Efficiently Mining δ-Tolerance Closed Frequent Subgraphs

Ichigaku Takigawa and Hiroshi Mamitsuka

Bioinformatics Center, Institute for Chemical Research, Kyoto University
Gokasho, Uji 611-0011, Japan

Institute for Bioinformatics Research and Development (BIRD)
Japan Science and Technology Agency (JST), Japan

Abstract

The output of frequent pattern mining is a huge number of frequent patterns, which are very redundant, causing a serious problem in understandability. We focus on mining frequent subgraphs for which well-considered approaches to reduce the redundancy are limited because of the complex nature of graphs. Two known, standard solutions are closed and maximal frequent subgraphs, but closed frequent subgraphs are still redundant and maximal frequent subgraphs are too specific. A more promising solution is δ-tolerance closed frequent subgraphs, which decrease monotonically in δ, being equal to maximal frequent subgraphs and closed frequent subgraphs for δ=0 and 1, respectively. However, the current algorithm for mining δ-tolerance closed frequent subgraphs is a naive, two-step approach in which frequent subgraphs are all enumerated and then sifted according to δ-tolerance closedness. We propose an efficient algorithm based on the idea of "reverse-search" by which the completeness of enumeration is guaranteed and for which new pruning conditions are incorporated. We empirically demonstrate that our approach significantly reduced the amount of real computation time of two compared algorithms for mining δ-tolerance closed frequent subgraphs, being pronounced more for practical settings.

1 Introduction

Mining frequent patterns is a major research subject in data mining and knowledge discovery over a wide variety of modern data, including itemsets, strings, trees and graphs. In particular, graphs are the most challenging data in knowledge discovery, with a lot of scientific applications such as chemoinformatics, bioinformatics and network analysis [17, 6]. Furthermore mining frequent subgraphs can be a basis for searching, indexing [25] and classifying [11] graphs in scientific databases, especially chemical compounds which are often found in chemistry, biology and pharmaceutical and medical sciences. The most important notion in mining frequent subgraphs is the *support* of a subgraph G, denoted by support(G), which is the number of records containing G in a given dataset of graphs. A subgraph is frequent if its support is not less than a given threshold, called *minimum support* or *minsup*. The purpose of frequent subgraph mining is to enumerate all frequent subgraphs in a given graph dataset, and efficient algorithms for this problem have been developed [23, 3, 15].

The number of outputs of frequent subgraph mining is usually huge and redundant, because all subgraphs of a frequent graph are frequent, due to the definition of frequent subgraphs. This huge number makes it difficult to understand and further analyze generated frequent graphs in terms of knowledge discovery. Thus reducing redundant outputs is an important issue, but because of the complex data structure of graphs, approaches for this problem are limited. The two most typical ideas are *closed* [24, 4] and *maximal* [14, 20] frequent subgraphs, which are natural extensions from closed and maximal frequent itemsets [22], respectively. A frequent subgraph G is closed unless the support of one of its supergraphs is the same as that of G. That is, frequent graph G can be removed from outputs if its support is the same as that of its frequent supergraph. This idea is powerful against transaction data of itemsets with a lot of duplications, but this cannot necessarily work well in scientific graph data such as a chemical library with no duplications, where the support of a supergraph of G is likely to be slightly smaller than the support of G. Thus, the number of closed frequent subgraphs is still too large in scientific applications. On the other hand, a frequent subgraph is maximal unless one of its supergraphs is frequent, meaning that all subgraphs of a frequent graph can be discarded. This drastically reduces the outputs, but maximal frequent subgraphs become very small in number and too specific.

Thus we need to further explore the problem of reducing the number of frequent subgraphs to an appropriate size. In fact, for frequent itemsets, a lot of ideas have been already considered for this issue, including non-derivable frequent sets, the top-k most frequent closed patterns, condensed patterns and k-summarized patterns etc [12]. However, for frequent subgraphs, to the best of our knowledge, only a small number of approaches have been presented. Their strategy is rather straightforward: they first generate closed frequent subgraphs and then choose representative patterns by clustering them [16, 7]. On the other hand, a more sophisticated idea is δ-tolerance closed frequent subgraphs [9], which were originally used in a method, called FG-Index, for indexing graphs. The δ-tolerance closed frequent subgraphs allow to smoothly link maximal and closed frequent subgraphs by using parameter δ, which takes a real value between zero and one. A frequent subgraph G is δ-tolerance closed unless the support of any supergraph of G is larger than or equal to $\max((1-\delta)\times\text{support}(G),\text{minsup})$. Thus δ-tolerance closed frequent subgraphs are a natural extension from closed frequent subgraphs by relaxing the strict definition on closedness to reduce redundant frequent subgraphs more. Furthermore, δ-tolerance closed frequent subgraphs have nice properties: 1) The number of δ-tolerance closed frequent subgraphs is monotonically decreasing in δ, which is a key to develop pruning rules in our enumeration algorithm. 2) δ-tolerance closed frequent subgraphs are equivalent to maximal frequent subgraphs and closed frequent subgraphs when δ=1 and 0, respectively.

Although δ-tolerance closed frequent subgraphs are very promising, the current algorithm for mining these graphs, i.e. the index extraction part of FG-Index [9], is straightforward (or naive). That is, the algorithm has simple two steps: All frequent subgraphs are first enumerated by using the gSpan algorithm, i.e. an existing frequent subgraph mining method, and then they are checked by the criterion of δ-tolerance closed frequent subgraphs. Of course the purpose of [9] was to build graph indices, and such a simple approach was enough for that purpose, because the upper limit was placed on the index size in [9]. However, in terms of frequent pattern mining, we need to enumerate all δ-tolerance closed frequent subgraphs, urging to develop a more efficient approach by considering pruning the search space more. In particular pruning must be more useful to improve the computational efficiency for the case with the minsup of a lower value, since

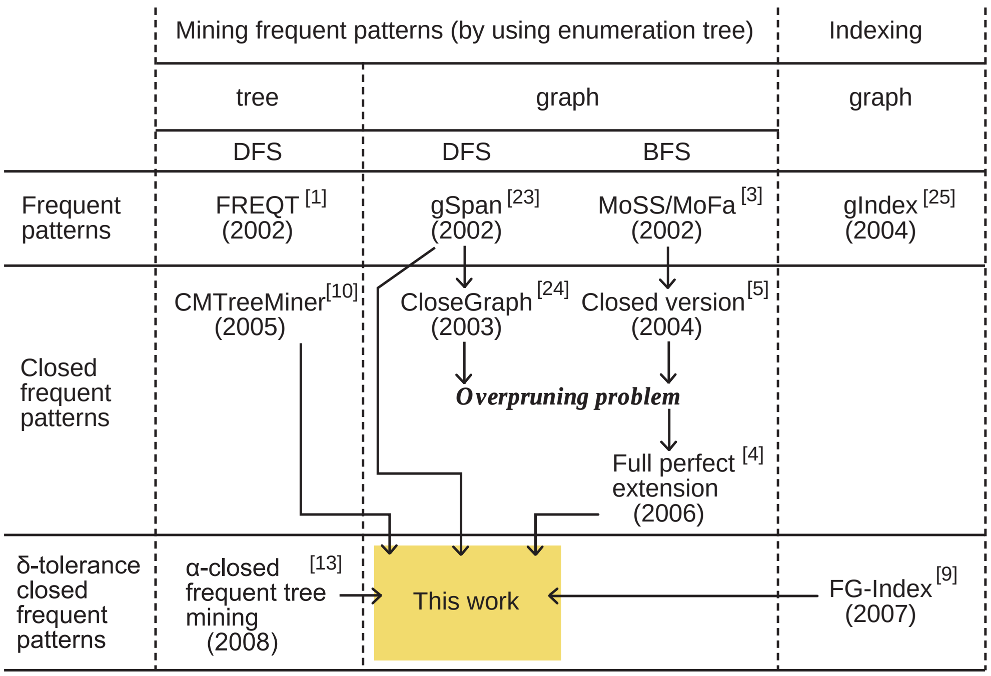


Figure 1: A map on related work

the number of outputs is larger with that minsup. Consequently it's clearly worth pursuing a time-efficient method for moderating the size of outputs in mining frequent subgraphs.

In this work, we present an efficient algorithm for mining δ-tolerance closed frequent subgraphs by incorporating a lot of pruning techniques to cut down the space of enumerating subgraphs. Our algorithm is closely related with those for mining closed frequent subgraphs (by depth-first search (DFS) [24] and by breadth-first search (BFS) [5]), which however sometimes have serious flaws on the completeness of enumerating all closed frequent graphs due to overpruning [21, 4]. Thus, in this work, to avoid the problem of overpruning, we started with formulating the problem of mining frequent subgraphs by using a general enumeration (or pattern growth) framework called "reverse-search" [2], for which the completeness of enumeration is guaranteed. We emphasize that this formulation makes the completeness and the uniqueness of frequent subgraphs clear not only in our problem but in more general subgraph enumeration from a given graph dataset. Under this framework, we develop a "partial-reverse-search" algorithm, by which traversing and pruning a search space can be clearly represented over a search tree (or an enumeration tree). We emphasize that such a well-organized search space allows our algorithm to use possible pruning rules. For example, our algorithm can incorporate so-called "right-blanket pruning" and "left-blanket pruning," which were modified from the original ones for closed frequent subtree mining [10, 13], and both pruning techniques have not been used in closed frequent subgraph mining yet. Fig. 1 shows a summary of the current literature flow of mining frequent trees and graphs and indexing graphs. One item of note is that while δ-tolerance closedness was developed for itemsets first [8] and then graphs [9] mainly in the literature of indexing, a totally similar idea, called α-closedness, was independently developed for trees [13] in the literature of bioinformatics.

We evaluated our method with a variety of real datasets of graphs, particularly chemical compounds. We first checked practical computation time of our algorithm on these datasets, under different settings of minsup, comparing our proposed algorithm with two naive methods, including the index extraction part of FG-Index [9]. Results showed that real computation time of the naive methods was drastically reduced by our algorithm, particularly for the case with minsup of lower values, such as 5%. In fact, this difference is sizable, because our method can be easily applied to cases, for which the naive methods cannot work at all practically. We then examined the number of frequent subgraphs generated by our mining method with changing the value of δ from zero to one. The result showed that our method could smoothly reduce the number of outputs from a very large number to a small size. We further checked real examples of δ-tolerance closed frequent subgraphs obtained by four methods: standard frequent subgraph mining and three different parameter values of δ-tolerance closed frequent subgraph mining: 0 (closed frequent subgraph mining), 0.2 and 1 (maximal frequent subgraph mining). We could see that subgraphs obtained by δ of zero are almost the same as those by standard frequent subgraph mining, being very redundant, while those by δ of one are too diverse. On the other hand, the subgraphs by δ of 0.2 are moderate and well-balanced. These results clearly revealed the effectiveness of our approach of efficiently mining δ-tolerance closed frequent subgraphs which are reduced from redundant frequent graphs, to find significant, key patterns out of graph data.

