph S, the total time complexity of Algorithm 4 for S is $O(|E(S)| \cdot |E(G)|)$.*

Proof. A pair (u, v), where $u \in V(S)$ and $v \in C(u)$, is added to Q when the state of $v \in C(u)$ transitions from undetermined to invalid. Since this transition happens at most once for each candidates $v \in C(u)$, each pair (u, v) is added into Q at most once. Consequently, each pair (u, v) is also popped from Q at most once. Within the for loop (lines 8-23), each operation takes $O(1)$ time. Thus, each iteration for a candidate $v \in C(u)$ takes time proportional to the number of candidates adjacent to $v \in C(u)$. The overall runtime is then bounded by the sum of the number of candidates adjacent to $v \in C(u)$ for all vertices $u \in V(S)$ and their candidates $v \in C(u)$. This sum is twice the number of edges in CS. Since the number of edges in CS is bounded by $O(|E(S)| \cdot |E(G)|)$, the total time complexity of algorithm 4 is $O(|E(S)| \cdot |E(G)|)$. □

7 EXPERIMENTAL EVALUATION

In this section, we conduct experiments to evaluate the effectiveness of the proposed algorithm `Minting` for top-k frequent subgraph mining. We compare our algorithm with three state-of-the-art algorithms, GRAMI [17], Peregrine [26], and FastPat[64, 65]. In subsection 7.1, we describe our experimental setting. Then, we compare our algorithm with GRAMI and Peregrine in subsection 7.2 and with FastPat in subsection 7.3. We present the size distribution of top-k subgraphs in subsection 7.4. Finally, we evaluate the effectiveness of our techniques in subsection 7.5.

7.1 Experimental Setting

Experiments are conducted on six real-world datasets, which are MiCo, Patents, Human, WCGoals, Yeast and Phy-Cit used in previous works [17, 29, 43, 64, 65]. MiCo [17] is a graph that models the co-authorship information in the Microsoft academic. Patents [32] is a citation network of U.S. patents. Human and Yeast [23] is a protein-protein interaction network. WCGoals [66] is a graph about the FIFA World Cup events. Phy-Cit [32] is a citation network covering e-print arXiv HEP-PH papers. The characteristics of the datasets are summarized in Table 3. Since most data graphs have edge labels, we extended our algorithm to handle graphs with edge labels.

Table 3: Datasets and their characteristics

Dataset	$\|V\|$	$\|E\|$	# vertex labels	# edge labels	Avg degree
MiCo	100K	1.08M	29	10	21.606
Patents	2.93M	13.96M	419	5	9.504
Human	4K	86K	44	-	36.920
WCGoals	49K	158K	11	13	6.443
Yeast	3K	13K	71	-	8.041
Phy-Cit	31K	347K	6	-	22.736

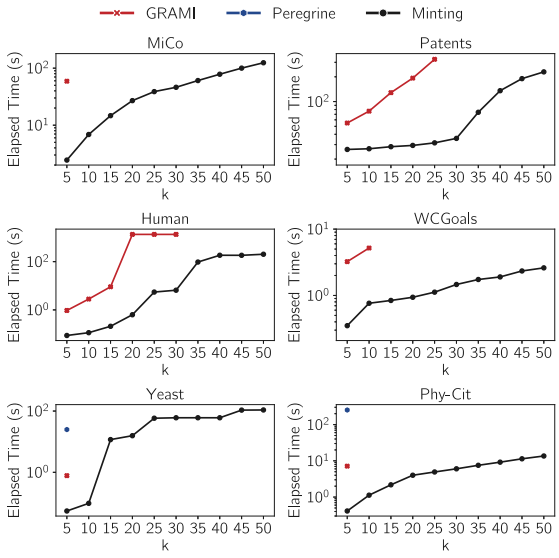


Figure 5: Elapsed time of GRAMI, Peregrine, and `Minting`. Points not shown indicate cases where an algorithm did not finish within the time limit.

