Mining Top-k Correlated Subgraphs in a Large Graph

ACM SIGMOD Submission ID XXX

ABSTRACT

Mining of correlated patterns, that represent an important class of regularities, has become increasingly important in data management and analytics. Surprisingly, the problem of correlated subgraphs mining from a single, large graph has received little attention. We investigate a novel and critical graph mining problem, called correlated subgraphs mining, which is defined as a pair of subgraph patterns that frequently co-occur in proximity within a single graph.

Correlated subgraphs are different from frequent subgraphs due to the flexibility in which the constituent subgraph instances are connected, thus the existing frequent subgraphs mining algorithms cannot be directly applied. Furthermore, correlation computation between two subgraph patterns require enumerating and finding distances between every pair of subgraph instances of both these patterns — which are both memory intensive and computationally demanding.

To this end, we design a novel, single-step, best-first exploration algorithm to detect correlated subgraph patterns: To further improve the efficiency, we develop a top-k pruning strategy, while to reduce the memory footprints, we develop a compressed data structure, called $\mathcal{R}$eplica that stores all instances of a subgraph pattern. Our empirical results show that the proposed algorithm not only finds interesting correlations, but also achieves good performance for finding correlated subgraphs in large graphs.

1 INTRODUCTION

2 PRELIMINARIES

2.1 Background

An attributed graph $G = (V, E, L)$ has a set of nodes V, a set of edges $E \subseteq V \times V$, and a label set $\mathbb{L}$ such that every node $v \in V$ is associated with a label, i.e., $L(v) \in \mathbb{L}$. In this work, we focus on bidirectional, node-labeled, and un-weighted graphs.

SIGMOD '20, June 14–19, 2020, Portland, Oregon, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

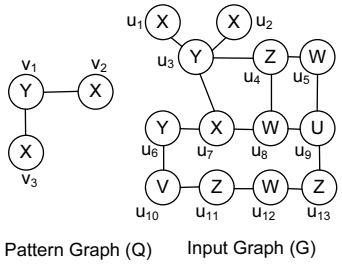


Figure 1: Subgraph isomorphism: $M(v_1) = u_3$, $M(v_2) = u_1$, $M(v_3) = u_2$. Two other subgraph- isomorphic mappings could be as follows: $M_1(v_1) = u_3$, $M_1(v_2) = u_1, M_1(V_3) = u_7$; and $M_2(v_1) = u_3, M_2(v_2) = u_2, M_2(v_3) = u_7$.

However, the proposed models and algorithms can also be applied to directed and edge-labeled graphs.

Subgraph Isomorphism. Given an input graph $G = (V, E, L)$, a graph pattern $Q = (V_Q, E_Q, L_Q)$, a subgraph isomorphism is an *injective function* $M : V_Q \rightarrow V$ s. t. (1) $\forall v \in V_Q, L_Q(v) = L(M(v))$, and (2) $\forall (v_1, v_2) \in E_Q, (M(v_1), M(v_2)) \in E$.

Subgraph isomorphism is depicted in Figure 1. M is called a subgraph-isomorphic *mapping*. The nodes $\{M(v) : v \in V_Q\}$ and the corresponding edges $\{(M(v_1), M(v_2)) : (v_1, v_2) \in E_Q\}$ form a subgraph-isomorphic *instance* of Q in G. There can be many subgraph-isomorphic mappings and instances of Q, e.g., in Figure 1 another mapping M_1 could be as follows: $M_1(v_1) = u_3, M_1(v_2) = u_2, M_1(v_3) = u_7$. Clearly, two different instances of the same pattern may overlap, as in our current scenario: the two instances, defined by mappings M and M_1, overlap at nodes u_2 and u_3, and also on the edge (u_2, u_3).

Support. To find frequent subgraphs from a single, large graph, existing literature proposed several definitions of subgraph support, denoted as σ, e.g., maximum independent sets (MIS) [**?**], minimum image-based (MNI) [**?**], and harmful overlap (HO) [**?**]. All these metrics are *downward-closure*: The support of a supergraph $Q_1 \geq Q$ is higher than that of its subgraph Q, i.e., $\sigma(Q_1) > \sigma(Q)$. However, these metrics differ in the amount of overlap that they allow between subgraph-isomorphic instances, and in the complexity of their computation.

