TKG: Efficient Mining of Top-K Frequent Subgraphs

Philippe Fournier-Viger[1](✉), Chao Cheng[2], Jerry Chun-Wei Lin[3], Unil Yun[4], and R. Uday Kiran[5,6]

[1] School of Natural Sciences and Humanities, Harbin Institute of Technology (Shenzhen), Shenzhen, China
philfv@hit.edu.cn

[2] School of Computer Sciences and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, China
tidescheng@gmail.com

[3] Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences (HVL), Bergen, Norway
jerrylin@ieee.org

[4] Department of Computer Engineering, Sejong University, Seoul, Republic of Korea
yunei@sejong.ac.kr

[5] The University of Tokyo, Tokyo, Japan
uday_rage@tkl.iis.u-tokyo.ac.jp

[6] National Institute of Information and Communications Technology, Tokyo, Japan

Abstract. Frequent subgraph mining is a popular data mining task, which consists of finding all subgraphs that appear in at least *minsup* graphs of a graph database. An important limitation of traditional frequent subgraph mining algorithms is that the *minsup* parameter is hard to set. If set too high, few patterns are found and useful information may be missed. But if set too low, runtimes can become very long and a huge number of patterns may be found. Finding an appropriate *minsup* value to find just enough patterns can thus be very time-consuming. This paper addresses this limitation by proposing an efficient algorithm named TKG to find the top-k frequent subgraphs, where the only parameter is k, the number of patterns to be found. The algorithm utilizes a dynamic search procedure to always explore the most promising patterns first. An extensive experimental evaluation shows that TKG has excellent performance and that it provides a valuable alternative to traditional frequent subgraph mining algorithms.

Keywords: Graph mining · Frequent subgraphs · Top-k subgraphs

1 Introduction

In the last decades, many studies have been carried out on designing efficient algorithms to discover interesting patterns in different types of data such as customer transactions [8] and sequences [6]. One of the most popular pattern

© Springer Nature Switzerland AG 2019
S. Madria et al. (Eds.): BDA 2019, LNCS 11932, pp. 209–226, 2019.
https://doi.org/10.1007/978-3-030-37188-3_13

mining task is Frequent Subgraph Mining (FSM) [10,14–16,18,20]. It consists of finding all subgraphs that appear in at least *minsup* graphs of a graph database, where *minsup* is a parameter set by the user. The number of graphs containing a pattern is called its *support*. FSM has several applications such as to analyze collections of chemical molecules to find common sub-molecules [10], and to perform graph indexing [22].

But discovering all frequent subgraphs in a set of graphs is a difficult task. To perform this task efficiently, various algorithms have been proposed using various data structures and search strategies [10]. However, traditional FSM algorithms have an important limitation, which is that it is often difficult for users to select an appropriate value for the *minsup* threshold. On one hand, if the threshold is set too low, few patterns are found, and the user may miss valuable information. On the other hand, if the threshold is set too high, millions of patterns may be found, and algorithms may have very long execution times, or even run out of memory or storage space. Since users typically have limited time and storage space to analyze patterns, they are generally interested in finding enough but not too many patterns. Finding a suitable *minsup* value that will yield just enough patterns is difficult because it depends on dataset characteristics that are generally unknown to the user. Thus, many users will run an FSM algorithm several times with different *minsup* values using a trial-and-error approach until enough patterns are found, which is time-consuming.

To address this issue, Li et al. [13] proposed the TGP algorithm to directly find the k most frequent closed subgraphs in a graph database, where k is set by the user instead of the *minsup* threshold. This approach has the advantage of being intuitive for the user as one can directly specify the number of patterns to be found. However, a major issue is that TGP explicitly generates all patterns to then find the top-k closed patterns. Since the number of patterns can increase exponentially with the size of a graph, this approach is inefficient even for moderately large graph databases. In fact, Li et al. [13] reported that the TGP algorithm could not be applied on the Chemical340 dataset, although it had been commonly used to evaluate prior FSM algorithms [20]. To cope with the fact that top-k subgraph mining is more difficult than traditional FSM, researchers have then developed approximate algorithms. The FS^3 algorithm can find an approximate solution to the top-k frequent subgraph mining problem using sampling. Moreover, two approximate top-k frequent subgraph mining algorithms based on sampling were proposed for mining a restricted type of graphs called induced subgraphs [3,4]. However, a major problem is that approximate algorithms cannot guarantee finding all patterns, and may thus miss important information.

To provide an efficient algorithm for top-k frequent subgraph mining that can guarantee finding all frequent subgraphs, this paper proposes an algorithm named *TKG* (Top-K Graph miner). It starts searching for patterns using an internal *minsup* threshold set to 0 and gradually raises the threshold as patterns are found. To ensure that the threshold can be raised as quickly as possible and efficiently reduce the search space, TKG relies on a search procedure that dynamically selects the next promising patterns to be explored. As it will be shown in the experimental evaluation of this paper, TKG has excellent performance on standard benchmark datasets, including the Chemical340 dataset for

which the TGP algorithm could not run. Moreover, it was observed that the performance of TKG is close to that of the state-of-the-art gSpan algorithm for FSM, even though top-k subgraph mining is a more difficult problem than FSM. Hence, TKG provides a valuable and efficient alternative to traditional FSM algorithms.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the problems of (top-k) frequent subgraph mining. Then, Sect. 4 presents the proposed algorithm, Sect. 5 describes the experimental evaluation, and Sect. 6 draws a conclusion.