We obtained the source codes of GRAMI, Peregrine, and FastPat online [1–3]. GRAMI and FastPat are implemented in Java, while Peregrine is implemented in C++. Our algorithm, employing the search process of VEQ, is implemented in C++. The experiments are conducted on a CentOS machine equipped with dual Intel Xeon E5-2680 v3 2.5GHz CPUs and 256GB of memory. The source code of `Minting` is available at https://github.com/SNUCSE-CTA/Minting.

We vary the parameter k from 5 to 50 in increments of 5 and measure the elapsed time for each algorithm. We set a time limit of 15 minutes for `Minting` and Peregrine, and set it to 30 minutes for GRAMI and FastPat, considering their implementations in Java.

7.2 Comparison with GRAMI and Peregrine

In this subsection, we conduct a comparison of `Minting` with GRAMI and Peregrine. While our algorithm takes k as input and finds the top-k frequent subgraphs, GRAMI takes a threshold value τ as input and finds subgraphs with an MNI greater than or equal to τ. Peregrine takes a threshold τ and an integer m as input, and finds subgraphs with m edges that have an MNI greater than or equal to τ. To conduct a comparison, we first run `Minting` to get the minimum MNI among the top-k results, and then run GRAMI and Peregrine using this minimum MNI as the threshold τ. The number m of edges (input for Peregrine) was set to the maximum number of edges among the top-k results from `Minting`. Since Peregrine does not handle edge labels, experiments for Peregrine were conducted only on datasets without edge labels (Human, Yeast, and Phy-Cit).

Figure 5 shows the elapsed time of the algorithms. `Minting` shows better performances than the other algorithms on all datasets. `Minting` always finishes within the time limit, while GRAMI fails to do so for large k in all datasets. In cases where both algorithms finish within the time limit, `Minting` outperforms GRAMI by up to three orders of magnitude (in Human when k is 20). Peregrine finished within the time limit only on the Yeast and Phy-Cit datasets when $k = 5$, and failed to complete within the time limit in other cases.

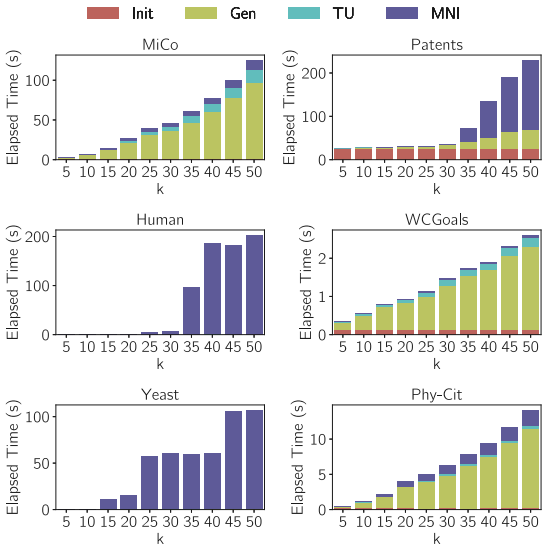


Figure 6: Breakdown of the overall time taken by `Minting`. 'Init' for graph reading and initial preprocessing, 'Gen' for generating subgraphs and constructing data structures, 'TU' for computing tight MNI upper bound, and 'MNI' for computing the MNI.

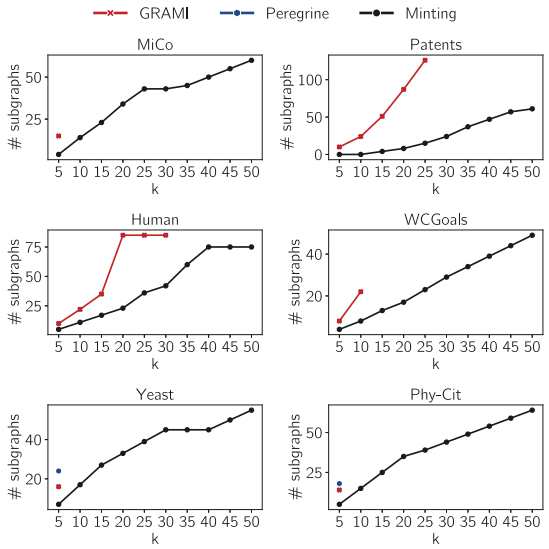


Figure 7: Number of subgraphs for which MNI computation is required for GRAMI, Peregrine and `Minting`