In this work, we adopt MNI [**?**] due to the following reasons. First, the MNI support can be efficiently computed; whereas the computation of MIS and HO are NP-complete [**? ?**]. Second, MNI provides a superset of the results of the two other metrics; thus the MIS or HO-based results can be identified via an expensive post-processing step, which prunes out the unqualified subgraphs [**?**]. Next, we formally define the MNI support.

Minimum Image-based (MNI) Support. Bringmann and Nijssen [**?**] developed the minimum image-based support. It is based on the number of unique nodes in G that a node of the pattern Q is mapped to. Formally,

$$\sigma(Q) = \min_{v \in V_Q} |\{M(v) : M \text{ is a subgraph-isomorphic mapping}\}| \quad (1)$$

In Figure 1, the MNI support of Q is 1, which is due to node v_1 having label Y, it is mapped to only one node in G, i.e., u_3 for all three mappings. The nodes in the set $\{M(v)\}$ for different mappings M are called the *images* of v.

Frequent Subgraphs. Given the input graph G, a user-defined minimum-support threshold Min-Sup, and a definition of support σ, the frequent subgraphs mining problem identifies all subgraphs Q of G, such that $\sigma(Q) \geq$ Min-Sup.

2.2 Problem Formulation

Informally speaking, our objective is to identify those pairs of subgraph patterns $\langle Q_1, Q_2 \rangle$ such that they occur closely for a sufficiently large number of times in the input graph G. We formalize this notion of correlation by incorporating the following constraints: (1) The correlation between two subgraph patterns must be symmetric, and (2) it shall be consistent with respect to the notion of MNI support.

To be consistent with the MNI support, we group subgraph instances as follows.

Definition 1 (Instance Grouping). *Given the input graph G, a graph pattern Q, and its instances in G denoted as $\mathbb{I} = \{I_1, I_2, \ldots, I_s\}$, let us define by v^* the node in Q which has the minimum number of images. We denote by $M(v^*) = \{M_1(v^*), M_2(v^*), \ldots, M_{\sigma(Q)}(v^*)\}$ the images of v^*. Notice that $\sigma(Q)$ is the MNI support of Q, $\sigma(Q) \leq s$, and M_j is a mapping of Q, for all $1 \leq j \leq \sigma(Q)$. Next, we form a grouping of instances, denoted as $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q)}\}$, where $I'_j = \{I : M_j(v^*) \in I, I \in \mathbb{I}\}$. Intuitively, I'_j is the group of instances containing the image node $M_j(v^*)$.*

Example 1. *For input graph G and graph pattern Q_1 in Figure 2, the instances are given by $\mathbb{I} = \{u_1u_3, u_2u_3, u_7u_3, u_7u_6\}$. However, its MNI support is two, since node v_2 has only two corresponding images: u_3 and u_6. Thus, we group the instances according to the presence of u_3 and u_6 as follows: $\mathbb{I}' = \{u_1u_2u_7u_3, u_7u_6\}$.*

Note that the grouping is not a partition of instances. It is possible for an instance to belong to multiple groups, especially when the pattern has multiple nodes with the same label. However, for a pattern Q, we ensure that the number of instance-groups would be $\sigma(Q)$.

Given two subgraph patterns Q_1 and Q_2, we compute their instance-groups: $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q_1)}\}$ and $\mathbb{J}' = \{J'_1, J'_2, \ldots, J'_{\sigma(Q_2)}\}$, respectively. Without loss of generality, let us assume that $\sigma(Q_1) \leq \sigma(Q_2)$. Next, we count, out of all $\sigma(Q_1)$ instance-groups of Q_1, how many of them are "close" to at least one instance-group of Q_2. We report this count as the *correlation* between Q_1 and Q_2 in G. Finally, we define that two instance-groups $I' \in \mathbb{I}'$ and $J' \in \mathbb{J}'$ are close if there exist at least two nodes u in I' and v in J', such that their distance $d(u, v) \leq h$, that is, u and v are no more than h-hops away in the input graph G. Clearly, h is a user-defined *distance-threshold* parameter that can be varied to support different amount of closeness between two co-occurrences of Q_1 and Q_2.