2 Related Work

The problem of frequent subgraph mining was introduced by Inokuchi et al. [9]. They proposed an algorithm named AGM that can discover all frequent connected and disconnected sub-graphs. It utilizes a breadth-first search where pairs of subgraphs of a size u are combined to generate candidate subgraphs of size $(u+1)$. A similar breadth-first search is used by the FSG algorithm [11]. A drawback of this approach is that it can generate numerous candidates that are infrequent or do not exist in the database, and thus these algorithms may waste a considerable time evaluating infrequent subgraphs. To address these issues, the gSpan algorithm [20] was proposed. To avoid generating candidates, gSpan adopts a pattern-growth approach, which recursively grows patterns by scanning the graph database. Furthermore, to efficiently detect if a newly found subgraph is isomorphic to an already found subgraph, a novel representation of graphs called *Depth-First-Search code* (DFS code) was introduced. Though several other FSM algorithms have then been proposed [10,16], gSpan remains by far the most popular due to its efficiency and because it can be easily extended to handle other subgraph mining problems and constraints [10]. For example, CloseGraph is a popular extension of gSpan [21] to mine a subset of frequent subgraphs called closed patterns (subgraphs that have no supergraph having the same support).

Although traditional FSM algorithms have many applications, how to set the *minsup* threshold is not intuitive. To address, this issue, the TGP [13] algorithm was designed to find the top-k closed subgraphs. For this problem, two key challenges are how to find top-k patterns and how to determine if a pattern is closed. The solution proposed in TGP is to initially scan the database to calculate the DFS codes of all subgraphs of each input graph, and combine all these DFS code in a huge structure called the *Lexicographical pattern net.* In this structure each subgraph is linked to its immediate super-graphs, which allows to quickly check if a subgraph is closed. Then, TGP starts to search for the top-k closed subgraphs using that structure, while gradually raising an internal *minsup* threshold initially set to 0. Though, this approach guarantees finding the top-k closed subgraphs, it is inefficient in time and memory because the DFS codes of all patterns must be calculated and stored in memory. This structure is huge because the number of subgraphs can increase exponentially with graph

size. As a result, TKG is unable to run on moderately large datasets such as the Chemical Compound benchmark dataset (also known as Chemical340), where the largest graph has 214 edges and 214 vertices. But traditional FSM algorithms relying on DFS codes such as gSpan and CloseGraph can run very efficiently on this dataset.

Then, the FS^3 (Fixed Sized Subgraph Sampler) algorithm was proposed to find an approximate set of top-k frequent subgraphs [17]. This algorithm was designed with the idea of trading result completeness and accuracy for efficiency. To apply FS^3, the user must specify a number of iterations, a fixed size p for subgraphs to be found, and the number of patterns k to discover. To find frequent patterns, FS^3 performs two phase sampling: (1) it first samples a graph from the database, and then (2) samples a p-size subgraph biased toward frequent subgraphs in the whole database using the Markov Chain Monte Carlo method. This process is repeated until the maximum number of iterations is reached, and a priority queue structure is used to maintain a list of the k best (most frequent) sampled subgraphs. An advantage of this approach is that it is very fast as it avoids calculating subgraph isomorphism, one of the costliest operations in FSM that is NP-complete. But an important drawback is that the support of patterns is approximately calculated. As a result, the FS^3 algorithm can not only miss frequent or top-k patterns due to sampling, but it may also return infrequent patterns. Moreover, another serious limitation is that a fixed subgraph size must be set by the user. Setting this parameter is not intuitive, and restricting the search to a fixed size can result in missing several interesting patterns.

Then, another approximate algorithm for mining top-k fixed size frequent subgraphs was proposed, named kFSIM [3]. It adopts a similar sampling approach as FS^3, which also avoids subgraph isomorphism checks but may incorrectly calculate the support of patterns. kFSIM relies on a novel measure called $indFreq$ to accelerate support calculation and improves its accuracy. kFSIM is designed for handling a restricted type of graphs called induced subgraphs [3,4], and was shown to outperform FS^3 in terms of accuracy and runtimes on real datasets. Then, the authors of kFSIM proposed a similar algorithm named kFSIM [3] for finding top-k frequent fixed size induced subgraphs in a stream using sampling and a window [3]. However, it also an approximate algorithm that cannot guarantee result completeness and accuracy.

To address the aforementioned drawbacks of previous algorithms, this paper present an efficient algorithm named TKG that is exact (find all top-k frequent subgraphs) and has runtimes that are close to those of the gSpan algorithm for traditional frequent subgraph mining. As it will be shown in the experimental evaluation section, TKG runs efficiently on the Chemical340 dataset, where TGP could not run. The next section introduces preliminaries and formally defines the problems of FSM and top-k FSM. Then, the next section presents the proposed algorithm.

3 Preliminaries and Problem Definition

Frequent subgraph mining is applied on a database of labeled graphs [9,10,20].

Definition 1 (Labeled graph). *Formally, a* labeled graph *is defined as a tuple* $G = (V, E, L_V, L_E, \phi_V, \phi_E)$ *in which* V*,* E*,* L_V *and* L_E *are the sets of edges, vertices, vertex labels and edge labels, respectively. Furthermore,* ϕ_V *and* ϕ_E *are functions that map vertices and edges to their labels, respectively (*$\phi_V : V \rightarrow L_V$ *and* $\phi_E : E \rightarrow L_E$*).*

Furthermore, it is assumed that graphs are connected (one can follows graph edges to reach a vertex from any other vertex), do not contain self-loops (an edge from a vertex to itself) and multiple edges between pairs of vertices.

Definition 2 (Graph database). *A* graph database $GD = \{G_1, G_2 \ldots G_n\}$ *is defined as a set of* n *labeled graphs.*

For example, Fig. 1 shows a graph database containing three graphs denoted as G_1, G_2 and G_3. The graph G_3 contains four vertices and four edges. The edge labels are $L_E = \{x, y, z, w\}$ and the vertex labels are $L_V = \{A, B\}$. This database will be used as running example.

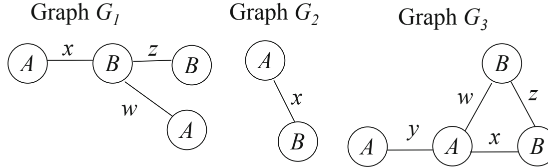

Fig. 1. A graph database containing three graphs

The goal of frequent subgraph mining is to find patterns having a high support (occurring in many graphs). The support is defined based on the concept of graph isomorphism and subgraph isomorphism.