2 Preliminaries

Here we show the notation and concepts which are already defined in mining frequent patterns (particularly subgraphs) and will be used in our proposed algorithm.

2.1 Notation

Given two tuples $a = (a_1, a_2, \ldots, a_m)$ and $b = (b_1, b_2, \ldots, b_n)$ where pairwise order $a_i < b_i$ between i-th elements is all available, we can define *lexicographical order* $a < b$ between these two tuples if $a_i = b_i$ $(\forall i \leq m)$ and $m < n$, or if there exists j such that $a_i = b_i$ $(\forall i < j)$ and $a_j < b_j$. Let set X be the set of all tuples, and the total order $<$ on X is $x < y$ or $y < x$ for any $x \in X$, $y \in X$ and $x \neq y$.

A *graph* $G = (V, E)$ is a collection of *nodes* V and *edges* E. An edge is denoted by (u, v); u and v are *adjacent*. A graph is *undirected* if its edges are unordered pairs of distinct nodes, and is *labeled* if one of several labels is assigned to each node and each edge. We write the label of node v as label(v), and that of edge (u, v) as label$((u, v))$. A graph $G' = (V', E')$ is a *subgraph* of G, denoted by $G' \subset G$, and G is a *supergraph* of G', if $V' \subseteq V$ and $E' \subseteq E$ and an *induced subgraph* if $(u, v) \in E' \Leftrightarrow (u, v) \in E$ for any $u, v \in V'$.

A sequence of distinct edges from v_1 to v_n: $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ in G is called a *cycle* if $v_1 = v_n$. An undirected graph is *connected* if, for every distinct pair of nodes, one node can be reached from the other node along with edges. A *tree* is a connected undirected graph which contains no cycles. A *spanning tree* of an undirected graph G is a subgraph which is a tree, containing all nodes of G. A *rooted* tree is a directed graph with the *root* node having no edges leading to it, all other nodes having one edge leading to them, and no cycles. For two nodes v and w, which are connected without using the root in a directed, rooted tree, if v is closer to the root, then v is called an *ancestor* of w and w is a *descendant* of v.

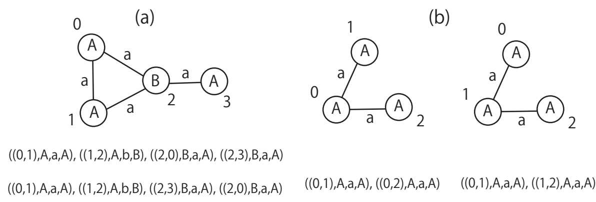

(a)
(b)
((0,1),A,a,A), ((1,2),A,b,B), ((2,0),B,a,A), ((2,3),B,a,A)
((0,1),A,a,A), ((1,2),A,b,B), ((2,3),B,a,A), ((2,0),B,a,A)
((0,1),A,a,A), ((0,2),A,a,A)
((0,1),A,a,A), ((1,2),A,a,A)

Figure 2: DFS Code Examples

A *depth-first search (DFS)* traverses all nodes in a graph. This search always proceeds to the next adjacent unvisited node until reaching a node that has no unvisited adjacent nodes, then backtracks to the previous node and continues along any as-yet unexplored edges from that node. By running a DFS, each node v has a *pre-order index*, denoted by $\mathrm{idx}(v)$, which is a time-stamp (or an index) when v is first discovered. For an edge (u, v), we can have a *pre-order index pair* $(\mathrm{idx}(u), \mathrm{idx}(v))$. Note that an edge with a pre-order index pair (i, j) is always either *tree edge* if $i < j$ or a *back edge* if $i > j$, when we consider undirected graphs with no self-loops, i.e. $i \neq j$.

In this paper, we assume that input graphs are undirected, connected and labeled, and we consider their induced subgraphs.

2.2 DFS Code for Graphs

2.2.1 Minimum DFS Code

A DFS gives an edge sequence with pre-order indices on nodes. Thus, we can order all edges in a graph as they are first discovered. For given graph G, each edge (u, v) can be a 4-tuple:

$$\big((\mathrm{idx}(u), \mathrm{idx}(v)), \mathrm{label}(u), \mathrm{label}((u, v)), \mathrm{label}(v)\big),$$

and a *DFS code* of G, denoted by $\mathrm{code}(G)$, can be an edge sequence by an ordered list of 4-tuples. Fig. 2 shows brief examples of two graphs ((a) and (b)) and their DFS codes.
Since a graph has at least one DFS code, we can examine all graphs by checking all possible DFS codes. However, since different codes can correspond to the same graph because of the *graph isomorphism problem*, we have to put equivalent DFS codes into one single code.

We then define the "order" for multiple DFS codes of a graph and take the minimum one as the representative among them. Since a DFS code is a tuple of 4-tuples, if the order between any 4-tuples is available, we can sort DFS codes in lexicographical order. For example, the order between two DFS codes in Fig. 2 (a) is decided by the order between two 4-tuples: $((2, 0), B, a, A)$ and $((2, 3), B, a, A)$. Similarly, for Fig. 2 (b), two DFS codes are sorted by the order between two 4-tuples: $((0, 2), A, a, A)$ and $((1, 2), A, a, A)$. According to this lexicographical order, we can simply choose the minimum code for our purpose. This coding by the *minimum DFS code* guarantees both completeness and uniqueness as representations of graphs. Here the completeness means that graphs can be enumerated without any oversight, while the uniqueness means that graphs can be enumerated without redundancy.

Property 1 (Minimum DFS Code [23]). *For a given graph G, the minimum DFS code of G, denoted by* $\min\{\mathrm{code}(G)\}$*, is unique and complete. Thus, it can be one canonical representation of graphs.*

Table 1: The order on pre-order index pairs

$x_1 < x_2$ (tree)	$x_2 < y_2$		$\Rightarrow$	$x \prec y$
	$x_2 = y_2$	$x_1 > y_1$	$\Rightarrow$	$x \prec y$
$y_1 < y_2$ (tree)	$x_2 = y_2$	$x_1 < y_1$	$\Rightarrow$	$x \succ y$
	$x_2 > y_2$		$\Rightarrow$	$x \succ y$
$x_1 > x_2$ (back)	$x_1 < y_1$		$\Rightarrow$	$x \prec y$
	$x_1 = y_1$	$x_2 < y_2$	$\Rightarrow$	$x \prec y$
$y_1 > y_2$ (back)	$x_1 = y_1$	$x_2 > y_2$	$\Rightarrow$	$x \succ y$
	$x_1 > y_1$		$\Rightarrow$	$x \succ y$
$x_1 < x_2$ (tree)	$x_2 \leq y_1$		$\Rightarrow$	$x \prec y$
$y_1 < y_2$ (back)	$x_2 > y_1$		$\Rightarrow$	$x \succ y$

A DFS code of a graph is called *minimal* among all DFS codes for the same graph, if this code is the minimum DFS code. We can show an important property of the minimum DFS code which allows to enumerate frequent subgraphs efficiently, by using the *prefix subcodes* of a DFS code $(a_1, a_2, \ldots, a_n)$ which can be defined as $(a_1, a_2, \ldots, a_i)$ where $i \leq n$.

Property 2. *Any prefix subcode of the minimum DFS code is also minimal for the corresponding graph.*

In order to define this canonical coding more precisely, it suffices to specify the order between any 4-tuples. Since we usually can order labels alphabetically, 4-tuples can also be sorted in lexicographical order by using the order on pre-order index pairs. Thus, the problem ends up in defining the order between any two pre-order index pairs. For given two pre-order index pairs: $x = (x_1, x_2)$ and $y = (y_1, y_2)$ $(x \neq y)$, the total order $\prec$ between any x and y can be defined as in Table 1.

Property 3 (Total Order on DFS Codes [23])**.** *The total order $\prec$ between two pre-order index pairs can define the total order between any two DFS codes, meaning that any pair of DFS codes, even* $\mathrm{code}(G)$ *and* $\mathrm{code}(G')$ *from two different graphs G and G', can be compared.*

Thus, we can search all possible subgraphs in this order, skipping corresponding DFS codes except minimum DFS codes.

2.2.2 Identifying the Minimum DFS Code

For given $c = \mathrm{code}(G)$, we can easily generate graph G from the definition of the DFS code. We write this as $G = \mathrm{graph}(c)$. Given graph G, we can directly generate the minimum DFS code $\min\{\mathrm{code}(G)\}$. Since DFS codes are defined in lexicographical order, any prefix subsequence of a code must be the minimum DFS code among all codes representing the corresponding subgraph of G. Then, starting from the 1-edge $((0,1), x, y, z)$ that the label (x, y, z) is minimal among all edges of G, we add the remaining edges one by one by DFS. Whenever we find multiple edges to be added next, we can take only the minimal one. If there are multiple minimal edges, we simply trace all of them until it turns out that they are not minimal. After adding all edges of G, we can find the minimum DFS code $\min\{\mathrm{code}(G)\}$ in the set of currently generating edge-sequences.

Hence, for identifying whether a code c is the minimum DFS code, we first generate graph $G = \text{graph}(c)$, then check whether $c = \min\{\text{code}(\text{graph}(c))\}$ or not. We note that this check can be efficient when c is not minimal, because this case, at some point, we will fail to find the prefix of c in any of the currently generating minimal edge sequences. Other heuristic pruning conditions are also possible, and if interested, please see the original paper [23]. In fact, in our method, we borrow the pruning techniques in [23] when checking whether a given code is minimal or not.

It should be noted that the graph isomorphism problem is a problem in NP, which is neither known to be solvable in polynomial time nor NP-complete. On the other hand, the more general subgraph isomorphism problem is known to be NP-complete.

2.3 δ-Tolerance Closed Frequent Subgraphs

Given a set of graphs, the *support* of a graph G is the number of graphs containing G, denoted by $\text{support}(G)$ (or $\text{support}(G|\mathcal{D})$ for given graph set $\mathcal{D}$). A *frequent* subgraph is a graph whose support is larger than or equal to a given cut-off value called *minimum support*, denoted by minsup. For efficiently mining frequent subgraphs, we can show an important property, which is used for all types of frequent patterns in common and is here shown for graphs.

Property 4 (Downward Closure)**.** *A graph is not frequent if any of its subgraphs is not frequent.*

This property indicates that the corresponding graph to any prefix subcodes of a frequent graph is also frequent. The gSpan algorithm [23], the most popular algorithm for mining frequent graphs, is based on enumeration or pattern growth, which uses this property as well as the minimum DFS codes of graphs. In brief the gSpan algorithm generates a supergraph from each graph in a depth-first manner, computing the support of each of the generated graphs and it is further grown only if the generated graph is frequent, due to the downward closure property.

A frequent subgraph is said to be *closed* if no supergraphs have the same support. Similarly, it is said to be *maximal* if no supergraph is frequent. Maximal frequent subgraphs are always closed from the definition. We introduce a parametric interpolation between a set of closed subgraphs and a set of maximal subgraphs as δ-tolerance closed subgraphs, which will be defined below.