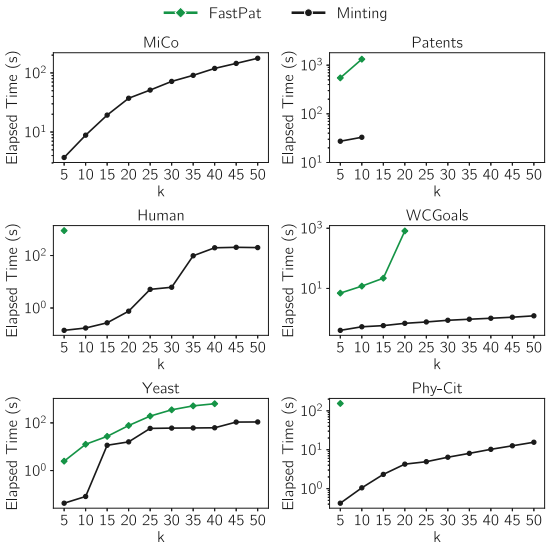


Figure 8: Elapsed time of FastPat and `Minting`. Points not shown indicate cases where an algorithm did not finish within the time limit.

When Peregrine completed within the time limit, `Minting` was hundreds of times faster. Considering that Peregrine's elapsed time significantly exceeded the 15-minute time limit when it failed to complete, `Minting` demonstrated a performance difference of over a thousand times.

Both GRAMI and Peregrine take τ as an input, which is the minimum MNI from the top-k results of `Minting`. This means τ contains more information than k, the input of `Minting`. Despite this, our algorithm still achieves significant performance improvements compared to GRAMI and Peregrine. This is because we reduce the number of subgraphs for which MNI computation is required and improve MNI computation itself.

GRAMI takes a substantial amount of time for MNI computation. The elapsed time for GRAMI consists of three components: initialization, subgraph generation, and MNI computation. In most cases with large k, MNI computation accounts for more than 70% of GRAMI's processing time, which leads to longer time to solve the problem. In contrast, `Minting` significantly reduces the time for MNI computation. Figure 6 shows the breakdown of the overall time taken by `Minting` into four components: 'Init' for graph reading and initial preprocessing, 'Gen' for generating subgraphs and constructing data structures such as marked CS, 'TU' for computing tight MNI upper bound, and 'MNI' for computing the MNI. In the Patents, Human, and Yeast datasets, MNI computation is a major part of the processing time. For other datasets, subgraph generation is the dominant part because `Minting` reduces the MNI computation time. Overall, `Minting` reduces the MNI computation time, so much as to make it faster than GRAMI.

Figure 7 compares the number of subgraphs for which MNI computation is required for the algorithms. For `Minting`, this is the number of calls to 'computeMNI' in Algorithm 1. `Minting` computes the MNI for fewer subgraphs than GRAMI and Peregrine in all datasets. This is due to our tight MNI upper bound, thus leading to a decrease in the overall time to solve the problem.

7.3 Comparison with FastPat

In this subsection, we conduct a comparison of `Minting` with FastPat. FastPat takes a directed graph, a core graph, and k as inputs, and it finds the top-k frequent subgraphs extended from the core graph. Since FastPat requires a directed graph as its input, we convert undirected graphs (MiCo, Human, Yeast) into bidirectional graphs for FastPat's input. To compare our algorithm with FastPat, we modify our algorithm to take a core graph and to find the top-k frequent subgraphs that are extended from this core graph. In our experiments, the core graph is the edge with the largest MNI in the data graph.