Definition 3 (Graph isomorphism). *Let there be a labeled graph* $G_x = (V_x, E_x, L_{xV}, L_{xE}, \phi_{xV}, \phi_{xE})$ *and another labeled graph* $G_y = (V_y, E_y, L_{yV}, L_{yE}, \phi_{yV}, \phi_{yE})$*. It is said that the graph* G_x *is isomorphic to* G_y *if there is a bijective mapping* $f : V_x \rightarrow V_y$ *meeting two conditions. First, for any vertex* $v \in V_x$*, it follows that* $L_{xV}(v) = L_{yV}(f(v))$*. Second, for any pair* $(u, v) \in E_x$*, it follows that* $(f(u), f(v)) \in E_y$ *and* $L_{xE}(u, v) = L_{yE}(f(u), f(v))$*.*

Intuitively, if G_x is isormorphic to G_y, it means that the two graphs are equivalent because labels from nodes and edges from one graph can be mapped to the other while preserving the same graph structure. To check if a subgraph appears in a graph, the relationship of subgraph isomorphism, and the concept of support are defined as follows.

Definition 4 (Subgraph isomorphism). *Let there be two graphs* $G_x = (V_x, E_x, L_{xV}, L_{xE}, \phi_{xV}, \phi_{xE})$ *and* $G_z = (V_z, E_z, L_{zV}, L_{zE}, \phi_{zV}, \phi_{zE})$*. It is said that* G_x *appears in the graph* G_z*, or equivalently that* G_x *is a subgraph isomorphism of* G_z*, if* G_x *is isomorphic to a subgraph* $G_y \subseteq G_z$*.*

Definition 5. *Let there be a graph database GD. The support (occurrence frequency) of a subgraph G_x in GD is defined as* $sup(G_x) = |\{g | g \in GD \wedge G_x \sqsubseteq g\}|$.

In other words, the support of a subgraph g in a database is the number of graphs that contains g. For example, the graph (A)—x—(B) appears in the graph G_1, G_2 and G_3, and thus has a support of 3. The graph (A)—w—(B) only appears in G_1 and G_3, and hence has a support of 2.

It is to be noted that the support of a subgraph remains the same if a subgraph appears once or multiple times in the same graph. The problem of frequent subgraph mining is defined as follows.

Definition 6 (Frequent Subgraph mining). *Let there be a user-defined threshold $minsup > 0$ and a graph database GD. The problem of frequent subgraph mining consists of finding all subgraphs that have a support no less than minsup.*

For example, Fig. 2 shows the seven frequent subgraphs found in the graph database of Fig. 1 for $minsup = 2$, denoted as $g_1, g_2, \ldots, g_7$.

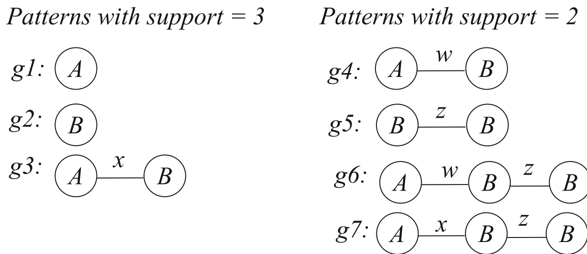

Fig. 2. Frequent subgraphs for $minsup = 2$, i.e. top-k frequent subgraphs for $k = 7$

The problem of mining the top-k frequent subgraphs addressed in this paper is a variation of the problem of frequent subgraph mining where $minsup$ is replaced by the parameter k.

Definition 7 (Top-k Frequent Subgraph mining). *Let there be a user-defined parameter $k \geq 1$ and a graph database GD. The problem of top-k frequent subgraph mining consists of finding a set T of k subgraphs such that their support is greater or equal to that of any other subgraphs not in T.*

For example, Fig. 2 shows the top-k frequent subgraphs found in the graph database of Fig. 1 for $k = 7$. If $k = 3$, only the subgraphs g_1, g_2 and g_3 are found. These subgraphs are the top-3 frequent subgraphs because no other subgraphs have a higher support.

It is important to note that in some cases, more than k patterns could be included in the set T, and thus that there can be several good solutions to top-k frequent subgraph mining problem. This is for example the case if $m > k$

patterns have exactly the same support. Moreover, it is possible that T contains less than k patterns for very small graph databases where the number of possible patterns is less than k.

The problem of top-k frequent subgraph mining is more difficult than the problem of frequent subgraph mining because the optimal minimum support value to obtain the k most frequent patterns is not known beforehand. As a consequence, all patterns having a support greater than zero may have to be considered to find the top-k patterns. Thus, the search space of top-k frequent subgraph mining is always greater or equal to that of frequent subgraph mining when the minimum support threshold is set to the optimal value.

To find the top-k frequent subgraphs efficiently, the next section presents the proposed TKG algorithm.

4 The TKG Algorithm

The designed TKG algorithm performs a search for frequent subgraphs while keeping a list of the current best subgraphs found until now. TKG relies on an internal *minsup* threshold initially set to 1, which is then gradually increased as more patterns are found. Increasing the internal threshold allows to reduce the search space.

To explore the search space of subgraphs, TKG reuses the concept of rightmost path extension and DFS code, introduced in the gSpan algorithm [20]. A reason for using these concepts is that it allows to explore the search space while avoiding generating candidates[1]. Moreover, it allows to avoid using a breadth-first search, which is key to design an efficient top-k algorithm. The reason is that if a top-k pattern has a very large size, a top-k algorithm based on a breadth-first search could be forced to generate all patterns up to that size, which would be inefficient. Using rightmost path extension and DFS codes allows to search in different orders such as using a depth-first search. Moreover, these concepts were shown to be one of the best way for tackling subgraph mining problems [10,21,24]. It is to be noted that while the gSpan algorithm utilizes a depth-first search for frequent subgraph mining, the proposed algorithm utilizes a novel search space traversal approach called *dynamic search* to always explore the most promising patterns first. This allows to guide the search towards frequent subgraphs, raise the internal *minsup* threshold more quickly, and thus reduce a larger part of the search space. As it will be shown in the experimental evaluation, the proposed dynamic search traversal greatly improves the efficiency of top-k frequent subgraph mining compared to using a depth-first search.