Definition 1 (δ-Tolerance Closed Frequent Subgraph [9])**.** *A frequent subgraph G is defined to be δ-tolerance closed if no frequent supergraphs have a support larger than or equal to $(1-\delta)\cdot\text{support}(G)$.*

This definition means that the strict condition of *closedness*, which is that $\text{support}(G) = \text{support}(G')$ for G' and G ($G' \supset G$), is relaxed to a milder one, which is that $\text{support}(G') \geqslant (1-\delta)\cdot\text{support}(G)$, although this condition is still stronger than that of *maximality*, which requires only that G' is frequent. In other words, a frequent subgraph is said to be δ-tolerance closed unless $\text{support}(G') \geqslant \max((1-\delta)\cdot\text{support}(G), \text{minsup})$ for any $G' \supset G$. The parameter δ is assumed to take a value between zero and one.

Given a set of graphs and a minsup, let $\mathcal{F}, \mathcal{C}$, and $\mathcal{M}$ be the set of all frequent subgraphs, that of all closed frequent subgraphs, and that of all maximal frequent subgraphs, respectively. We can first observe the following nested hierarchies in frequent subgraphs.

Property 5. *Let $\mathcal{A}_\delta$ be the set of all δ-tolerance closed frequent subgraphs for some fixed δ. Then, it always satisfies that $\mathcal{M} \subseteq \mathcal{A}_\delta \subseteq \mathcal{C} \subseteq \mathcal{F}$ for $0 \leqslant \delta \leqslant 1$.*

Input: Graph set $\mathcal{D}$, minsup, δ
Output: All δ-tolerance closed frequent subgraphs $\mathcal{T}_\delta$ in $\mathcal{D}$

```
1: procedure BuildIndex($\mathcal{D}$, minsup, $\delta$)
2:     Enumerate all frequent subgraphs $\mathcal{F}$ using gSpan
3:     Sort $\mathcal{F}$ s.t. $\forall G_1, G_2 \in \mathcal{F}$, $G_1$ is higher ranked than $G_2$ if $G_1 \preceq G_2$.
4:     $\mathcal{T}_\delta \leftarrow \mathcal{F}$
5:     Partition $\mathcal{F}$ into $\mathcal{P}_1, \mathcal{P}_2, \dots$ where $\mathcal{P}_i$ is the set of graphs with $i$ edges.
6:     for $i \in 1, 2, \dots$ do
7:         for $G \in \mathcal{P}_i$ do
8:             for $G' \in \mathcal{P}_{i+1}$ do
9:                 if $G \subset G'$ then /* solve subgraph isomorphism */
10:                    if support($G'$) $\geqslant (1-\delta) \cdot$ support($G$) then
11:                        $\mathcal{T}_\delta \leftarrow \mathcal{T}_\delta - \{G\}$
12:                    end if
13:                end if
14:            end for
15:        end for
16:    end for
17: end procedure
```

Figure 3: A naive algorithm: δ-tolerance closed frequent subgraph extraction part in FG-Index

We can then find the following nice properties on $\mathcal{A}_\delta$.

Property 6. *A parametric family $\mathcal{A}_\delta$ is monotonically decreasing when parameter δ is increasing: $\mathcal{A}_\delta \subseteq \mathcal{A}_{\delta'}$ for any $\delta \geqslant \delta'$. Furthermore, the largest set $\mathcal{A}_0$ corresponds to the closed frequent subgraphs $\mathcal{C}$, while the smallest set $\mathcal{A}_1$ to the maximal frequent subgraphs $\mathcal{M}$.*

Property 7. *Any subgraph of a δ-tolerance closed frequent subgraph is also frequent. Thus, we can retrieve all frequent subgraphs from $\mathcal{A}_\delta$.*

We consider of enumerating all elements in $\mathcal{A}_\delta$ for a given set of graphs. Due to Property 5 and 6, the possible largest output is $\mathcal{C}$ when $\delta = 0$, while we are able to enumerate $\mathcal{M}$ even when $\delta = 1$ in the same framework.

2.4 Index Construction Part of FG-Index: Obtaining δ-Tolerance Closed Frequent Subgraphs

In order to have δ-tolerance closed frequent subgraphs, we can first think of the following primitive strategy (or a *naive* method), due to Property 5.

Method 1 (Naive)**.** *(1) Apply the gSpan algorithm to enumerate all frequent subgraphs $\mathcal{F}$, (2) for each subgraph of $\mathcal{F}$, check the condition of δ-tolerance closedness and output it if the condition is satisfied.*

The index construction part of FG-Index [9] implements this approach, and Fig. 3 shows the pseudocode of this part, which is directly extracted from [9]. In this figure, the graph set order $\preceq$ can be defined in the following manner.

Definition 2 (Graph Set Order [9])**.** *Given a set of graphs G, a graph set order $\preceq$ on G is a total order which can be defined as follows: Let $G_1, G_2 \in G$. $G_1 \preceq G_2$ if one of the following three statements is true.*

1. $\text{size}(G_1) < \text{size}(G_2)$
2. $\text{size}(G_1) = \text{size}(G_2)$ *and* $\text{support}(G_1) > \text{support}(G_2)$
3. $\text{size}(G_1) = \text{size}(G_2)$*,* $\text{support}(G_1) = \text{support}(G_2)$*, and* $h(G_1) \leqslant h(G_2)$ *where* $h : G \to \mathbb{N}$ *is an injective function that assigns a unique ID* $n \in \mathbb{N}$ *to each subgraph in* G*.*

We can further define $G_1 \prec G_2$ if $G_1 \preceq G_2$ and $G_1 \neq G_2$.

Fig. 3 shows the pseudocode of the naive method. As shown in this figure, this algorithm simply generates all frequent subgraphs and repeats checking whether each of them is δ-tolerance closed frequent subgraph or not. This strategy covers all solutions since $\mathcal{A}_\delta \subseteq \mathcal{F}$, but can be improved because we do not need to search all of $\mathcal{F}$ for enumerating $\mathcal{A}_\delta$. More concretely, while the search by the gSpan algorithm runs for $\mathcal{F}$, if we find that no descendants of the current subgraph can be δ-tolerance closed, we do not have to search the descendants in the enumeration tree. This means that we can prune the corresponding edges in the enumeration tree.

3 Reverse Search Reformulation for Enumerating Frequent Subgraphs

Before moving on to our algorithm, we first reformulate the enumeration by the gSpan algorithm in our context, giving another view to the gSpan algorithm as an example of reverse search enumeration. In the enumeration of subgraphs, we have to examine possible subgraphs one by one, avoiding 1) overlooking some subgraphs to be enumerated, and 2) checking the same subgraph in multiple times because of efficiency. However, it is not apparent in what order graphs should be examined in an efficient search.

This issue can be solved by an abstract technique which was developed in another field. In fact, enumerating all objects that satisfy a specified property is a fundamental problem not only in data mining but also in a lot of other fields such as combinatorics, computational geometry, and operations research. Avis and Fukuda [2] presented an exhaustive search technique, called *reverse search*, in a general framework which includes various enumeration problems in broader applications. In reverse search, we first define a rooted spanning tree, called *enumeration tree*, in which nodes are the set to be enumerated. In our case, an enumeration tree can be defined over $\mathcal{F}$. Once we can have an enumeration tree, we simply traverse it from the root to leaves. Since a spanning tree covers all nodes, the search examines each node (a frequent subgraph) exactly only once. Thus, we can avoid any redundant search, such as checking the same subgraph multiple times, without overlooking any necessary subgraphs. This fact can guarantee the completeness and uniqueness of the enumeration. If a node to be examined next is always given at any node, the spanning tree on $\mathcal{F}$ can be defined implicitly. More concretely, we first define a unique parent in $\mathcal{F}$ for each subgraph (child) in $\mathcal{F}$. By doing this without causing any cycles, we can generate an enumeration tree by switching the direction of examining node from the above child-to-parent to the "reverse" direction, i.e., parent-to-child. Fig. 4 gives an illustrative example of generating an enumeration tree. Thus, it is a key to uniquely define the parent of a node as well as to find an efficient way to "reverse" the direction of examining nodes.

Precisely speaking, the objects to be enumerated in our problem is the set of minimum DFS codes corresponding to $\mathcal{F}$. We can use Properties 2 and 4 to define the unique parent for a subgraph. Then, for any graph $G \in \mathcal{F}$, we can always uniquely assign its parent $G' \in \mathcal{F}$: For the minimum DFS code $c = (a_1, a_2, \ldots, a_m)$ of G, its parent G' is defined by its prefix subcodes $(a_1, a_2, \ldots, a_{m-1})$ with removing the last edge from c. We assume the special node, called the *root* denoted by $\perp$, as the parent of 1-edge codes like (a_1). Since the graph size of the parent is smaller by one edge than those of the children, this parent-child relationship does not form any cycle. Thus, by "reversing" this parent-child relationship, we can build an enumeration tree rooted at $\perp$ for *reverse search* of $\mathcal{F}$.

Property 8 (Reverse Search)**.** *For a given minimum DFS code $(a_1, a_2, \ldots, a_m)$, its children can be examined by finding all minimum DFS codes, which are $(a_1, a_2, \ldots, a_m, b)$ with one additional edge b, and whose corresponding graphs are frequent. We call this procedure a local search. Starting with root $\perp$, recursively iterating a local search in the depth-first manner completes the reverse search for enumerating $\mathcal{F}$.*

In other words, given minimum DFS code $(a_1, a_2, \ldots, a_m)$ at each recursion, for its possible children, it is sufficient to consider only the following three points: (1) whether to take a form of $(a_1, a_2, \ldots, a_m, b)$, (2) whether to be minimal, and (3) whether to be frequent. We note that the edge extension defined by (1) and (2) is also known as *rightmost extension*, which was originally defined for enumerating frequent subtrees [1]. Practically, the above procedure is done as follows: We first find all frequent 1-edge subgraphs as frequent patterns and save the locations where corresponding graphs are found in given graphs. We then attempt to "grow" the frequent pattern by adding a possible adjacent edge b and tracing all locations of the frequent patterns simultaneously (which was originally done in [3]). We note that since the above edge extension in local search can be ordered, we can attempt to add an edge in this order. This will help making enumeration efficient in mining δ-tolerance closed frequent subgraphs as described later.