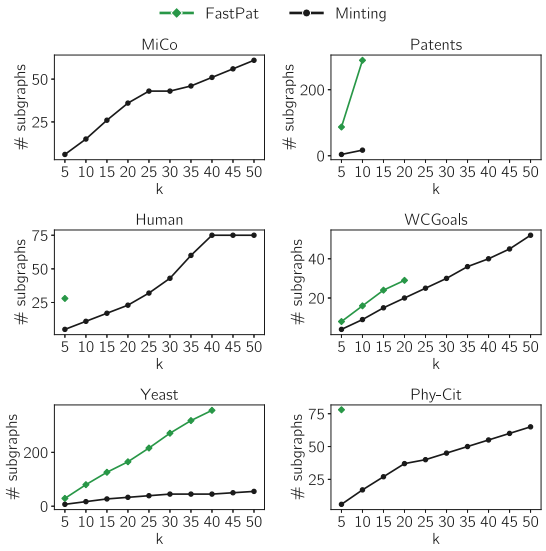


Figure 9: Number of subgraphs for which MNI computation is required for FastPat and `Minting`

Figure 8 shows the elapsed time of the algorithms. `Minting` finishes within the time limit except for the Patents dataset, whereas FastPat cannot finish in most cases of MiCo, Patents, Human, WCGoals, and Phy-Cit. In cases where both algorithms finish within the time limit, `Minting` outperforms FastPat by up to three orders of magnitude (in Human when k is 5, and in WCGoals when k is 20).

FastPat also takes a substantial amount of time for MNI computation. The elapsed time for FastPat consists of three components: initialization, subgraph generation, and MNI computation. In most cases with large k, MNI computation accounts for more than 90% of FastPat's processing time. In contrast, `Minting` effectively reduces the time for MNI computation. Figure 9 compares the number of subgraphs for which MNI computation is required for `Minting` and FastPat. `Minting` consistently computes the MNI for fewer subgraphs than FastPat due to our tighter MNI upper bound.

For the Patents dataset, our algorithm finishes in Figure 5 where we find top-k frequent subgraphs, but it cannot finish in many cases in Figure 8 where we find top-k frequent subgraphs extended from the core graph. In Figure 8, our algorithm starting from the core graph needs a deeper exploration into the subgraph lattice, which results in encountering subgraphs (cycles with 13 or 14 edges) that demand a large amount of time for MNI computation. In contrast, in Figure 5 where our algorithm starts from many frequent edges, these time-consuming subgraphs are not encountered, allowing the algorithm to complete the task within the time limit.

7.4 Size Distribution of Top-k Subgraphs

Figure 10 shows the size (number of edges) distribution of top-50 frequent subgraphs for MiCo and Patents. In general, the sizes of top-k subgraphs are small when k is small, and they get larger as k increases. Particularly, in the Patents dataset until k reaches 10, every top-k frequent subgraph consists of a single edge only. To find more interesting subgraphs, it is necessary to increase the value of k, e.g., when $k = 50$, we find larger frequent subgraphs. In graph mining, therefore, mining top-k frequent subgraphs for

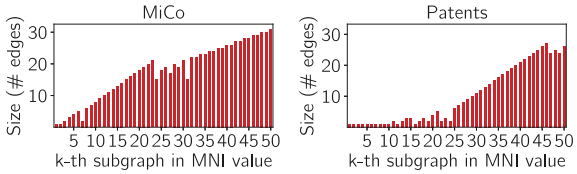


Figure 10: Size (number of edges) distribution of top-50 frequent subgraphs for MiCo and Patents, where top-50 subgraphs are listed in descending order of MNI values in x-axis.

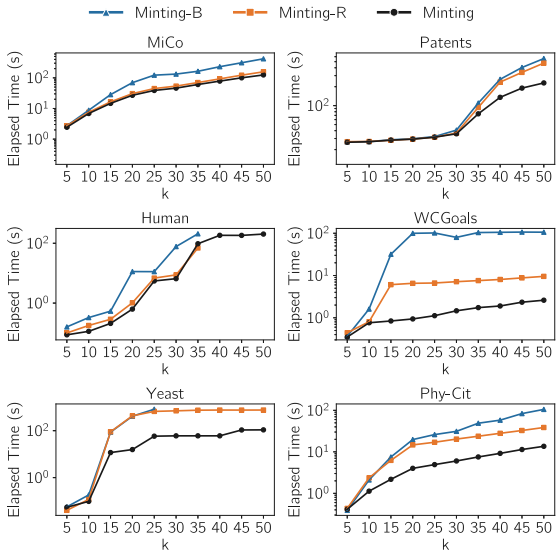


Figure 11: Elapsed time of our algorithm and its variants

large k values is important. The varying sizes of the subgraphs can benefit visual query interfaces by providing a diverse set of subgraph patterns for formulating a graph query [63].