Before presenting the details of the proposed algorithm, the next subsection introduces key concepts related to rightmost path extension and DFS codes.

[1] *Here, generating a candidate* means to combine two subgraphs to obtain another subgraph that may or may not exist in the database [20]. This is done by algorithms such as AGM [9] and FSG [11] to explore the search space.

4.1 Rightmost Path Extensions and DFS Codes

A key challenge in frequent subgraph mining is to have a method that allows to systematically enumerate all subgraphs appearing in a graph. A popular solution to this problem is to use the concept of rightmost path extension [20], which allows to consider all edges of a graph using a depth-first search, without considering the same edge twice. In that context, an *extension* means an edge of a graph that can extend a subgraph.

Definition 8 (Rightmost path extension). *A depth-first search can be performed over a graph using a recursive stack. Vertices in the recursive stack are said to form the rightmost path in the graph, and the currently processed vertex is called the rightmost vertex. Rightmost path extension consists of performing two types of extensions: forward extensions and backward extensions. Backward extensions are performed before forward extensions and are used for visiting edges that will form cycles (larger cycles are preferred). Forward extensions are used for visiting edges that lead to new vertices. For forward extensions, extension of the rightmost vertex is considered first, and then extensions of vertices on the rightmost path (which makes it a depth-first search).*

For instance, consider a depth-first search over the graph G_3 of the running example, where nodes are visited according to the order depicted on Fig. 3 (left), where numbers '1', '2', '3' denote the visiting order of vertices. When the depth-first search reaches node '3', vertex 3 is the rightmost vertex, (A)—x—(B) is a backward edge and (A)—y—(A) is a forward edge. Thus, the edge (A)—x—(B) will be considered next to pursue the search, and then (A)—y—(A).

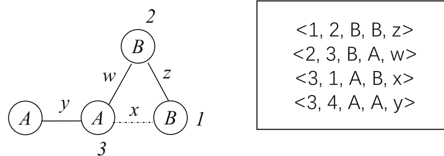

Fig. 3. (left) A rightmost path over the graph G_3 of Fig. 1, where '1', '2' and '3' is the vertex visiting order, $\langle 1, 2, 3 \rangle$ is a rightmost path and vertex 3 is the rightmost vertex. (right) The DFS code obtained after extending that path with the backward edge A—x—B and then the forward edge A—y—A.

For a subgraph g and a graph G_i of a graph database, the gSpan algorithm [20] first finds an isomorphic mapping from g to G_i. Then, it applies the concept of rightmost path extension to finds subgraphs that can extend g with an additional edge, and at the same time calculates its support. Recursively applying the concept of rightmost extension ensures that all subgraphs can be eventually considered. However, two extensions may still yield two subgraphs that are isomorphic (that are equivalent). It is thus important to identify all

such duplicates during the mining process to avoid considering a same subgraph multiple times. To solve this problem, a code called DFS code was proposed to represent each subgraph [20], which allows to identify duplicates.

Definition 9 (Extended edges). *Let there be an edge between two vertices* v_i *and* v_j *and* ϕ *be the labeling function (*ϕ_V *or* ϕ_E*). A tuple* $\langle v_i, v_j, \phi(v_i), \phi(v_j), \phi(v_i, v_j)\rangle$ *representing the edge, its label and the vertex labels is called an extended edge.*

For example, in Fig. 3, $\langle 1, 2, B, B, z\rangle$ and $\langle 2, 3, B, A, w\rangle$ are extended edges. Moreover, the edges (A)—x—(B) and (A)—y—(A) are represented by the extended edges $\langle 3, 1, A, B, x\rangle$ and $\langle 3, 4, A, A, y\rangle$, respectively.

Definition 10 (DFS code). *The DFS code of a graph is a sequence of extended edges, sorted in depth-first search order.*

Continuing the previous example of Fig. 3 (left), if the backward edge (A)—x—(B) and forward edge (A)—y—(A) are used as extension, the DFS code of the resulting subgraph is the sequence of four extended edges, shown in Fig. 3 (right).

From a DFS code, one can recover the corresponding graph in the original visiting order. A graph can have many different DFS codes. To consider a single DFS code for each graph, a total order on extended edges is defined [20].

Definition 11 (Total order of extended edges). *Let* t_1 *and* t_2 *be two extended edges:*

$$t_1 = \langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j)\rangle$$
$$t_2 = \langle v_x, v_y, L(v_x), L(v_y), L(v_x, v_y)\rangle$$

The edge t_1 *is said to be smaller than* t_2 *if and only if (i)* $(v_i, v_j) <_e (v_x, v_y)$ *(ii)* $(v_i, v_j) =_e (v_x, v_y)$ *and* $\langle L(v_i), L(v_j), L(v_i, v_j)\rangle <_l \langle L(v_x), L(v_y), L(v_x, v_y)$*. Relationship* $<_e$ *is consistent with the rule for rightmost path extension, that is, for* $e_{ij} = (v_i, v_j)$ *and* $e_{xy} = (v_x, v_y)$*,* $e_{ij} <_e e_{xy}$ *if and only if* (a) e_{ij} *and* e_{xy} *are both forward edges, then* $j < y$ *or* $j = y$ *and* $i > x$*;* (b) e_{ij} *and* e_{xy} *are both backward edges, then* $i < x$ *or* $i = x$ *and* $i > x$*;* (c) e_{ij} *is a forward edge and* e_{xy} *is a backward edge, then* $j \leq x$*;* (d) e_{ij} *is a backward edge and* e_{xy} *is a forward edge, then* $i < y$*.* $<_l$ *is consistent with the lexicographic order.*

This total order allows to order DFS codes. For example, in the simple graph (A)—x—(B)—z—(B), a DFS code beginning with $\langle 0, 1, A, B, x\rangle$ is smaller than those beginning with $\langle 0, 1, B, A, x\rangle$ or $\langle 0, 1, B, B, z\rangle$.