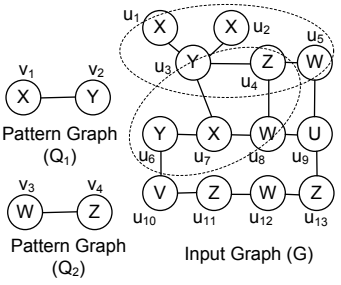


Figure 2: Correlation between subgraphs Q_1 and Q_2 in G

Definition 2 (Correlation). *Given two subgraphs Q_1 and Q_2 in the input graph G, their instance-groups $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q_1)}\}$ and $\mathbb{J}' = \{J'_1, J'_2, \ldots, J'_{\sigma(Q_2)}\}$, respectively, and a user-defined distance-threshold $h \geq 0$, let us assume that $\sigma(Q_1) \leq \sigma(Q_2)$. We define the correlation $\tau(Q_1, Q_2, h)$ as:*

$$\tau(Q_1, Q_2, h) = |\{I' \in \mathbb{I}' : \exists J' \in \mathbb{J}', \exists u \in I', \exists v \in J', d(u, v) \leq h\}| \quad (2)$$

The correlation, for the case $\sigma(Q_2) < \sigma(Q_1)$, can be defined analogously. We note that the correlation between two subgraphs Q_1 and Q_2 is *symmetric*, that is, $\tau(Q_1, Q_2, h) = \tau(Q_2, Q_1, h)$.

Example 2. *Let us consider two subgraph patterns Q_1 and Q_2 in the input graph G (Figure 2), and the distance-threshold $h = 1$. The instance-groups of Q_1 are given by: $\mathbb{I}' = \{u_1u_2u_7u_3, u_7u_6\}$, where the groupings are performed based on the images of node v_2 in Q_1. Similarly, the instance-groups of Q_2 are given by: $\mathbb{J}' = \{u_5u_4, u_8u_4, u_{11}u_{12}u_{13}\}$, here the groupings are performed based on the images of node v_3 in Q_2. We have, $\sigma(Q_1) = 2 < \sigma(Q_2) = 3$. Thus, we count, out of two instance-groups of Q_1, how many of them are within $h = 1$-hop of at least one instance-group of Q_2. This gives us the correlation $\tau(Q_1, Q_2, h = 1) = 2$.*

We are now ready to define our problem formally.

Problem 1. Top-k **Correlated Subgraphs.** *Given the input graph G, a user-defined distance-threshold $h \geq 0$, a*

minimum support threshold Σ, find the top-k *pairs of subgraph patterns $\langle Q_1, Q_2\rangle$ of G, having the maximum correlations $\tau(Q_1, Q_2, h)$, and for each subgraph pattern $\sigma(Q_1) \geq \Sigma$, $\sigma(Q_2) \geq \Sigma$.*

In the aforementioned problem, if Q_1 is a subgraph of Q_2, or vice versa, the correlation between them is not interesting. Thus, in our algorithms, we only identify those pairs which are not related by subgraph and supergraph relationships.

2.3 Theoretical Characterization

The correlation metric satisfies several interesting properties.

LEMMA 1. *The correlation metric $\tau(Q_1, Q_2, h)$ is not downward-closure.*

LEMMA 2. *The correlation metric $\tau(Q_1, Q_2, h)$ is not upward-closure.*

LEMMA 3. *The following inequality holds: $\tau(Q_1, Q_2, h) \leq \min\{\sigma(Q_1), \sigma(Q_2)\}$.*

Lemma 3 directly follows from the definition of correlation (Definition 2), which is computed from the instance-groups of that subgraph pattern having the smaller support.