Property 9 (DFS Lexicographical Ordered Tree)**.** *Since the total order between DFS codes is available, any siblings in an enumeration tree can be ordered. Thus, the enumeration tree is an ordered spanning tree.*

4 Proposed Procedure: Partial Reverse Search for Efficiently Enumerating δ-Tolerance Closed Frequent Subgraphs

Avis and Fukuda [2] introduced a simple modification of reverse search called *partial reverse search* for solving a certain class of hard optimization problems. This technique also can be applied to enumerating δ-tolerance closed frequent subgraphs $\mathcal{A}_\delta$, although our problem is an enumeration problem rather than an optimization problem. We here can consider two types of

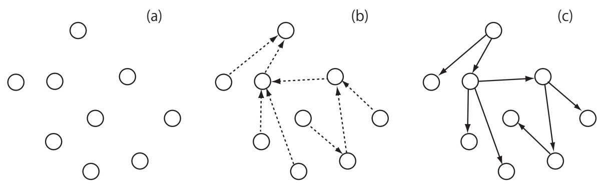


Figure 4: (a) Objects to be enumerated, (b) unique parents of nodes, and (c) reverse search: rooted spanning tree on set (a).

pruning conditions on closedness and δ-tolerance closedness. Our algorithm is then summarized in the following:

Method 2 (Proposed)**.** *(1) Apply reverse search for enumerating $\mathcal{F}$, (2) during the search procedure, at some subgraph, (2-1) if we find all of its descendants not to be closed, prune this branch of the enumeration tree; (2-2) we can further check the condition of δ-tolerance closedness and output the subgraph if it satisfies the condition.*

The feature of our proposed algorithm can be summarized into two points: pruning conditions on closedness and efficiently checking the condition on δ-tolerance closedness, which are shown in Sections 4.1 and 4.2, respectively. Section 4.1 shows a new algorithm, which is specialized to mining closed graphs and free from overpruning, and Section 4.2 shows an efficient algorithm on checking the δ-tolerance closedness.

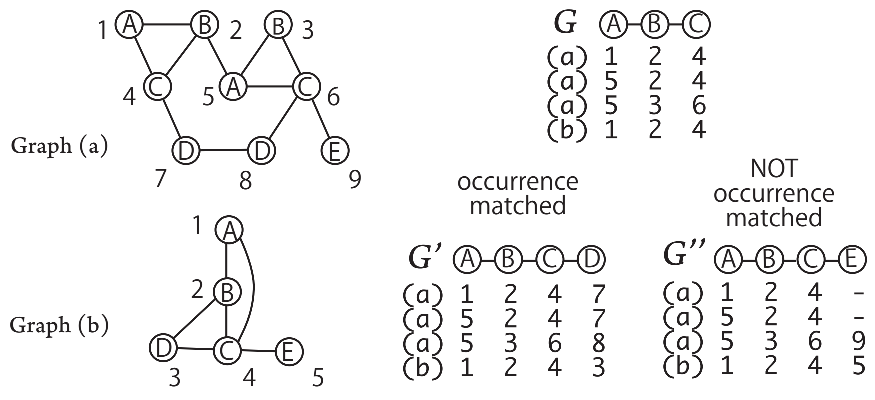
4.1 Pruning by Occurrence-Matched Graphs and Feasible Edges

We first introduce *occurrence-matched graphs*, which was originally defined in mining frequent subtrees in [10] and is also important for graphs, being equivalent to *equivalent occurrence* in [24] and another idea in [5] (Note: They extended their idea to *perfect extension* of [4] where the overpruning issue described below was avoided.). If we prune edges by using this notion only, they can include the cases which should not be pruned, resulting in overpruning. This means that we need another notion by which we can avoid overpruning. We then define *feasible edges* and attempt to use both occurrence-matched graphs and feasible edges for pruning, resulting in the idea of right- and left-blanket pruning for the case of mining closed frequent subgraphs. We emphasize that there have been no methods which realize both right- and left-blanket pruning for mining closed frequent subgraphs.

4.1.1 Occurrence-Matched Graphs and Feasible Edges

Definition 3 (1-Edge Extension)**.** *For a given graph G, a supergraph which can be generated by adding another adjacent edge e to any node of G is said to be 1-edge extension of G, denoted by $G + e$, and e is called an extended edge.*

Definition 4 (Blanket)**.** *For a given graph G, a blanket of G, denoted by $B(G)$, is a set of all possible 1-edge extensions of G. The set $B(G)$ can be divided into two subsets: the right-blanket $B_R(G)$ and the left-blanket $B_L(G)$. The right-blanket $B_R(G)$ is a set of supergraphs of G which can be the children of G in the enumeration tree. The left-blanket is defined as the complementary set of $B_R(G)$ in $B(G)$.*



G	A	B	C
(a)	1	2	4
(a)	5	2	4
(a)	5	3	6
(b)	1	2	4

G'	A	B	C	D
(a)	1	2	4	7
(a)	5	2	4	7
(a)	5	3	6	8
(b)	1	2	4	3

G''	A	B	C	E
(a)	1	2	4	-
(a)	5	2	4	-
(a)	5	3	6	9
(b)	1	2	4	5

Figure 5: An example of occurrence-matched graphs and not occurrence-matched graphs.

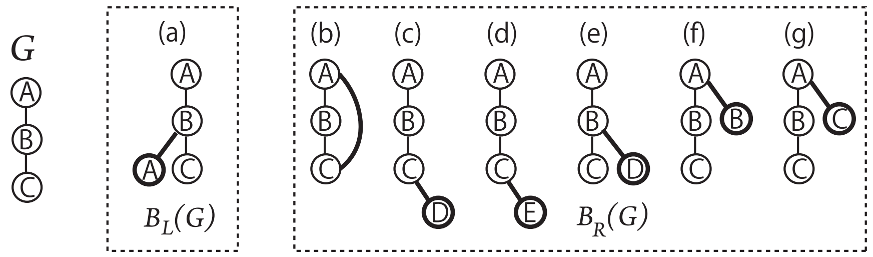


Figure 6: Left- and right-blankets of G in Fig. 5

In short, the right-blanket is a set of all 1-edge rightmost extensions of G, while the left-blanket is a set of 1-edge extensions which cannot be generated by rightmost extension from G. Here we first note that the rightmost extension is described in Section 3. Then once again we show the following property which recalls the important relationship between minimum DFS codes and corresponding nodes in the enumeration tree. We will then not mention the minimum DFS codes any more after this property.

Property 10 (Enumeration Tree with Minimum DFS Codes). *Given an enumeration tree and a graph G where $\min\{\text{code}(G)\} = (a_1, a_2, \ldots, a_n)$, if there is a node corresponding to G in the enumeration tree, a child of the node corresponding to G has minimum DFS code $(a_1, a_2, \ldots, a_n, b)$.*

We then define occurrence-matched graphs.

Definition 5 (Occurrence-Matched Graphs). *Two subgraphs G and $G' \in B(G)$ are occurrence-matched, if G' always appears at the same location as that of G in any of given graphs. We write $G \overset{OM}{\longleftrightarrow} G'$ when G and G' are occurrence-matched.*

Fig. 5 shows an illustrative example of occurrence-matched graphs. In this figure, given two graphs (a) and (b), G and G' are occurrence-matched, because G' always appears at the same location as that of G. For example, when G appears at nodes 1, 2 and 4, G' appears at nodes 1, 2, 4 and 7, and so on. On the other hand, G and G'' are not occurrence-matched, because G'' does not appear when G appears at nodes 1, 2 and 4.

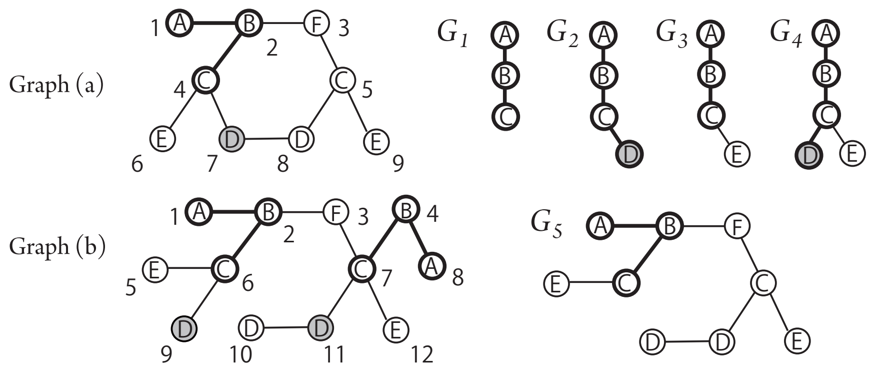


Figure 7: Feasible edges for pruning.

For trees, we can define pruning conditions by using blanket and occurrence-matched graphs only [10]. However, graphs have intrinsically cycles, thus letting us need another concept to define pruning conditions for graphs. We first define a *bridge* to be an edge where the number of connected components of the graph increases when this edge is removed. This means that bridges cannot be a part of any cycle in a graph. For example, in Fig. 5, bridges are C6-E9 in (a) and C4-E5 in (b) only. Using bridges and the back edge defined in Section 2.1, we can define *feasible edges*, which must be taken care of when our pruning strategy is applied.

Definition 6 (Feasible Edges)**.** *For a 1-edge extension $G + e$, edge e is feasible if it is a back edge, or if it is a tree edge and all corresponding edges of e in graphs containing G are bridges.*

We illustrate examples of feasible edges. Fig. 6 shows all sorted left- and right-blankets of G in Fig. 5. For each 1-edge extension $G + e \in B(G)$ in Fig. 6, e is drawn by a thick line. In this figure, the extended edges in (b) and (d) only are feasible, since the extended edge in (b) is a back edge, and in Fig. 5 all edges corresponding to the extended edge of (d) are bridges in both two graphs ((a): C6-E9 and (b): C4-E5).

4.1.2 Right-Blanket Pruning

We now consider the pruning conditions based on these notions, and the first condition is given for right-blanket.

Proposition 1 (Right-Blanket Pruning)**.** *For given G, if we have $G' = G + e \in B_R(G)$ such that $G' \xleftrightarrow{OM} G$ and e is feasible, then graph $G + e', e' \neq e$ and any of its descendants in the enumeration tree can be pruned.*

Proof. In the enumeration tree, G can be replaced with G' if $G'(= G + e) \xleftrightarrow{\text{OM}} G$ and e is feasible. Then, since graph $G + e', e' \neq e$ and any of its descendants cannot be closed, we can prune all 1-edge extensions of G except G'. □

We further illustratively explain this pruning condition, focusing on the difference between frequent subgraphs and frequent subtrees. Fig. 7 shows an example of mining from two graphs with minsup = 2. For G_1:A-B-C and G_2:A-B-C-D, we can see $G_1 \xleftrightarrow{\text{OM}} G_2$ because A-B-C-D always appears at the same locations of A-B-C: A1-B2-C4 in (a), A1-B2-C6 and A8-B4-C in (b). So any A-B-C can be always replaced with A-B-C-D, implying that we do not have to think about G_1 any more, resulting in that branches corresponding to all 1-edge extensions of G_1, i.e. $B(G_1)$, in the enumeration tree can be all pruned except G_2. For example, G_3 from G_1 is not closed because of G_4 and we can obtain G_4 also from G_2. Thus, if input graphs are trees we can use $G_1 \overset{\text{OM}}{\longleftrightarrow} G_2$ only for pruning, but this is not necessary enough for graphs which include cycles. In fact, we may miss closed subgraphs such as G_5 in Fig. 7, if we use $G_1 \overset{\text{OM}}{\longleftrightarrow} G_2$ only.