Definition 12 (Canonical DFS code). *A DFS code is called canonical if and only if it has the least order among all DFS code corresponding to the same graph.*

The property that each graph has only one canonical DFS code allows to efficiently detect duplicate subgraphs. During the search for subgraphs, a graph can be checked for canonicity and if it is non canonical, it can be ignored. This eliminates the need of comparing a subgraph with previously considered subgraphs to determine if it is a duplicate [20].

4.2 The Algorithm

The proposed TKG algorithm takes as input a graph database and a parameter k. It outputs the set T of the top-k frequent subgraphs. The pseudocode is shown in Algorithm 1.

Algorithm 1. The TKG algorithm

```
input : GD: a graph database, k: a user-specified number of patterns
output: the top-k frequent subgraphs
1  Initialize a priority queue Q_K for storing the current top-k frequent
   subgraphs, where subgraphs with smaller support have higher priority.
2  Initialize a priority queue Q_c for storing candidate subgraphs for next
   extension, where subgraphs with higher support have higher priority.
   Initially, contains an empty graph.
3  minsup = 1
4  while Q_c is not empty do
5      g ← pop highest priority subgraph from Q_c
6      ε ← rightMostPathExtensions(g, GD) // Finds edges that can extend g
       and compute their support values.
7      foreach (t, sup(t)) ∈ ε do
8          g' ← g ∪ {t} // Add the edge t to the DFS code of graph g
9          sup(g') ← sup(t)
10         if sup(g') ≥ minsup and isCanonical(g') then
11             // Save pattern g' in list of current top-k patterns
12             Insert g' into Q_K
13             if Q_K.size() ≥ k then
14                 // Raise the internal threshold
15                 if Q_K.size() > k then pop the highest priority (least support)
                   subgraph from Q_K;
16                 minsup = sup(Q_K.peek())
17             end
18             // Save g' as a candidate for future extension instead of doing a
               depth-first search
19             Insert g' into Q_c
20         end
21     end
22 end
23 Return Q_K
```

To dynamically search for top-k frequent subgraphs, the algorithm relies on two priority queues. The first one, Q_K, is used for storing at any time the k most frequent subgraphs found until now (Line 1). In that queue, subgraphs with lower support have higher priority. The second queue, Q_c stores subgraphs that may be extended to find larger subgraphs (Line 2). In that queue, graphs with higher support have higher priority. Initially, this queue contains only one

element that is an empty graph (without edges and vertices). The algorithm utilizes an internal *minsup* threshold, initially set to the lowest value (e.g. 1) (Line 3). While Q_c is not empty (Line 4), the algorithm considers extending the most promising subgraph g (the one that has the largest support) in the queue Q_c of subgraphs to be extended (Line 5). The assumption is that subgraphs having a high support should be extended first because they are more likely to yield subgraphs having a high support, and thus to help increase the internal *minsup* threshold more quickly to reduce the search space. This graph g is popped from Q_c (Line 5). Then, the procedure $rightMostPathExtensions()$ is called with g to find all of its extensions (extended edges) and their supports (Line 6). For each extension t, the algorithm combine the extension with the original subgraph g to form a one edge larger subgraph g' (Line 7–8). If the support of g' is larger than the current *minsup* threshold, and if the newly formed g' is canonical (tested by calling the $isCanonical()$ procedure), g' is inserted into Q_K as one of the current k best frequent subgraphs (Line 9–12). Then, if the size of Q_K is larger than k, the subgraph having the highest priority (lowest support) in Q_K is popped from Q_k (Line 13–15). Moreover, if the size of Q_k is greater or equal to k, the *minsup* threshold is set to the support of the subgraph having the smallest support in Q_K (Line 16). Then, rather than immediately considering extending g', the algorithm stores g' in Q_c as a graph that may be eventually extended (Line 19). Then, the algorithm processes other extensions of g (Line 7–20). Then, the algorithm continues the while loop (Line 4–22) such that the graph having the highest priority in Q_c will next be considered for extensions. We name *dynamic search* this approach of extending subgraphs having the highest support first. Note that using this approach, subgraphs are generated using a different order than the depth-first search used by gSpan. In the experimental evaluation the performance of the two search order will be compared. When the algorithm terminates, the set Q_k contains top-k frequent subgraphs.

In the proposed algorithm, the procedure $rightMostPathExtension(g, GD)$ finds all extensions of a graph g (represented by its DFS code). This is done by finding all isomorphic mappings of that code to each graph G_i in the input database, and then by finding the forward and backward extensions of each mapping. The procedure $isCanonical(g')$ performs canonicality checking by recovering the graph corresponding to a DFS code g', generating the canonical DFS code g'' and comparing g' with g''. If the two codes are same, then g' is canonical. These two procedures are implemented as in the gSpan algorithm [20], and thus details about these procedure are omitted from this paper.

In terms of implementation, the TKG algorithm represents all graphs as DFS codes. In other words, g, g', and subgraphs in Q_K and Q_C are internally stored as DFS codes. And when the algorithm terminates, the DFS codes of the top-k subgraphs can be saved as graphs in an output file. In terms of data structures, priority queues can be implemented using heaps or other structures such as red-black trees. Such structures provide low complexity for inserting, deleting elements, and obtaining the element having the highest priority.

The proposed TKG algorithm is correct and complete since it relies on the concepts of DFS code and rightmost path extension introduced in gSpan to ensure that all patterns can be visited, to detect duplicates, and to calculate their support. Then, to ensure that top-k patterns are found, the algorithm starts from $minsup = 1$ and raises the minimum support threshold when at all least k patterns have been found, using the least support among the current top-k patterns. By doing so, a set of top-k most frequent subgraphs is found.

It is to be noted that a top-k problem may have more than one good solution (as explained in previous section). For example, if there is more than k patterns that have exactly the same support, more than k patterns may be considered as top-k patterns. In that case, TKG will return k of those subgraphs because the user wants k patterns. And this satisfies the problem definition. However, if one wants to keep more than k patterns, it is easy to modify Line 13 to 17 of TKG to keep more than k patterns.