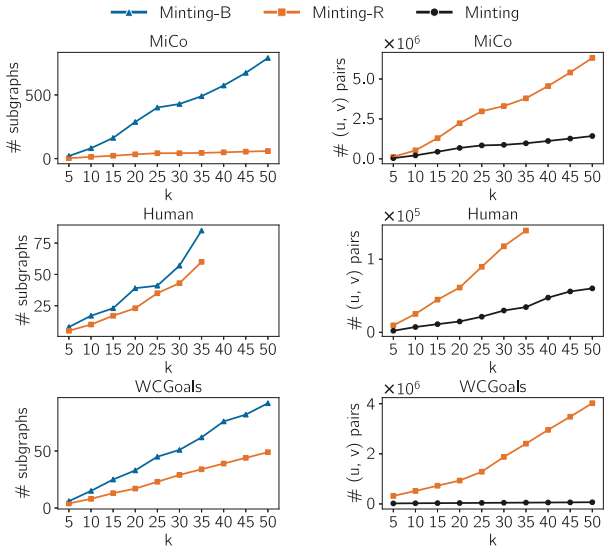
7.5 Evaluation of Techniques

In this subsection, we evaluate the effectiveness of our individual techniques in reducing the elapsed time to solve top-k frequent subgraph mining. To measure the performance gains achieved by each technique, we run our algorithm and its variants as follows:

- `Minting-B`: a baseline version that excludes the method for reducing the number of subgraphs and uses the basic method for computing MNI;
- `Minting-R`: a version that reduces the number of subgraphs by filtering and uses the basic method for computing MNI;
- `Minting`: our algorithm that reduces the number of subgraphs by filtering, computes MNI using lower and upper bounds of MNI, and optimizes the checking of the DFScode canonical form.

Figure 11 shows the elapsed time of these algorithms for top-k frequent subgraph mining. In all datasets, `Minting-R` consistently outperforms `Minting-B`, and `Minting` consistently outperforms `Minting-R`.

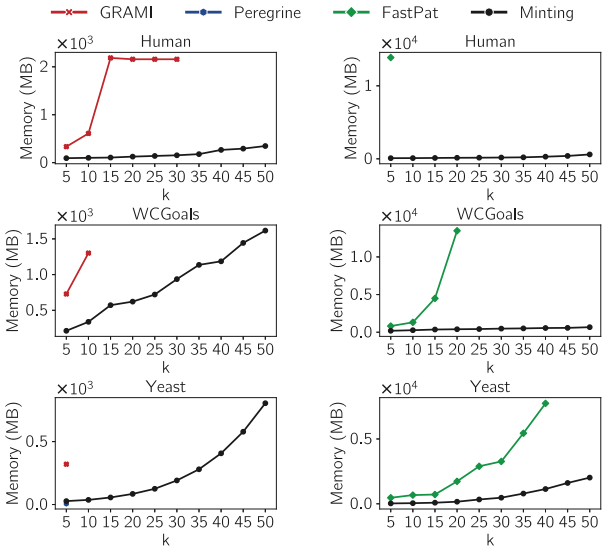
Our contributions on performance are 1) reducing the number of subgraphs (performance difference between `Minting-B` and `Minting-R`) and 2) improving MNI computation (performance difference between `Minting-R` and `Minting`), and when these two



(a) Number of subgraphs for which MNI computation is required

(b) Number of (u, v) pairs to find an embedding that maps u to v.

Figure 12: Number of subgraphs and (u, v) pairs



(a) Memory usage of GRAMI, Peregrine, and `Minting`

(b) Memory usage of FastPat and `Minting`

Figure 13: Memory usage of algorithms

techniques are combined together, they produced the significant performance improvements shown in our experiments. In Figure 11, the performance difference between `Minting-B` and `Minting-R` is big in MiCo, Human, WCGoals, and the performance difference between `Minting-R` and `Minting` is big in Patents, WCGoals, Yeast, Phy-Cit.