We further explain why G_5 can be overlooked even though there are no exceptions of this pruning rule for trees. Let us focus on G_2 and G_1. We first think about the inputs which are all trees. If an input has both G_1 and G_2 at the same location, D can be reached from any node of G_1, i.e. A, B and C, by using only C-D, because if we could reach D from A without using C-D, this means that there is a cycle which should not be in trees. For example, D9 and D11 in Fig. 7 (b) would not be examined to be added to G_1, if there is not C-D. We then think about the case that inputs have graphs with cycles. When an input has both G_1 and G_2 at the same location, D may be reached from a node of G_1, say B, without using C-D, because we may have two different ways from B to D via a cycle, which is allowed in graphs. For example, in Fig. 7 (a), D7 can be reached from B2 via B2-C4-D7 or B2-F3-C5-D8-D7. This means that in G_5, we can reach D from B via B-F-C-D-D even if we do not have C-D. Thus, this fact contradicts the above rule that "any A-B-C can be always replaced with A-B-C-D". In fact, G_5 cannot be generated from G_2 because A-B-C-D is not in G_5, but can be from G_1, and so we need to save G_1. More generally, on each input which has both G and $G + e$ ($e = (u, v)$) at the same location, we have to ensure that any node of G cannot reach to v without e, i.e. that e is a bridge. If e is a bridge for all inputs, we can prune the branch of each $G' \in B(G)$ ($G' \neq G + e$). We here note that when e is a back edge (For example, (b) in Fig. 6), since v is already included in G, we do not need to consider whether to be a bridge. Thus we need two conditions for right-blanket pruning: $G'(= G + e) \overset{\text{OM}}{\longleftrightarrow} G$ and e is feasible.

4.1.3 Left-Blanket Pruning

We can have the following:

Lemma 1. *In DFS of the enumeration tree, $G' \in B_L(G)$ is examined before G is examined:*

$$\min\{\text{code}(G')\} < \min\{\text{code}(G)\} \textit{ for } G' \in B_L(G)$$

Proof. This can be directly derived from Property 9 and the definition of minimum DFS codes. □

This means that if a graph and its 1-edge extension in the left-blanket are occurrence-matched and the adjacent edge between these two is feasible, the graph can be pruned.

Proposition 2 (Left-Blanket Pruning)**.** *For given G, if we have $G' = G + e \in B_L(G)$ such that $G' \overset{OM}{\longleftrightarrow} G$ and e is feasible, then both G and any of its descendants in the enumeration tree can be pruned.*

Proof. This just follows Proposition 1 and Lemma 1. □

Hereafter we write $B_L^{f\text{-OM}}(G) := \{G + e \in B_L(G) \mid G \overset{\text{OM}}{\longleftrightarrow} G + e\}$, where e is feasible. For checking $B_L^{f\text{-OM}}(G) \neq \varnothing$ in the left-blanket pruning, we can derive a practically time-efficient procedure (which is summarized into the upper part of Fig. 10) as follows: We first note that each G' ($\in B_L^{f\text{-OM}}(G)$) must appear at *all* occurrences (locations) of G in a given dataset $\mathcal{D}$. This means that $B_L^{f\text{-OM}}(G)$ can be obtained by 1) first checking all possible 1-edge extensions of G at each occurrence of G, 2) keeping $G + e$ ($\in B_L(G)$) if e is feasible, and 3) taking the *intersection* of the stored 1-edge extensions over all locations of G. From an opposite viewpoint, this means that once we find G' ($= G + e$) where e is not feasible at some location of G, G' must not be in $B_L^{f\text{-OM}}(G)$. We can then prepare a tentative graph set Y_i (which contains all $G+e$ ($\in B_L(G)$) with feasible edge e for the i-th location of G) and run the following procedure by using a tentative graph set X with Y_i to implement the above idea: 1) At $i = 1$, $X \leftarrow Y_i$ and 2) repeat $X \leftarrow X \cap Y_i$, incrementing i, and once if $X = \varnothing$, it is guaranteed that $B_L^{f\text{-OM}}(G) = \varnothing$, meaning that G does no longer satisfy $B_L^{f\text{-OM}}(G) \neq \varnothing$ and we can soon quit checking $B_L^{f\text{-OM}}(G) \neq \varnothing$. The upper part of Fig. 10 shows the pseudocode of the above procedure on checking $B_L^{f\text{-OM}}(G) \neq \varnothing$.

4.2 Condition of δ-Tolerance Closedness

We first show the testing condition on δ-tolerance closedness.

Proposition 3 (δ-Tolerance Closedness Testing)**.** *For given G, let $B^\delta(G) \subseteq B(G)$ be a set defined by*

$$B^\delta(G) := \{G' \in B(G) \mid \text{support}(G') \geqslant \max((1-\delta)\cdot\text{support}(G), \text{minsup})\}.$$

Then, G is δ-tolerance closed if and only if $B^\delta(G) = \varnothing$.

Proof. If G is δ-tolerance closed, the statement $B^\delta(G) = \varnothing$ simply follows Definition 1. Thus, we show the reverse implication: Suppose that $B^\delta(G) = \varnothing$. Then, $G' \in B(G)$ satisfies $\text{support}(G') < \max((1-\delta)\cdot\text{support}(G), \text{minsup})$. On the other hand, for any supergraph G'' of G, there always exists $G' \in B(G)$ that $G \subseteq G' \subseteq G''$, meaning $\text{support}(G) \geqslant \text{support}(G') \geqslant \text{support}(G'')$. Therefore, it always holds that $\text{support}(G'') < \max((1-\delta)\cdot\text{support}(G), \text{minsup})$ for any supergraph G'' of G, and we can conclude that G is δ-tolerance closed. □

In the procedure for checking $B_L^{f\text{-OM}}(G) = \varnothing$ of Section 4.1, we used the fact that $B_L^{f\text{-OM}}(G)$ can be obtained by an intersection of the sets over all locations of G. On the other hand, $B^\delta(G)$ cannot be defined as an intersection of sets. We can raise differences between $B^\delta(G)$ and $B_L^{f\text{-OM}}(G)$ as follows: 1) We need to consider not only left-blanket but also right-blanket. 2) e is not needed to be feasible for $G + e \in B^\delta(G)$. 3) We do not have to check each location of G, and instead, given graph set $\mathcal{D}$ we can check G for each $J \in \mathcal{D}$. 4) We cannot stop even if some tentative set X becomes empty because $B^\delta(G)$ is not an interaction of sets. For formally presenting our procedure, we first introduce a general notion, *partial support*, which gives a basis for developing an efficient procedure.

Definition 7 (Partial Support)**.** *For given graph set $\mathcal{D}$, the partial support of subgraph G in $\mathcal{S} \subseteq \mathcal{D}$ is defined by the number of graphs containing G in $\mathcal{S}$, denoted by* $\text{support}(G \mid \mathcal{S}) := |\{J \in \mathcal{S} \mid G \subseteq J\}|$.

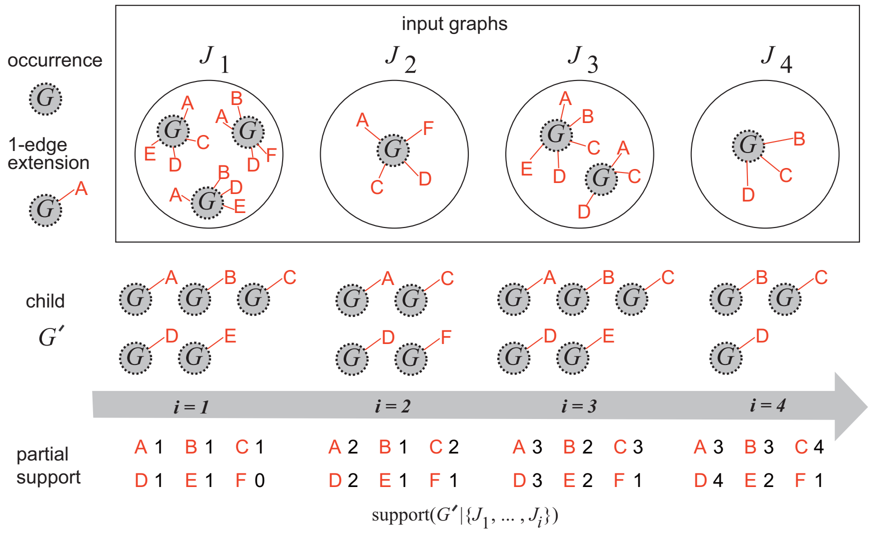


Figure 8: An example of graph G and partial support.

Let $J_1, \ldots, J_i, \ldots, J_{\mathrm{support}(G)}$ be graphs in $\mathcal{D}$ containing G. Fig. 8 shows one example of $\mathcal{D}$ with four graphs $J_1, \ldots, J_4$. In this figure, assuming that there are only six types of nodes A to F for graph G, we can consider six types of 1-edge extensions of graph G. From J_1 to J_4, we can then count each of six types of 1-edge extensions, resulting in that we can have the partial support of each possible 1-edge extension, $\mathrm{support}(G'|\{J_1, \ldots, J_i\})$, for each G' of six types.

We can implement a practically fast procedure for checking $B^\delta(G) = \varnothing$, by using tentative graph sets Y_i (which is the set of graphs in $B(G)$ for J_i) and X: 1) At $i = 1$, $X \leftarrow Y_i$, and 2) with increasing i, repeat $X \leftarrow X \cup Y_i$ for i. We note that this time we update X by $X \leftarrow X \cup Y_i$, instead of $X \leftarrow X \cap Y_i$. This means that the size of X does not monotonically decrease but increases. However, for X, we can use a nice property of G, which is the following upper bound of the partial support of G which can be used in improving the efficiency of testing the δ-tolerance closedness.

Lemma 2. *Let $J_1, \ldots, J_{\mathrm{support}(G)}$ be graphs containing G. Suppose we check each $\{J_i\}$ from $i = 1$ to* $\mathrm{support}(G)$*. At i, for G' $(\in B(G))$, we can see that $G' \notin B^\delta(G)$ if G' satisfies*

$$\mathrm{support}(G' \mid \{J_1, \ldots, J_i\}) < i - \mathrm{support}(G) + \max((1-\delta)\cdot \mathrm{support}(G), \mathrm{minsup})$$

Proof. For $G' \in B(G)$, we can obtain the following upper bound.

$$\begin{aligned}
\mathrm{support}(G') &= \mathrm{support}(G' \mid \{J_1, \ldots, J_i\}) + \mathrm{support}(G' \mid \{J_{i+1}, \ldots, J_{\mathrm{support}(G)}\}) \\
&\leqslant \mathrm{support}(G' \mid \{J_1, \ldots, J_i\}) + |\{J_{i+1}, \ldots, J_{\mathrm{support}(G)}\}| \\
&= \mathrm{support}(G' \mid \{J_1, \ldots, J_i\}) + \mathrm{support}(G) - i.
\end{aligned}$$

If G' $(\in B(G))$ is in $B^\delta(G)$, it needs that $\mathrm{support}(G') \geqslant \max((1-\delta)\cdot\mathrm{support}(G), \mathrm{minsup})$. Hence, if $\mathrm{support}(G' \mid \{J_1, \ldots, J_i\}) < i - \mathrm{support}(G) + \max((1-\delta)\cdot\mathrm{support}(G), \mathrm{minsup})$ holds, then the above upper bound must be less than $\max((1-\delta)\cdot\mathrm{support}(G), \mathrm{minsup})$. Then this naturally leads to $\mathrm{support}(G') < \max((1-\delta)\cdot\mathrm{support}(G), \mathrm{minsup})$, which indicates that $G' \notin B^\delta(G)$. ☐

To implement a time-efficient procedure, we need to keep both X and the partial support of all subgraphs in X during scanning over input graphs. At each i, we can then apply Lemma 2 to X to remove G', which satisfies Lemma 2. That is, we can run the following steps for each i, 1) we check all G' $(= G + e \in B(G))$ for all locations of G to be in Y_i, 2) X is updated by merging with Y_i as $X \leftarrow X \cup Y_i$, and 3) if $G' \in X$ satisfies Lemma 2, G' is removed.