Lastly, note that using the dynamic search instead of a depth-first search does not influence TKG's correctness and completeness since subgraphs are just visited in a different order. However, the dynamic search is useful to improve performance as it can help raising the internal *minsup* threshold more quickly.

4.3 Additional Optimizations

Besides using a dynamic search to explore the search space of frequent subgraphs, two other optimizations are also proposed in the designed TKG algorithm for speeding up the pattern mining process.

The first optimization is called the *skip strategy*. Recall that for a candidate graph g for extension, the procedure $rightMostPathExtensions(g, GD)$ finds all extensions from each graph G_i in the input database. After processing a graph G_j, let $hsup$ be the highest support among the found extensions and rn be the number of remaining graphs. If $hsup + rn < minsup$, it indicates that the subgraph g cannot be used to find any frequent extensions. Therefore, the procedure $rightMostPathExtensions(g, GD)$ can stop processing the remaining graphs of the database and empty the current extension list to avoid further checking. This strategy can decrease runtimes.

The second optimization is to initially scan the database to calculate the support of all single edge graphs and then to use this information to update Q_k, $minsup$ and Q_c, before performing the dynamic search. Doing so decreases the processing time for single edge graphs.

5 Experimental Evaluation

To evaluate the performance of the proposed algorithm, extensive experiments have been done. The testing environment is a workstation running Ubuntu 16.04, equipped with an Intel(R) Xeon(R) CPU E3-1270 3.60 GHz, and 64 GB of RAM. The TKG and gSpan algorithms were implemented in Java. Both algorithm implementations use the same code for loading datasets, outputting patterns,

performing canonical testing and generating DFS codes. In the experiments, runtime and memory usage were measured using the standard Java API. Four standard benchmark datasets have been used, which have varied characteristics. They are described in Table 1 in terms of number of input graphs, average number of nodes per graph, average number of edges per graph, total number of vertices and total number of edges.

The four datasets are all bio- or chemo-informatics datasets. The protein dataset [1] contains 1113 graphs, each representing information about the structures of proteins. The nci1 dataset [19] contains 4110 graphs representing chemical compounds related to cancer research. The enzymes datasets [1] contains 600 graphs representing enzymes from a database called BRENDA. Lastly, the Chemical340 dataset contains 340 graphs [20], where each vertex represents an atom, and its label provides information about the atom element and type. Furthermore, each edge represents the bond between two atoms and the edge label indicates bond types. In that dataset, the largest graph contains 214 vertices and 214 edges.

Table 1. Dataset characteristics

Dataset	$\|GD\|$	Avg. nodes	Avg. edges	$\|L_V\|$	$\|L_E\|$
Protein	1113	39.05	72.82	3	1
nci1	4110	29.87	32.3	37	3
Enzymes	600	32.63	62.13	3	1
Chemical340	340	27.02	27.40	66	4

5.1 Influence of k on the Performance of TKG

In a first experiment, the parameter k was varied to evaluate its influence on the performance of TKG in terms of runtime and memory usage. Three versions of TKG are compared: (1) TKG (with all optimizations), (2) TKG without dynamic search (using the depth-first search of gSpan), and (3) TKG without the skip strategy. Results for runtime are shown in Fig. 4 for the four datasets. Results for memory usage are shown in Fig. 5.

It is first observed that as k is increased runtime and memory usage increase. This is reasonable since as k is increased, more patterns must be found. As a result, TKG may need to consider more patterns to fill Q_k and be able to raise the internal *minsup* threshold to reduce the search space.

It is also observed that the dynamic search strategy generally greatly decreases runtime, and considerably decrease memory usage on the protein and enzymes datasets. For example, on the protein dataset, when $k = 200$, TKG with dynamic search is up to 100 times faster than TKG using a depth-first search and consumes up to 8 times less memory. On the enzymes dataset, when

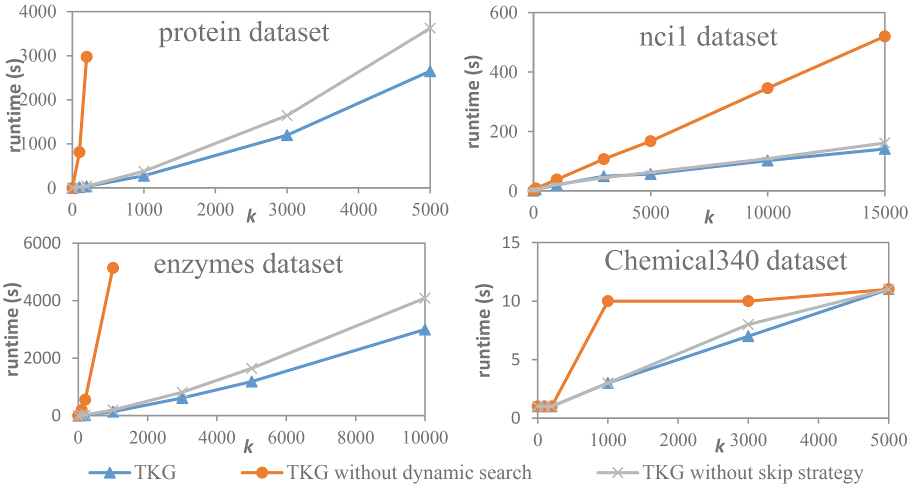

Fig. 4. Influence of k on runtime for different datasets.