3 EXACT ALGORITHM

3.1 Overview

OBSERVATION 1. *If two subgraphs have higher support values individually, it is very likely that the pair will also have a higher correlation.*

OBSERVATION 2. *For highly (e.g., top-k) correlated subgraphs mining, generally a breadth-first or a best-first exploration of the search space is more efficient compared to a depth-first traversal of the search space.*

Setting k to infinity would enable us to mine all pairs of correlated subgraph patterns. However, on the contrary, it is hard to control the value of Min-sup to get the result of a particular k of Top-k correlated subgraphs. That is to say, the Min-Sup problem can be transfered from Top-k problem. As a result, we concentrate on Top-k problem in the following sections.

Pattern Search Tree. All operations including correlation computation and subgraph pattern extension are performed on patterns following an order on a *pattern search tree*. We denote this tree as T. Each node $Q \in T$ represents a subgraph pattern. We denote the set of current *leaf* nodes in T as $Leaf(T)$. Each element $Q_l \in Leaf(T)$ is a pattern that has not yet been *operated* for correlation computation. A node $Q_l \in Leaf(T)$, once operated, is inserted into the *operated* set.

The correlated subgraph mining algorithm is an iterative procedure, essentially consisting of the following steps:

1) select a node Q_i from $Leaf(T)$, i.e. $Q_i \in Leaf(T)$, and pop out Q_i from $Leaf(T)$, i.e. $Leaf(T) = Leaf(T) \setminus Q_i$.

2) calculate $\tau(Q_i, Q_j, h)$, for all $Q_j \in operated$

3) try to extend Q_i to $Ex(Q_i)$, where $Ex(Q_i)$ is the set of all the possible subgraphs extended from Q_i.

4) Check the frequency of $Q_i', Q_i' \in Ex(Q_i)$ based on MNI support, $\sigma(Q')$, if $\sigma(Q_i') \geq$ Min-Sup. we put Q_i' into $Leaf(T)$.

THEOREM 1. *The order of subgraph generation and correlation calculation will not miss any correlated pairs between any of two frequent subgraphs.*

3.1.1 Best-first Exploration. Observation 1 suggests that faster convergence of the algorithm could be expected if a subgraph pattern having a higher support is *operated* preceding every other pattern in $Leaf(T)$. As a result, we use a naive rule to determine the priority of the leaf nodes in $Leaf(T)$: Q_1 has a higher priority than Q_2 $\forall Q_1, Q_2 \in Leaf(T)$ if and only if:

$$\sigma(Q_1) > \sigma(Q_2)$$

Utilizing the estimating rule above, if the current leaf node Q_k in the search tree has a highest support among all the other elements in the set of the leaf nodes, i.e. $\sigma(Q_k) = \max\{\sigma(Q_i)|Q_i \in Leaf(T)\}$. Then, Q_k has the priorty to be calculated and extended.

3.1.2 Termination Criteria. Obviously, our purpose is to find k pairs of correlated subgraphs and guarantee that the other pair of subgraphs could not have a higher correlation τ. Retrospect the properties mentioned in Section **??**, assume Q is a frequent subgraph of the data graph, Q_k is an arbitary frequent subgraph of the data graph. We denote $Sup(Q)$ as the set of all the possible supergraphs of Q and $Q' \in Sup(Q)$. Then, the following condition always holds:

$$\tau(Q', Q_k, h) \leq \sigma(Q')$$

$$\sigma(Q') \leq \sigma(Q)$$

It is easy to get following upperbound of $\tau(Q', Q_k, h)$ by combining the two conditions above:

$$\tau(Q', Q_k, h) \leq \sigma(Q)$$

Consider this upperbound, if $\sigma(Q)$ does not reach the minimum number of being a candidate of Top-k set, i.e. $\sigma(Q) < \tau(Q_i, Q_j, h)$ for all the i, j in the current Top-k set, then all the correlations containing Q, $\tau(Q, Q_k, h)$, as well as all the correlations containing Q', $\tau(Q', Q_k, h)$, are impossible to be the elements in Top-k set.