Figure 12a shows the number of subgraphs for which MNI computation is required, which is the number of calls to 'computeMNI' in Algorithm 1, for `Minting-B` and `Minting-R`. Theoretically, the number of subgraphs that need to be checked is $O(k \cdot k! \cdot {d_{max}}^{(k-1)})$, where d_{max} is the maximum degree of graph G. The initial k subgraphs are the most frequent single edges. The number of new subgraphs that can be generated from each subgraph S is at most $|V(S)| * d_{max}$. Due to the anti-monotone property of MNI, the maximum number of vertices in a subgraph is k+1. Since the number of subgraphs generated from a single edge is $O(2 \cdot d_{max} \cdot 3 \cdot d_{max} \cdot \ldots \cdot k \cdot d_{max}) = O(k! \cdot {d_{max}}^{(k-1)})$, we get the bound above.

However, the actual number of subgraphs for which MNI needs to be computed is smaller than this complexity. For the baseline `Minting-B` on the MiCo dataset with $k = 50$, the number of subgraphs to compute MNI was 790. With filtering applied, `Minting-R` reduced the number of subgraphs for MNI computation to 60 (Figure 12a). Thus, our filtering using the upper bound minVL(S) reduced the number of subgraphs to approximately 1/13th.

The most time-consuming part in Algorithm 6 (that computes MNI of a subgraph S in G) is to find an embedding that maps u to v (line 10), so it is important to reduce the number of times this is done. Figure 12b shows the number of (u, v) pairs to find an embedding that maps u to v. This count is the number of calls to line 8 in Algorithm 5 for `Minting-R` and line 10 in Algorithm 6 for `Minting`. Theoretically, the number of (u, v) pairs that need to be checked for embeddings is $O(|V(S)| \cdot |V(G)|)$. However, the actual number of (u, v) pairs checked for embeddings is smaller than this complexity. In the WCGoals dataset when k is 50, the basic MNI computation method (Algorithm 5, i.e., `Minting-R`) checks 4,055,287 (u, v) pairs for embeddings. Our algorithm utilizing the upper and lower bounds of MNI (Algorithm 6, i.e., `Minting`) reduces the number of (u, v) pairs checked for embeddings to 68,217 (Figure 12b). That is, our MNI computation using upper and lower bounds reduces the number of (u, v) pairs checked for embeddings to approximately 1/60th.

Figure 13a shows the memory usage of GRAMI, Peregrine, and `Minting`, and Figure 13b shows the memory usage of FastPat and `Minting`. `Minting` consistently uses less memory than both GRAMI and FastPat across all datasets. Peregrine uses less memory than `Minting` on Yeast when $k = 5$. In other cases, however, Peregrine failed to complete within the time limit. In general, the memory usage of `Minting` remains competitive.

Overall, our algorithm `Minting` is a feasible solution for top-k frequent subgraph mining, even for large k.

8 CONCLUSION

In this paper, we have introduced a new data structure called *marked CS*, and proposed key concepts minVL(S) and minCF(S) based on the data structure, which work as an upper bound and an lower bound of the MNI value of a subgraph S, respectively. Using these concepts, we designed an algorithm `Minting` for top-k frequent subgraph mining, which outperforms state-of-the-art algorithms in both time and space. It will be an interesting future work to find more applications of these concepts. Developing an efficient parallel algorithm for top-k frequent subgraph mining is also an interesting future work.

ACKNOWLEDGEMENTS

S. Lee, Y. Lee, and K. Park were supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems).

REFERENCES

[1] 2024. *FastPat-KG.* Retrieved 2024-12-17 from https://github.com/DBGroup-SUSTech/FastPat-KG

[2] 2024. *GraMi.* Retrieved 2024-12-17 from https://github.com/ehab-abdelhamid/GraMi

[3] 2024. *Peregrine.* Retrieved 2024-12-17 from https://github.com/pdclab/peregrine

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

[5] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. 2017. Incremental frequent subgraph mining on large evolving graphs. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2710–2723.