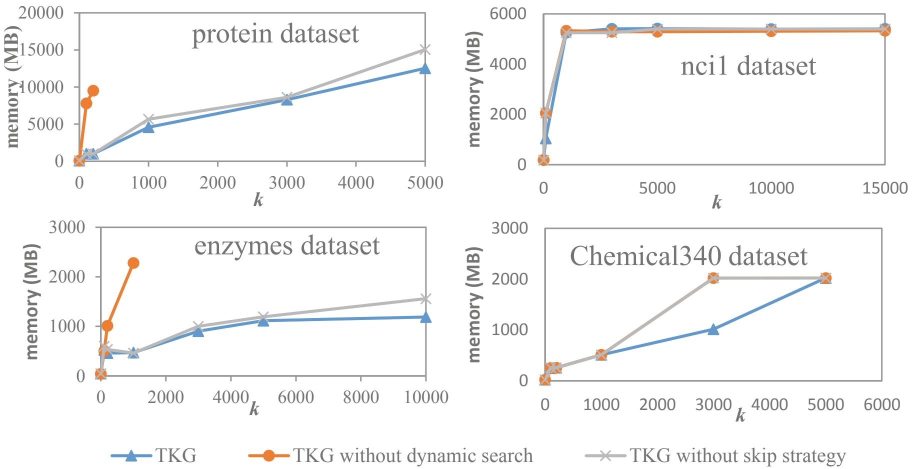

Fig. 5. Influence of k on peak memory usage for different datasets.

$k = 1000$, TKG with dynamic search is up to 40 times faster than TKG using a depth-first search and consumes up to 4 times less memory. On the ncil and Chemical340 datasets, not much memory reduction is achieved because although the dynamic search can help raise the internal *minsup* threshold more quickly and thus reduce the number of subgraphs considered, the queue Qc must be maintained in memory, which offsets those benefits.

It is also found that using the skip strategy is useful to reduce runtime, and can slightly reduce memory usage on some datasets. For example, on the protein dataset, when $k = 1000$, it can reduce runtime by up to 50% and memory

by up to 25%. The skip counting strategy reduces runtime because when the strategy is applicable, less comparisons are made and less extensions are stored in memory.

It is also interesting to observe that the proposed TKG algorithm has a small runtime (less than 15 s) for k values of up to 5000 on the Chemical340 dataset, while the TGP algorithm for top-k closed subgraph mining was reported to be unable to terminate on that dataset [13]. The main reason, discussed in the related work section, is that TGP must calculate and store the DFS codes of all patterns in memory to then find the top-k patterns, which is inefficient for moderately large graphs. The TKG algorithm does not use this approach. It instead explores the search space using a dynamic search to guide the search toward the most promising patterns, and TKG reduces the search space using the internal *minsup* threshold to avoid generating the DFS codes of all patterns. Moreover, TKG does not need to keep all patterns in memory.

5.2 Performance Comparison with gSpan Set with an Optimal *minsup* Threshold

In another experiment, the performance of TKG was compared with that of gSpan. The goal of this experiment is to assess if top-k frequent subgraph mining using TKG can have similar performance to that of the traditional task of frequent subgraph mining. This question is interesting because top-k frequent subgraph mining is a more difficult problem than frequent subgraph mining. The reason is that in top-k frequent subgraph mining, the search for patterns must start from $minsup = 1$, while in frequent subgraph mining the *minsup* threshold is fixed beforehand by the user. The comparison of TKG with gSpan is also interesting because TKG reuses some techniques from gSpan.

TKG was run with k values from 1 to 5000 on each dataset. Then, the gSpan algorithm was run with the optimal *minsup* value to obtain the same number of patterns. The runtime and peak memory usage was measured. Tables 2 and 3 show results for the protein and enzymes datasets, respectively. Results for the other two datasets are not shown but similar trends were observed.

From these results, it is found that TKG and gSpan have very similar runtimes. This is a good result since the problem of top-k subgraph mining is more difficult than the traditional problem of frequent subgraph mining.

In terms of memory, TKG generally consumes more memory than gSpan (up to twice more). This is reasonable since TKG needs to keep a priority queue Q_k to store the current top-k patterns, and another priority queue Q_c to store patterns to be extended by the dynamic search.

It is important to note that this experiment was done by setting an optimal *minsup* value for gSpan to obtain the same number of patterns as TKG. But in real-life, the user typically don't know how to set the *minsup* threshold. Setting k is more intuitive than setting *minsup* because the former represents the number of patterns that the user wants to analyze. For example, consider that a user wants to find 200 to 1000 patterns in the protein dataset. According to Table 2, the range of *minsup* values that would satisfy the user is $[0.5247, 0.6289]$,

which has a length of $0.6289 - 0.5247 = 0.1042$. Thus, the user has about 10.4% chance of setting correctly the *minsup* threshold of gSpan. Now consider the same scenario for the enzymes datasets. In that case, the range of suitable *minsup* values is $[0.6650, 0.7817]$ according to Table 3. Hence, the user has about 11.7% chance of correctly setting the *minsup* threshold of gSpan. If the user sets the *minsup* threshold too low, gSpan may find too many patterns and may become very slow, while if the threshold is set too high, the user may need to run the algorithm again until a suitable value is found, which is time-consuming. To avoid such trial-and-error approach to find a suitable *minsup* value, this paper has proposed the TKG algorithm, which let the user directly specify the number of patterns to be found. Because the runtime of TKG is close to that of gSpan, TKG can be considered a valuable alternative to gSpan.

Table 2. Comparison of TKG and gSpan with optimal *minsup* threshold on the protein dataset

k	*minsup*	TKG runtime (s)	gSpan runtime (s)	TKG memory (MB)	gSpan memory (MB)
1	0.9227	1	1	85	85
100	0.6720	14	13	1020	507
200	0.6289	31	31	1019	976
1000	0.5247	321	275	4583	3503
3000	0.4618	1205	1198	4583	3503
5000	0.4367	2673	2650	8310	6182

Table 3. Comparison of TKG and gSpan with optimal *minsup* threshold on the enzymes dataset

k	*minsup*	TKG runtime (s)	gSpan runtime (s)	TKG memory (MB)	gSpan memory (MB)
1	0.9767	1	1	46	46
100	0.8067	12	8	527	276
200	0.7817	19	15	462	252
1000	0.6650	151	134	469	296
3000	0.600	625	612	1016	902
5000	0.5700	1280	1249	1113	1060

6 Conclusion

This paper has presented a novel algorithm named TKG to find the top-k frequent subgraphs in a graph database. The user only needs to set a parameter k,

which controls the number of patterns to be found. To quickly raise the internal *minsup* threshold, the algorithm utilizes a dynamic search procedure that explores the most promising patterns first. Moreover, a skip strategy has been integrated in the algorithm to improve its performance. An extensive experimental evaluation has shown that TKG has excellent performance. In particular, the dynamic search and optimizations can decrease the runtime of TKG by up to about 100 times and its memory by up to 8 times. It was also found that TKG has similar runtimes to gSpan and thus that it provides a valuable alternative to traditional frequent subgraph mining algorithms.