On the other hand, we have to consider another possible circumstance, where there are no k correlated pairs in this graph at all. In that case, the number of elements in Top-k set has not reached k but all the support value of the leaf nodes are already below the minimum support. As a consequence, this condition is also a signal telling us to stop the search.

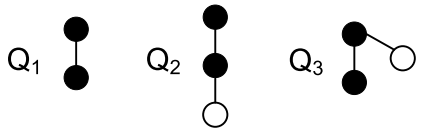


Figure 3: For 3 subgraphs Q_1, Q_2, Q_3, with $\sigma(Q_1) = 5$, $\sigma(Q_2) = 2$, $\sigma(Q_3) = 3$. The current Top-k set is $\{4, 5, 6\}$, the input parameters are $k = 3$, Min-sup= 3

Formally, we fix the condition of not adding Q to $Leaf(T)$, which is, suppose $|Topk|$ is the number of elements in Top-k set, min_sup is the minimum support of the correlation, we stop our search if either both of the following conditions holds.

$$\sigma(Q) \leq min\{t|t \in Top\_k\}$$

$$|Top\_k| = k$$

or following condition holds.

$$\sigma(Q) \leq min\_sup$$

In this case, all the $Q, Q \in Leaf(T)$ is possible to have the correlation we want. Obviously, on the other hand, if $Leaf(T) = \emptyset$, we report all the Top-k correlated subgraphs and cease our search.

EXAMPLE 3. *In Figure 3, the minimum element in* Top-k *set is* 4*, Q_1 is the only leaf node at the beginning, i.e. $Leaf(T) = \{Q_1\}$ and $\sigma(Q_1) > 4$ so that Q_1 can be extended to Q_2, Q_3. After extension, $Q_1 \notin Leaf(T)$ and $Leaf(T) = \{Q_2, Q_3\}$. $Q_2 <$* Min-sup*, and $Q_3 =$* Min-sup *but $Q_3 < 4$, and there are already* 3 *elements in* Top-k *set, i.e. $|Top\_k| = 3$. Thus, all the correlation of Q_2, Q_3 would not satisfies the condition we want and we cease the search.*

3.2 Replica-based Graph Instance Storage

3.2.1 Replica Graph Data Structure. Considering the large amount of overlaps of the instances in dense graph, we use a novel structure to store a subgraph pattern, which not only get rid of the expensive cost of tackling dense graph, but also be the robust foundation of carrying out an efficient correlation calculation based on instance grouping.

Our search tree storage unit is extremely direct and naive. Instead of considering the instances of a pattern, we just create a reproduce of the occurrences of the vertices and the edges of the pattern, called **replica**. We record all the vertex identifications and the edge connections in the replica, we do not record the edge labels.

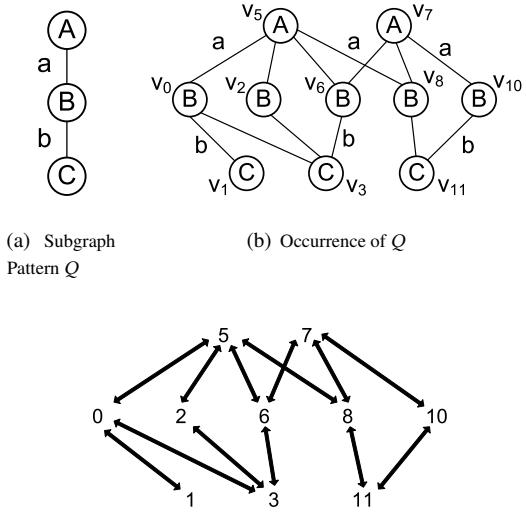
After Q has been *operated*, we remove $replica(Q)$ from memory.

• **Replica Unit.** The illustration of replica is as Figure 5. The details of replica is as follows.

1) For each $Q \in T$, there is a corresponding replica $R(Q)$.

2) For each $u_i \in Q$, there is a hash-map, $R_i(Q)$, recording the vertex identifications, and $R(Q) = (R_1(Q), R_2(Q), ..., R_{|V_Q|}(Q))$.