We can raise an example of this procedure by using Fig. 8 where $\text{support}(G) = 4$. In this figure, if minsup is 2 and $\delta = 0.3$, $\text{support}(G')$ must be larger than 2.8, according to Proposition 3 (Note that $2.8 = \max((1 - 0.3) \times 4, 2)$). We can scan $J_i \in \mathcal{D}$ from $i = 1$ to 4, to compute the partial support of each G' at each i. At $i = 3$, we can find the partial support of G', i.e. $\text{support}(G'|\{J_1, J_2, J_3\})$, is only 1 for G' with the additional node labeled with F. We then do not have to consider this G' at $i = 4$, because $\text{support}(G')$ cannot be 3 or larger, even if this G' is in J_4. This is formalized in Lemma 2.

We note that for $B^{\delta}(G)$, we cannot stop even if $X = \varnothing$ at some i, since even if $X = \varnothing$, we cannot see that $B^{\delta}(G) = \varnothing$, because $B^{\delta}(G)$ is not an intersection over all locations. This is an important difference from that $B_L^{f\text{-OM}}(G) = \varnothing$. This leads the following proposition regarding the stopping conditions of scanning $\mathcal{D}$.

Proposition 4. *For each i, X is updated by merging with Y_i, and Lemma 2 is applied to X for removing G' in X. We can then terminate if either the following (i) or (ii) is satisfied, under the assumption that* $\text{minsup} > 0$*: (i) $X = \varnothing$ and $i > \text{support}(G) - \max((1-\delta) \cdot \text{support}(G), \text{minsup})$ because $B^{\delta}(G) = \varnothing$, (ii) $X \neq \varnothing$ and $\max_{G' \in X}\{\text{support}(G' \mid \{J_1, \ldots, J_i\})\} > \max((1-\delta) \cdot \text{support}(G), \text{minsup})$ because $B^{\delta}(G) \neq \varnothing$.*

Proof. An element (graph) of X must be in graphs of $\{J_1, \ldots, J_i\}$. This directly means that if $X = \varnothing$, it is guaranteed that there are no graphs which are in $B^{\delta}(G)$ and in $\{J_1, \ldots, J_i\}$. However, there might be a graph G' which is not in any of $\{J_1, \ldots, J_i\}$ but in any of $\{J_{i+1}, \ldots, J_{\text{support}(G)}\}$, and satisfies $\text{support}(G') \geqslant \max((1-\delta) \cdot \text{support}(G), \text{minsup})$. This is why we cannot immediately conclude $B^{\delta}(G) = \varnothing$ even if $X = \varnothing$ at i. We thus need to preclude the possibility that a graph like the above G' exists.

When $X = \varnothing$ at i, since the G' is not in any of $\{J_1, \ldots, J_i\}$, we can have the following upper bound of the support of G'.

$$\begin{aligned} \text{support}(G') &= \text{support}(G' \mid \{J_{i+1}, \ldots, J_{\text{support}(G)}\}) \\ &\leqslant |\{J_{i+1}, \ldots, J_{\text{support}(G)}\}| = \text{support}(G) - i \end{aligned}$$

This means that if $i > \text{support}(G) - \max((1-\delta) \cdot \text{support}(G), \text{minsup})$, the upper bound of $\text{support}(G')$ is given by

$$\text{support}(G') < \max((1-\delta) \cdot \text{support}(G), \text{minsup})$$

That is, there is not any graph which is in $B^{\delta}(G)$ or satisfies $\text{support}(G') \geqslant \max((1-\delta) \cdot \text{support}(G), \text{minsup})$. As a consequence, the statement (i) is proven to be true.

On the other hand, there is another way to terminate the procedure by finding $B^{\delta}(G) \neq \varnothing$. That is, if we found $\text{support}(G' \mid \{J_1, \ldots, J_i\}) > \max((1-\delta) \cdot \text{support}(G), \text{minsup})$ for some $G' \in X$, then we can conclude $B^{\delta}(G) \neq \varnothing$ without considering the remaining $\{J_{i+1}, \ldots, J_{\text{support}(G)}\}$. This is because $\text{support}(G') \geqslant \text{support}(G' \mid \{J_1, \ldots, J_i\})$. The statement (ii) simply follows this observation. □

```
Input: Graph set $\mathcal{D}$, minsup, $\delta$
Output: All $\delta$-tolerance closed frequent subgraphs $\mathcal{A}_\delta$ in $\mathcal{D}$
 1: procedure EnumDTolClosed($\mathcal{D}$, minsup, $\delta$)
 2:     scan $\mathcal{D}$ once to mark all bridges with DFS
 3:     $\mathcal{A}_\delta \leftarrow \varnothing$
 4:     $\mathcal{E} \leftarrow$ all possible 1-edge subgraphs in $D$
 5:     for $G \in \mathcal{E}$ in DFS order do
 6:         LocalSearch($G$, $\mathcal{D}_G$, minsup, $\delta$, $\mathcal{A}_\delta$)
 7:     end for
 8: end procedure

 9: procedure LocalSearch($G$, $\mathcal{D}_G$, minsup, $\delta$, $\mathcal{A}_\delta$)
10:     Let $c$ is the passing DFS code for $G$
11:     if support($G$) $<$ minsup : return                                  ▷ Property 4
12:     if $c \neq \min\{\text{code}(G)\}$ : return                            ▷ Property 8
13:     if NonEmptyFeasibleOM($G$, $\mathcal{D}_G$) : return                  ▷ Proposition 2
14:     if IsDTolClosed($G$, $\mathcal{D}_G$, minsup, $\delta$) : $\mathcal{A}_\delta \leftarrow \mathcal{A}_\delta \cup G$   ▷ Proposition 3
15:     for $G + e \in B_R(G)$ in DFS order do
16:         LocalSearch($G + e$, $\mathcal{D}_{G+e}$, minsup, $\delta$, $\mathcal{A}_\delta$)
17:         if $G \overset{\text{OM}}{\longleftrightarrow} G + e$ where $e$ is feasible : return   ▷ Proposition 1
18:     end for
19: end procedure
```

Figure 9: The δ-tolerance closed subgraph mining algorithm

The lower part of Fig. 10 shows the pseudocode, which implements Lemma 2 and Proposition 4.

One possible, additional remark is the following: When $\delta = 0$, X equals to the intersection of 1-edge extensions of G over all graphs in $\mathcal{D}$. Thus our approach is a way to improve the efficiency in mining closed frequent subgraphs and has not been implemented so far, meaning that our efficient procedure includes this case as a special example for $\delta = 0$.

4.3 Proposed Algorithm

Finally, gathering the pieces of work, we now show the pseudocode of our algorithm in Figs. 9 and 10, where $\mathcal{D}_G := \{G' \in \mathcal{D} \mid G \subset G'\}$. Note that in the pseudocode, we begin with graphs having only one edge (instead of graphs having only one node), although starting with graphs with one node only is possible in principle. At line 2, we first check whether each edge is a bridge or not. Since we assume input graphs are connected, a trivial way to detect all bridges in a graph is to repeat removing each edge and listing the number of connected components by DFS. However, the time complexity of this method for a graph with m edges reaches roughly $O(m^2)$. Instead we use a well-known linear-time algorithm by DFS [19], keeping the computational complexity at $O(m)$. Since all bridges are marked at line 2, we can check whether each extended edge is feasible in lines 13 and 17. In local search, we first check the downward closure property (line 11), and the minimum DFS code (line 12). We can then run the left-blanket pruning (line13), the δ-tolerance

```
procedure NonEmptyFeasibleOM($G$, $\mathcal{D}_G$)                    ▷ Check $B_L^{f\text{-OM}}(G) \neq \varnothing$
    $X \leftarrow \varnothing$; $i \leftarrow 1$
    for each location of $G$ appearing in $\mathcal{D}_G$ do
        $Y_i \leftarrow \varnothing$
        for each of all possible $G + e \in B_L(G)$ do
            if $e$ is a back edge or a bridge: $Y_i \leftarrow \{G + e\} \cap Y_i$
        end for
        if $i = 1$ then
            $X \leftarrow Y_1$
        else
            $X \leftarrow X \cap Y_i$
            if $X = \varnothing$ : return false
        end if
        $i \leftarrow i + 1$
    end for
    return true
end procedure

procedure IsDTolClosed($G$, $\mathcal{D}_G$, minsup, $\delta$)                  ▷ Check $B^\delta(G) = \varnothing$
    $X \leftarrow \varnothing$
    $\theta \leftarrow \max((1 - \delta) \cdot \mathrm{support}(G), \mathrm{minsup})$
    $m \leftarrow 0$ ; $i \leftarrow 1$
    for each graph $D_i$ containing $G$ in $\mathcal{D}_G$ do
        $Y_i \leftarrow \{G + e \in B(G) \mid G + e \text{ in } D_i\}$
        $X \leftarrow X \cup Y_i$
        for $G' \in X$ do
            $\mathrm{sup}_i \leftarrow \mathrm{support}(G' \mid \{D_1, \ldots, D_i\})$
            if $\mathrm{sup}_i < i - \mathrm{support}(G) + \theta$ then $X \leftarrow X \setminus \{G'\}$      ▷ Lemma 2
            if $\mathrm{sup}_i > m$ then $m \leftarrow \mathrm{sup}_i$
        end for
        if $X = \varnothing$ and $i > \mathrm{support}(G) - \theta$ : return true      ▷ Proposition 4 (i)
        if $X \neq \varnothing$ and $m \geqslant \theta$ : return false              ▷ Proposition 4 (ii)
        $i \leftarrow i + 1$
    end for
end procedure
```

Figure 10: Pseudocodes of subroutines in Fig. 9

closedness testing (line 14) and the right-blanket pruning (lines 15-17). At lines 5 and 15, 'in DFS order' refers to Property 3. Fig. 10 shows the pseudocodes of subroutines in the part of Fig. 9.

Table 2: Graph datasets used in our experiments.