[6] Isam A Alobaidi, Jennifer L Leopold, and Ali A Allami. 2019. The Use of Frequent Subgraph Mining to Develop a Recommender System for Playing Real-Time Strategy Games. In *Industrial Conference on Data Mining*. 146–160.

[7] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1–26.

[8] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1199–1214.

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

[10] 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.

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

[12] Yifan Chen, Xiang Zhao, Xuemin Lin, Yang Wang, and Deke Guo. 2018. Efficient mining of frequent patterns on uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 2 (2018), 287–300.

[13] Young-Rae Cho and Aidong Zhang. 2009. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on Information Technology in Biomedicine* 14, 1 (2009), 30–36.

[14] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2186–2198.

[15] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[16] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1036–1050.

[17] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* 7, 7 (2014), 517–528.

[18] Mathias Fiedler and Christian Borgelt. 2007. Subgraph support in a single large graph. In *Seventh IEEE International Conference on Data Mining Workshops*. IEEE, 399–404.

[19] Philippe Fournier-Viger, Chao Cheng, Jerry Chun-Wei Lin, Unil Yun, and R Uday Kiran. 2019. Tkg: Efficient mining of top-k frequent subgraphs. In *Proceedings of Big Data Analytics*. 209–226.

[20] Michael R Garey and David S Johnson. 1979. *Computers and intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[21] Valerie Guralnik and George Karypis. 2001. A scalable algorithm for clustering sequential data. In *Proceedings of IEEE International Conference on Data Mining*. 179–186.

[22] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1429–1446.

[23] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 337–348.

[24] Jun Huan, Wei Wang, and Jan Prins. 2003. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of IEEE International Conference on Data Mining*. 549–552.

[25] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, approximate graph pattern mining at scale. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation*. 745–761.

[26] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[27] Yi Jia, Jintao Zhang, and Jun Huan. 2011. An efficient graph-mining method for complicated and noisy data with real-world applications. *Knowledge and Information Systems* 28 (2011), 423–447.

[28] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105.

[29] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 925–937.

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

[31] Ngoc-Thao Le, Bay Vo, Lam BQ Nguyen, Hamido Fujita, and Bac Le. 2020. Mining weighted subgraphs in a single large graph. *Information Sciences* 514 (2020), 149–165.

[32] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 177–187.

[33] Ruirui Li and Wei Wang. 2015. REAFUM: Representative approximate frequent subgraph mining. In *Proceedings of SIAM International Conference on Data Mining*. 757–765.

[34] Brendan D McKay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *Journal of Symbolic Computation* 60 (2014), 94–112.

[35] Jinghan Meng and Yi-cheng Tu. 2017. Flexible and feasible support measures for mining frequent patterns in large labeled graphs. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 391–402.

[36] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[37] Aida Mrzic, Pieter Meysman, Wout Bittremieux, Pieter Moris, Boris Cule, Bart Goethals, and Kris Laukens. 2018. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining* 11 (2018), 1–24.

[38] Muhammad Anis Uddin Nasir, Cigdem Aslay, Gianmarco De Francisci Morales, and Matteo Riondato. 2021. Tiptap: approximate mining of frequent k-subgraph patterns in evolving graphs. *ACM Transactions on Knowledge Discovery from Data* 15, 3 (2021), 1–35.

[39] Dheepikaa Natarajan and Sayan Ranu. 2016. A scalable and generic framework to mine top-k representative subgraph patterns. In *Proceedings of IEEE International Conference on Data Mining*. 370–379.

[40] Lam BQ Nguyen, Bay Vo, Ngoc-Thao Le, Vaclav Snasel, and Ivan Zelinka. 2020. Fast and scalable algorithms for mining subgraphs in a single large graph. *Engineering Applications of Artificial Intelligence* 90 (2020), 103539.

[41] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1099–1114.