3) For each element $v \in R_i(Q)$, there is a record of all the other vertices which v is connected to in the occurrences of Q.



(a) Subgraph Pattern Q

(b) Occurrence of Q

(c) Replica of Q

Figure 4: All the edge in Figure 4(b) from A to B, B to C has the edge label a, b respectively. Figure 4(c) is replicated from Figure 4(b) without edge labels.

3.2.2 Generation of a Replica Graph. Algorithms 1 and 2 together describe the complete procedure to construct the replica graph for a subgraph pattern. Replica construction for any pattern R requires knowledge of the replica graph of the pattern Q that is extended to generate it. Henceforth, we refer to such a pattern R as a *child pattern* of Q and Q as the *parent pattern* of R. Since the search procedure processes a pattern only after its parent, the replica graph of the *parent pattern* can be used for constructing the replica graph of the *child pattern*.

Algorithm 1 essentially describes a procedure to find every mapping (also referred to as *instance*) of the *child pattern* in the input graph using the mappings of the *parent pattern* and thus obtain its *replica* graph. The algorithm begins with a depth-first search (DFS) procedure (line 1) executed on the *parent pattern* Q, selecting the vertex from which the *candidate edge* is extended as the *root*. We call this vertex the *extending vertex* in Q. The edges encountered in the depth-first traversal are recorded in an ordered list called the *DFS List*, which helps guide the instance enumeration procedure performed subsequently. The algorithm iterates over all mappings of the *extending index* in the *replica* of the *parent pattern* (line 2) and attempts to enumerate all (if any) instances of the *child pattern* in the data graph one-by-one. More specifically, for every vertex m of $replica(Q)$ that maps to the *extending vertex* u, the algorithm iterates over its adjacent edges in the input graph that map to the *candidate edge*

Algorithm 1: GETREPLICA

Input: Graph G, parent Q, $replica(Q)$, child R, extending index: $u \in V(Q)$, extension: $candidate\ edge(u, v) \in E(R)$
Output: $replica(R)$

1 $DFS\ List \leftarrow$ get rooted DFS of Q with u as $root$
2 $instance := \emptyset$
3 **foreach** $u' \in Mappings(u,\ replica(Q))$ **do**
4 $\quad instance \leftarrow \{(u, u')\}$
5 $\quad$ **foreach** edge $e(u', v') \in E(G)$ that maps to $candidate\ edge(u, v)$ **do**
6 $\quad\quad instance \leftarrow instance \cup \{(v, v')\}$
7 $\quad\quad \mathbb{I} \leftarrow$ FINDALLINSTANCES(R, $instance$, $\mathbb{I}$, $DFS\ List$, ...)
8 $\quad\quad$ UPDATEREPLICA($replica(R)$, $\mathbb{I}$, ...)
9 $\quad\quad instance \leftarrow instance \setminus \{(v, v')\}$
10 $\quad instance \leftarrow instance \setminus \{(u, u')\}$
11 **return** $replica(R)$

(line 3) and invokes the find all instances method (Algorithm 2) to enumerate every *instance* of *child pattern* R from $replica(Q)$ that contains this adjacent edge. Algorithm 2, thus invoked, recursively enumerates all instances of R in a depth-first manner following the *DFS List* of Q (computed earlier). In the general case (lines 4-10), the algorithm selects an appropriate graph edge as the mapping of an edge in the *child pattern* R (line 5) and recursively invokes the method to find a mapping of the next edge in the *DFS List* that is consistent with the edge mappings selected so far (line 7). Once all edges in the *DFS List* have been mapped to graph edges, the base case (lines 1-3) gets executed where the instance of R thus enumerated is simply returned. Thus, the set of all instances found is returned at the end of Algorithm 2 (line 10) and recorded by Algorithm 1 to update the *replica* graph (line 6, Algorithm 1). In the update step, the algorithm simply appends all edges of every instance in $\mathbb{I}$ to the existing set of edges of $replica(R)$, and also updates the *mappings set* for every vertex of pattern R to include the vertices of the newly-discovered instances in $\mathbb{I}$.