Name	# graphs	Ave. # nodes	Max. # nodes	Ave. # edges	Max. # edges	URL: http://
DrugBank	976	25.3	101	27.1	103	redpoll.pharmacy.ualberta.ca/drugbank/
Mutag	188	17.9	28	19.8	33	chemcpp.sourceforge.net/
PTC	417	14.4	64	14.5	71	www.predictive-toxicology.org/ptc/
CPDB	684	14.1	90	14.6	96	potency.berkeley.edu/cpdb.html
HIV-CA	423	39.6	189	42.3	196	dtp.nci.nih.gov/docs/aids/aids_data.html
HIV-CM	1081	31.8	222	34.3	234	dtp.nci.nih.gov/docs/aids/aids_data.html
HIV-CI	41185	25.3	214	27.3	251	dtp.nci.nih.gov/docs/aids/aids_data.html

5 Experimental Results

5.1 Data

We used seven datasets all of which are libraries of chemical compounds. Table 2 shows the detail of each dataset and the website from which each dataset can be retrieved. Throughout the datasets, we deleted nodes labeled by hydrogens and edges extending to hydrogens.

5.2 Implementation

Since we cannot have another implementation for mining δ-tolerance closed frequent subgraphs, we checked the validity of our implementation in the following manner: We first implemented the gSpan algorithm by using our idea of reverse search and confirmed the number of outputs of our implementation was the same as that by the publicly available binary of the gSpan algorithm [23]. We then implemented the naive algorithm or the index extraction part of FG-Index, having the output of the gSpan algorithm as its input. We then implemented our proposed algorithm on mining δ-tolerance closed frequent subgraphs and confirmed that the number of our outputs is totally consistent with that of the naive algorithm. Finally we confirmed that the number of outputs of our algorithm with $\delta = 0$, i.e. closed frequent subgraphs, was consistent with that of closed frequent subgraphs outputted by ParMol [18].

5.3 Computation Time

Fig. 11 shows the real computation time of our proposed algorithm, comparing with two other algorithms for generating δ-tolerance closed frequent subgraphs, for all seven datasets in Table 2, with changing minsup. The two other competing methods are: 1) Naive: the naive method whose pseudocode is shown in Fig. 3, i.e. generating frequent subgraphs first (by the gSpan algorithm) and then running a post-processing regarding δ-tolerance closedness, and 2) CNaive: generating closed frequent subgraphs first and then running a post-processing part, which is almost[1] the same as the corresponding part[2] of the naive method. Thus the difference of these two is that

[1]The difference in the post-processing parts of Naive and CNaive is caused by the following: for frequent subgraphs, we just check the δ-tolerance closedness on the children of each node (corresponding to a frequent subgraph) in the enumeration tree, while for closed frequent subgraphs, a child of a node (corresponding to a frequent subgraph) might be removed by closedness already and this case we need to check its children further.

[2]We note that our implementation of the post-processing part for both Naive and CNaive is very fast, since its most time-consuming part is that for dealing with the subgraph isomorphism problem, which can be however solved by using minimum DFS codes efficiently. That is, this problem can be solved by checking whether the minimum

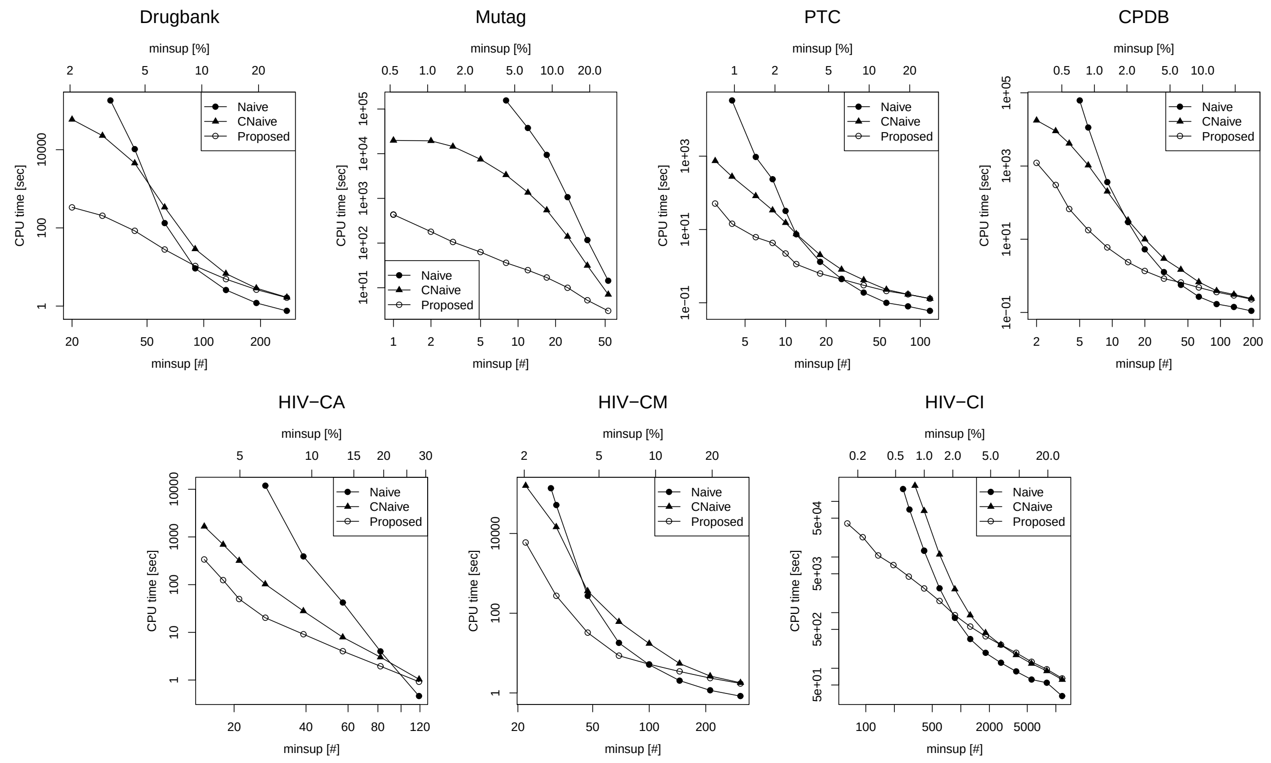


Figure 11: Real computation time for enumerating δ-tolerance closed frequent subgraphs, being varied by minsup. Each line with white circles shows the real computation time of our proposed algorithm, while lines with triangles and black circles show that of CNaive and Naive, respectively. Results are shown for seven datasets in Table 2.

Naive generates frequent subgraphs first while CNaive generates closed frequent subgraphs first. For all cases, we fixed δ at 0.2 and used a server with 64-bit Linux, dual-core AMD Opteron processor 2222SE and a memory of 48GBytes. We note that Naive is exactly the index extraction part of FG-Index [9]. Out of the seven datasets, in Mutag, our proposed algorithm was clearly much more time-efficient than the other two methods for all cases of minsup. For example, for the minsup of 10%, our algorithm is around 10^2 and more than ten times faster than Naive and CNaive, respectively. In the other six datasets, for higher values of minsup, such as 10 to 20%, the amount of computation time of Naive (CNaive) were comparable with our proposed algorithm. However, as the minsup was reduced, the computational efficiency of our proposed algorithm was significantly better than both Naive and CNaive. For example, for DrugBank, all three methods need around five to ten seconds for the minsup of 10%, while for the minsup of 5%, our method needed less than 100 seconds but Naive (and CNaive) needed around 10,000 seconds, the difference between our method and Naive (or CNaive) reaching approximately 10^2

DFS code of one of the two inputs corresponds to a node of the enumeration tree of the other, since each node of an enumeration tree corresponds to a minimum DFS code. We emphasize that in this experiment, both Naive and CNaive were implemented to run as fast as possible.

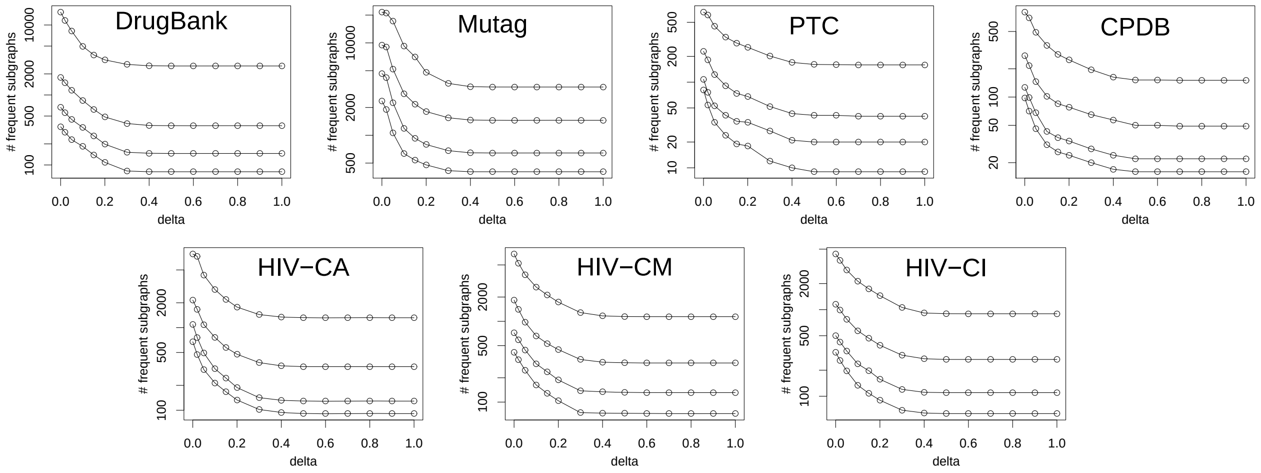


Figure 12: The number of δ-tolerance frequent subgraphs being varied by δ is shown for seven datasets in Table 2. The four lines of each dataset were obtained when minsup was 5, 10, 15 and 20% from the top to the bottom.

times. In fact, for all seven datasets, for smaller values of minsup the computational efficiency of our proposed algorithm was more pronounced and more sizable, comparing with both Naive and CNaive. These results clearly indicates the effectiveness of our method in terms of practical computation time.

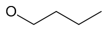
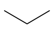
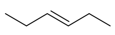
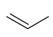
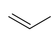
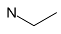
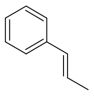
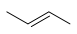
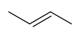
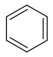
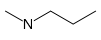
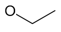
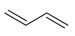
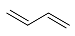
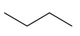
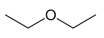
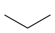
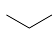
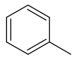
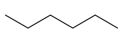
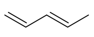
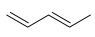
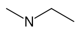
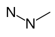
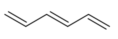
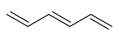
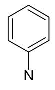
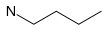
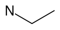
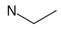
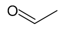
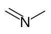
5.4 Output Reduction

Fig. 12 shows the results obtained by examining the number of outputted δ-tolerance closed frequent patterns with varying δ for all seven datasets shown in Table 2. In Fig. 12, for each dataset, four lines were obtained when minsup was 5, 10, 15 and 20% from the top to the bottom. We first note that $\delta = 0$ and 1 are equal to closed and maximal frequent subgraphs, respectively. We can easily see that a large number of frequent patterns at δ of 0 is smoothly reduced by δ to a small number at δ of 1. For example, when minsup was 5% for DrugBank, larger than 10,000 frequent patterns at δ of 0 were reduced to around 2,000 at δ of 1. The fact that the number of frequent subgraphs can be controlled by δ can be seen for all cases in Fig. 12. Another finding was that the number of subgraphs when δ=0.4 or larger was almost the same as that of maximal frequent subgraphs in all cases.