[42] Arneish Prateek, Arijit Khan, Akshit Goyal, and Sayan Ranu. 2020. Mining top-k pairs of correlated subgraphs in a large network. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1511–1524.

[43] Giulia Preti, Gianmarco De Francisci Morales, and Matteo Riondato. 2023. Maniacs: Approximate mining of frequent subgraph patterns through sampling. *ACM Transactions on Intelligent Systems and Technology* 14, 3 (2023), 1–29.

[44] Sayan Ranu and Ambuj K Singh. 2009. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *Proceedings of IEEE International Conference on Data Engineering*. 844–855.

[45] Tanay Kumar Saha, Ataur Katebi, Wajdi Dhifli, and Mohammad Al Hasan. 2017. Discovery of functional motifs from the interface region of oligomeric proteins using frequent subgraph mining. *IEEE/ACM transactions on Computational Biology and Bioinformatics* 16, 5 (2017), 1537–1549.

[46] Yinglong Song, Huey Eng Chua, Sourav S Bhowmick, Byron Choi, and Shuigeng Zhou. 2018. BOOMER: Blending visual formulation and processing of p-homomorphic queries on large networks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 927–942.

[47] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[48] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[49] 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.

[50] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos 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.

[51] Lini T Thomas, Satyanarayana R Valluri, and Kamalakar Karlapalem. 2010. Margin: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data* 4, 3 (2010), 1–42.

[52] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[53] Saif Ur Rehman, Kexing Liu, Tariq Ali, Asif Nawaz, and Simon James Fong. 2021. A graph mining approach for ranking and discovering the interesting frequent subgraph patterns. *International Journal of Computational Intelligence Systems* 14 (2021), 1–17.

[54] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 763–782.

[55] Tongtong Wang, Hao Huang, Wei Lu, Zhe Peng, and Xiaoyong Du. 2018. Efficient and scalable mining of frequent subgraphs using distributed graph processing systems. In *Proceedings of International Conference on Database Systems for Advanced Applications*. 891–907.

[56] Xin Wang, Zhuo Lan, Yu-Ang He, Yang Wang, Zhi-Gui Liu, and Wen-Bo Xie. 2022. A cost-effective approach for mining near-optimal top-k patterns. *Expert Systems with Applications* 202 (2022), 117262.

[57] Da Yan, Wenwen Qu, Guimu Guo, and Xiaoling Wang. 2020. Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *Proceedings of IEEE International Conference on Data Engineering*. 1938–1941.

[58] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. 2008. Mining significant graph patterns by leap search. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 433–444.

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

[60] Xifeng Yan and Jiawei Han. 2003. Closegraph: mining closed frequent graph patterns. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 286–295.

[61] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 335–346.

[62] Zhengwei Yang, Ada Wai-Chee Fu, and Ruifeng Liu. 2016. Diversified top-k subgraph querying in a large graph. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1167–1182.

[63] Zifeng Yuan, Huey Eng Chua, Sourav S Bhowmick, Zekun Ye, Wook-Shin Han, and Byron Choi. 2021. Towards plug-and-play visual graph query interfaces: data-driven selection of canned patterns for large networks. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1979–1991.

[64] Jian Zeng, Xiao Yan, Mingji Han, Bo Tang, et al. 2021. Fast core-based top-k frequent pattern discovery in knowledge graphs. In *Proceedings of IEEE International Conference on Data Engineering*. 936–947.

[65] Jian Zeng, Xiao Yan, Yan Li, Mingji Han, Bo Tang, et al. 2024. Extracting Top-k Frequent and Diversified Patterns in Knowledge Graphs. *IEEE Transactions on Knowledge and Data Engineering* 36, 2 (2024), 608–626.

[66] Gensheng Zhang, Damian Jimenez, and Chengkai Li. 2018. Maverick: Discovering exceptional facts from knowledge graphs. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 1317–1332.

[67] Changben Zhou, Jian Xu, Ming Jiang, Donghang Tang, and Sheng Wang. 2023. Mining Top-k Frequent Patterns in Large Geosocial Networks: A Mnie-Based Extension Approach. *IEEE Access* 11 (2023), 27662–27675.