This replica storage strategy not only builds a foundation for the sequential efficient correlation calculation, but also benefits the MNI support counting in the single large graph since we can directly get $\sigma(R)$ when we record the replica of R just by counting all the sizes of the images $M(v)$, where $v \in R$.

3.2.3 Indexing to Facilitate Replica Deletion.

3.3 Subgraph Extension

3.3.1 Extension Rule. We subscript the node patterns in a subgraph patterns to identify their order of discovery.

Algorithm 2: FINDALLINSTANCES (EXACT)

Input: Graph G, parent Q, $replica(Q)$, child R, $DFS\ List$, partial isomorphism of R: $instance$, $\mathbb{I}$
Output: $\mathbb{I}$: set of all instances of R in G consistent with input partial isomorphism $instance$

1 **if** $|instance| = |V(R)|$ **then**
2 $\quad$ **return** $instance$
3 **else**
4 $\quad e(p, c) :=$ NEXTQUERYEDGE($DFS\ List$, ...)
5 $\quad P_c :=$ FILTERCANDIDATES($instance, c$, ...)
6 $\quad$ **foreach** $w \in P_c$ such that w is not yet matched **do**
7 $\quad\quad instance \leftarrow instance \cup \{(c, w)\}$
8 $\quad\quad \mathbb{I} \leftarrow \mathbb{I} \cup$ FINDALLINSTANCES($R, instance$, ...)
9 $\quad\quad instance \leftarrow instance \setminus \{(c, w)\}$
10 $\quad$ **return** $\mathbb{I}$

DEFINITION 3 (VERTICES SUBSCRIPTING). *Let Q be a frequent subgraph in T, apart from vertex identification v_i, we use another metric to subscript the vertices in Q by the order we discover the pattern Q, denoted as $S_Q = (s_0, s_1, ..., s_n)$, where $n = |V_Q|$. And for s_i and s_j, if $i < j$, then the vertex v_i is discovered earlier than the vertex v_j.*

Taking advantage of subscripting, it is easy to get the right-most path of a subgraph. Then, we only grow the new vertices in right-most path. Corresponding to the best-first-search strategy, the node with highest MNI support will be chosen, suppose Q. Then, for all the vertices in its right-most path of the replica of Q, we try one edge extension from them respectively, see Algorithm 3.

Algorithm 3: SUBGRAPHEDGEEXTENSIONS

Input: Graph G, parent Q, $replica(Q)$
Output: $Ex(Q)$: set of candidate edge extensions for Q

1 $Ex(Q) := \emptyset$
2 $rmpath \leftarrow$ right-most path of Q from $DFS\ Code(Q)$
3 **foreach** $v \in rmpath$ **do**
4 $\quad$ **foreach** $v' \in Mappings(v, replica(Q))$ **do**
5 $\quad\quad E \leftarrow$ set of all edges $(v', w') \in E(G)$ extending Q
6 $\quad\quad Ex(Q) \leftarrow Ex(Q) \cup E$
7 **return** $Ex(Q)$

3.3.2 Duplicated Subgraph Prunning. To avoid the duplicated subgraph judgement, we take the advantage of the DFS code, and the minimum DFS code in gSpan[**?**], whenever a subgraph is founded, we get its DFS code, denoted as $C(Q)$ and remodel this subgraph by using this code. Then, we build the minimum DFS code of this subgraph Q. This minimum

DFS code, denoted as $Z(Q)$ is the canonical label of this subgraph in our definition.

We use a dictionary $\mathbb{D}$ to store all the minimum DFS code we have discovered so far. When a subgraph Q is discovered and its graph code $C(Q)$ has been transformed to minimum DFS code $Z(Q)$, we search $Z(Q)$ in the dictionary. If $Z(Q) \in \mathbb{D}$, then Q must have been discovered before, so we prune Q.

3.4 Correlation Computation

3.4.1 Global Index. We use a global index to record all the distance information. Each vertex of the data graph stores two catogories of distance information.