The source code of the TKG and gSpan algorithms, as well as the datasets can be downloaded from http://www.philippe-fournier-viger.com/spmf/tkgtkg/, and will also be integrated into the next release of the open-source SPMF data mining software [5].

For future work, designing efficient algorithms for other graph pattern mining tasks will be considered such as for discovering significant trend sequences in dynamic attributed graphs [2], subgraphs in graphs [12], high utility patterns [7] and rare subgraphs [23].

Acknowledgements. The work presented in this paper has been partly funded by the National Science Foundation of China.

References

1. Borgwardt, K.M., Ong, C.S., Schönauer, S., Vishwanathan, S.V.N., Smola, A.J., Kriegel, H.P.: Protein function prediction via graph kernels. Bioinformatics **21**(Suppl 1), 47–56 (2005)
2. Cheng, Z., Flouvat, F., Selmaoui-Folcher, N.: Mining recurrent patterns in a dynamic attributed graph. In: Kim, J., Shim, K., Cao, L., Lee, J.-G., Lin, X., Moon, Y.-S. (eds.) PAKDD 2017. LNCS (LNAI), vol. 10235, pp. 631–643. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57529-2_49
3. Duong, V.T.T., Khan, K.U., Jeong, B.S., Lee, Y.K.: Top-k frequent induced subgraph mining using sampling. In: Proceedings 6th International Conference on Emerging Databases: Technologies, Applications, and Theory (2016)
4. Duong, V.T.T., Khan, K.U., Lee, Y.K.: Top-k frequent induced subgraph mining on a sliding window using sampling. In: Proceedings 11th International Conference on Ubiquitous Information Management and Communication (2017)
5. Fournier-Viger, P., et al.: The SPMF open-source data mining library version 2. In: Berendt, B., et al. (eds.) ECML PKDD 2016. LNCS (LNAI), vol. 9853, pp. 36–40. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_8
6. Fournier-Viger, P., Lin, J.C.W., Kiran, U.R., Koh, Y.S.: A survey of sequential pattern mining. Data Sci. Pattern Recogn. **1**(1), 54–77 (2017)
7. Fournier-Viger, P., Chun-Wei Lin, J., Truong-Chi, T., Nkambou, R.: A survey of high utility itemset mining. In: Fournier-Viger, P., Lin, J.C.-W., Nkambou, R., Vo, B., Tseng, V.S. (eds.) High-Utility Pattern Mining. SBD, vol. 51, pp. 1–45. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-04921-8_1
8. Fournier-Viger, P., Lin, J.C.W., Vo, B., Chi, T.T., Zhang, J., Le, B.: A survey of itemset mining. WIREs Data Min. Knowl. Discov. (2017)

9. Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data. In: Zighed, D.A., Komorowski, J., Żytkow, J. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 13–23. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45372-5_2
10. Jiang, C., Coenen, F., Zito, M.: A survey of frequent subgraph mining algorithms. Knowl. Eng. Rev. **28**, 75–105 (2013)
11. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: Proceedings 1st IEEE International Conference on Data Mining (2001)
12. Lee, G., Yun, U., Kim, D.: A weight-based approach: frequent graph pattern mining with length-decreasing support constraints using weighted smallest valid extension. Adv. Sci. Lett. **22**(9), 2480–2484 (2016)
13. Li, Y., Lin, Q., Li, R., Duan, D.: TGP: mining top-k frequent closed graph pattern without minimum support. In: Proceedings 6th International Conference on Advanced Data Mining and Applications (2010)
14. Mrzic, A., et al.: Grasping frequent subgraph mining for bioinformatics applications. In: BioData Mining (2018)
15. Nguyen, D., Luo, W., Nguyen, T.D., Venkatesh, S., Phung, D.Q.: Learning graph representation via frequent subgraphs. In: Proceedings 2018 SIAM International Conference on Data Mining, pp. 306–314 (2018)
16. Nijssen, S., Kok, J.N.: The gaston tool for frequent subgraph mining. Electron. Notes Theor. Comput. Sci. **127**, 77–87 (2005)
17. Saha, T.K., Hasan, M.A.: FS3: a sampling based method for top-k frequent subgraph mining. In: Proceedings 2014 IEEE International Conference on Big Data, pp. 72–79 (2014)
18. Sankar, A., Ranu, S., Raman, K.: Predicting novel metabolic pathways through subgraph mining. Bioinformatics **33**(24), 3955–3963 (2017)
19. Wale, N., Watson, I.A., Karypis, G.: Comparison of descriptor spaces for chemical compound retrieval and classification. In: Proceedings 6th International Conference on Data Mining, pp. 678–689 (2006)
20. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: Proceedings 2nd IEEE International Conference on Data Mining (2002)
21. Yan, X., Han, J.: CloseGraph: mining closed frequent graph patterns. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2003)
22. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: Proceedings of the 2004 SIGMOD Conference (2004)
23. Yun, U., Lee, G., Kim, C.H.: The smallest valid extension-based efficient, rare graph pattern mining, considering length-decreasing support constraints and symmetry characteristics of graphs. Symmetry **8**(5), 32 (2016)
24. Zhu, F., Yan, X., Han, J., Yu, P.S.: gPrune: a constraint pushing framework for graph pattern mining. In: Proceedings of the 11th Pacific-Asia Conference on Knowledge Discovery and Data Mining (2007)