5.5 δ-tolerance Closed Frequent Subgraphs

We then checked actual frequent subgraphs, i.e. outputs, of mining methods, sorting them by their supports. Table 3 shows the top 10 frequent subgraphs obtained from CPDB by the following four methods: mining frequent subgraphs (or "Freq"), mining closed frequent subgraphs (δ=0 or "Closed"), mining δ-tolerance closed frequent subgraphs at δ of 0.2 (or "δ20") and mining maximal frequent subgraphs (δ=1 or "Maximal"), where minsup is set at 10% for all four methods. We can easily see that the subgraphs in Freq were all small and very redundant and must be boring to

Table 3: Top 10 frequent subgraphs from CPDB.

Rank	Frequent mining (Freq)		δ=0 (Closed)		δ=0.2 (δ20)		δ=1 (Maximal)	
	Support	Graph	Support	Graph	Support	Graph	Support	Graph
1	618		618		618		97	
2	468		468		401		95	
3	456		456		369		92	
4	429		429		354		92	
5	410		410		309		88	
6	401		401		265		87	
7	401		401		242		86	
8	398		398		205		83	
9	369		369		196		83	
10	369		369		196		82	

experts. These subgraphs were totally the same as those in Closed, implying that mining closed frequent subgraphs did not work well for reducing redundant subgraphs. On the other hand, this set was mildly changed to a more attractive one in δ20. For example, in δ20, we could find four graphs with two edges: C-C-C (ranked 2nd), N-C-C (ranked 3rd), O-C-C (ranked 5th) and O=C-C (ranked 10th), which were not overlapped with each other. This tendency was true of two three edge subgraphs, C-C-C-C (ranked 6th) and C-N-C-C (ranked 8th). They are a good summary of top ten frequent subgraphs (Freq and Closed). The subgraphs in Maximal were drastically different from each other. These subgraphs might be interesting in some sense, but their supports were all small and some graphs were very special, e.g. S-C (ranked 5th) and N-N-C (ranked 8th). From these results, we can say that mining δ-tolerance closed frequent subgraphs allows to flexibly change the size of outputs, improving the understandability of data mining.

5.6 Discussion

We compared the computation time of our method with that of Naive and CNaive, showing the results in Fig. 11. This figure showed that although the computation time of the three methods were comparable under relatively higher values of minsup, say around 20%, our proposd method became more time-efficient than Naive and CNaive, for lower values of minsup, say 5% of less, for all seven datasets. Particularly, the computation time of Naive was improved by CNaive for six out of all seven datasets, and further improved by our proposed method for all seven cases. The performance improvement depends upon each dataset, since for example if a dataset contains many similar graphs, the number of its δ-closed frequent subgraphs can be smaller. In Fig. 11, the advantage of the proposed method over CNaive was rather slight for PTC and HIV-CA, comparing with other datasets, even for smaller values of minsup (less than 5%). We then

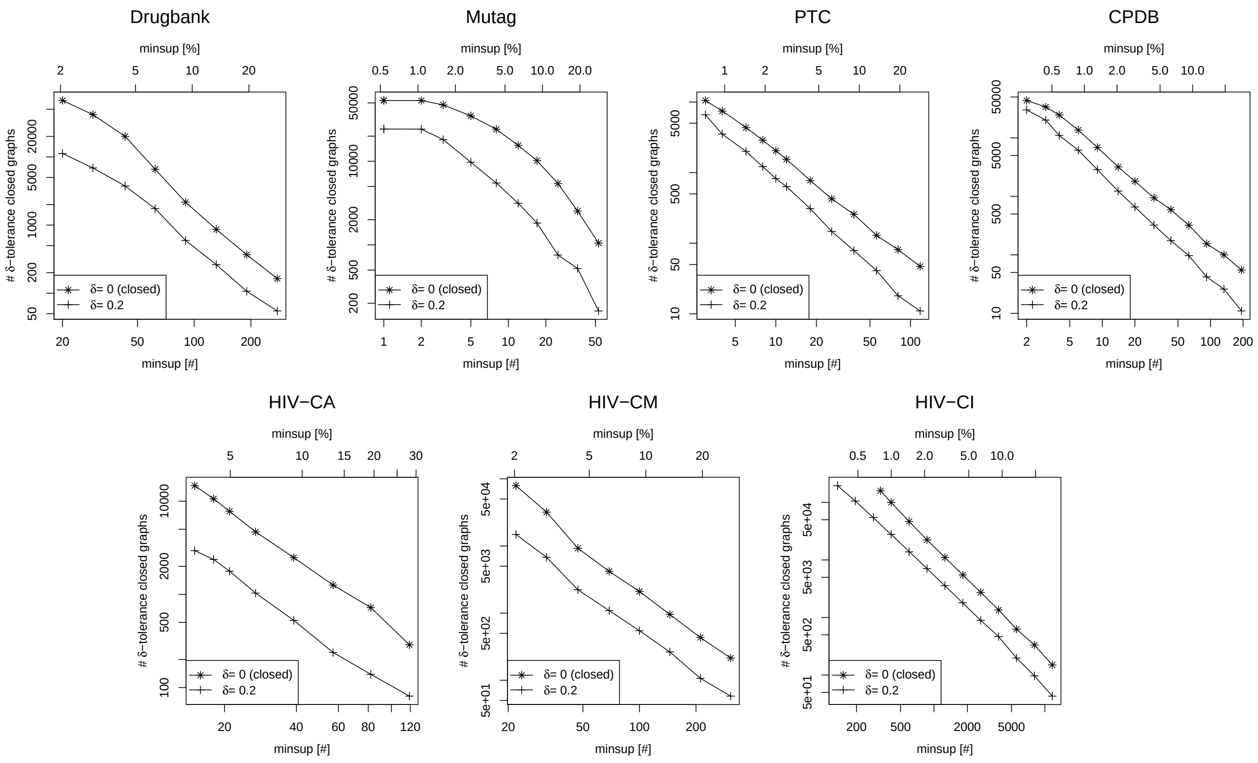


Figure 13: # of outputs (δ-tolerance closed frequent subgraphs) for $\delta = 0.2$ (+), being accompanied with the case of $\delta = 0$ ($*$), corresponding to closed frequent subgraphs when we vary minsup. Results are shown for seven datasets in Table 2.

checked the number of δ-closed frequent subgraphs for each of the seven datasets. Fig. 13 shows the number of outputs for δ=0.2, with the number of closed frequent subgraphs ($\delta = 0.0$), showing approximately a 'linear' correlation on the log-log plot between minsup and the number of outputs (δ-tolerance closed frequent subgraphs of δ=0.2) for all seven datasets. This figure shows that even for smaller values of minsup, the number of outputs in PTC and HIV-CA was around 4,000 or less, being smaller than that of another dataset. From this result, we can see that for these small-scale datasets, the amount of real computation for various pruning became not necessarily small, comparing with other parts, resulting in that the time-efficiency of our method for these datasets was rather limited. This result implies that the time-efficiency of the proposed method can be pronounced more for larger-scale datasets. This point can be confirmed by another point of our results: The size of seven datasets is around 1,000 or less, except HIV-CI which has around 41,000 graphs, and we can see that from Fig. 11, the performance improvement of our method over Naive and CNaive on HIV-CI was more outstanding than that on other datasets. These results indicate that our method is more advantages for larger datasets as well as for smaller values of minsup. This conclusion further imply that our proposed method is useful for real settings more.

6 Concluding Remarks

We have proposed an efficient algorithm for mining δ-tolerance closed frequent subgraphs by which the number of outputs can be reasonably reduced and controlled. Our algorithm reformulates the enumeration of mining frequent subgraphs in the framework of "reverse-search" and further makes the most of available pruning techniques in mining closed graphs and trees. We emphasize that our method is the first approach which uses both two types of pruning techniques, i.e. right- and left-blanket pruning, for mining closed frequent subgraphs, implying that our approach must be the fastest method of mining closed frequent subgraphs. Experimental results confirmed the efficiency and the effectiveness of our approach using a variety of real scientific datasets of chemical compounds.

References

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.

[2] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Appl. Math.*, 65(1-3):21–46, 1996.

[3] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.

[4] C. Borgelt and T. Meinl. Full perfect extension pruning for frequent graph mining. In *MCD*, 2006.

[5] C. Borgelt, T. Meinl, and M. R. Berthold. Advanced pruning strategies to speed up mining closed molecular fragments. In *SMC*, pages 4565–4570, 2004.

[6] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006.

[7] C. Chen, C. X. Lin, X. Yan, and J. Han. On effective presentation of graph patterns: A structural representative approach. In *CIKM*, pages 299–308, 2008.

[8] J. Cheng, Y. Ke, and W. Ng. δ-tolerance closed frequent itemsets. In *ICDM*, pages 139–148, 2006.

[9] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[10] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *TKDE*, 17(2):190–202, 2005.

[11] M. Deshpande, M. Kuramochi, and N. Wale. Frequent substructure-based approaches for classifying chemical compounds. *TKDE*, 17(8):1036–1050, 2005.

[12] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *DMKD*, 15:55–86, 2007.

[13] K. Hashimoto, I. Takigawa, M. Shiga, M. Kanehisa, and H. Mamitsuka. Mining significant tree patterns in carbohydrate sugar chains. *Bioinformatics*, 24(16):i167–i173, 2008. Proceedings of the Seventh European Conference on Computational Biology (ECCB 2008).

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

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

[16] Y. Liu, J. Li, and H. Gao. Summarizing graph patterns. In *ICDE*, pages 903–912, 2008.

[17] G. M. Maggiora and V. Shanmugasundaram. *Molecular Similarity Measures*, volume 275 of *Methods in Molecular Biology*, chapter 1, pages 1–50. 2004.

[18] T. Meinl, M. Wörlein, O. Urzova, I. Fischer, and M. Philippsen. *The ParMol package for frequent subgraph mining*, pages 1–12. EASST, 2007. ISSN:1863-2122.

[19] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comp.*, 1(2):146–160, 1972.

[20] L. T. Thomas, S. R. Valluri, and K. Karlapalem. MARGIN: Maximal frequent subgraph mining. In *ICDM*, pages 1097–1101, 2006.

[21] M. S. Wörlein. Extension and parallelization of a graph-mining-algorithm. Friedrich-Alexander-Universität, 2006. Diploma Thesis.

[22] S. B. Yahia, T. Hamrouni, and E. M. Nguifo. Frequent closed itemset based algorithms: A thorough structural and analytical survey. *ACM SIGKDD Explorations*, 8(1):93–104, 2006.

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

[24] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *KDD*, pages 286–295, 2003.

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