- **Proximity Vertices.** For each $u \in G$, we store the information of proximity vertices of u, denoted as $CorV(u)$, for each vertex $u \in CorV(u)$, there exists $d(u, v) \leq h$.
- **Proximity Patterns.** For each $u \in G$, we store the information of proximity patterns of u, denoted as $CorP(u)$, for each pattern $Q \in CorP(u)$, suppose the instance-groups of Q is $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q)}\}$, there exists $I' \in \mathbb{I}'$, $\exists v \in I'$, $d(u, v) \leq h$.

With the global index acquired before the search, there is no need to consider anything about the distance (hop-constraints) in the search steps. The detail of the maintenance of these two indices is specified in Section 3.4.2.

3.4.2 Calculating Correlation.

Definition 4 (Positive Instance Group). *Given two subgraphs Q_1 and Q_2 in the input graph G, their instance-groups $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q_1)}\}$ and $\mathbb{J}' = \{J'_1, J'_2, \ldots, J'_{\sigma(Q_2)}\}$, respectively, and a user-defined distance-threshold h, assume that $\sigma(Q_1) \leq \sigma(Q_2)$. Then, we say an instance group of Q_1, $I'_i \in \mathbb{I}'$ is a* **positive instance group** *of Q_2 if following condition is satisfied.*

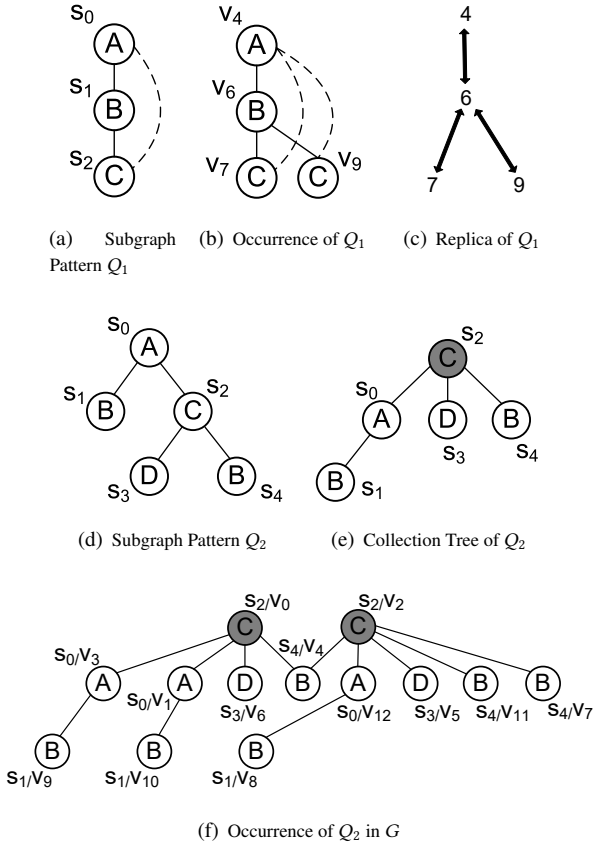
$$\exists J' \in \mathbb{J}, \exists u \in I', \exists v \in J', d(u, v) \leq h \tag{3}$$

Then, we denote $P(I'_i, Q_2, h) = 1$, otherwise $P(I'_i, Q_2, h) = 0$.

Theorem 2. *Let Q_1, Q_2 be two subgrpah patterns, and an instance group of Q_1, I'. Then, if $Q_2 \in \cap CorP(v), v \in I'$, $P(I'_i, Q_2, h) = 1$. otherwise, $P(I'_i, Q_2, h) = 0$.*

The process of correlation calculation could be considered as a **collection**. Suppose we are calculating the correlation of Q_1, i.e. $\tau(Q_1, Q_k, h)$, for all $Q_k \in Cor(T)$. Taking advantage of the global index in Section 3.4.1, by traversing all the vertices in a group can we know the correlation $\tau()$ of all the v. We collect these sets one-by-one and finally we could know all the correlation of Q_1, i.e. $\tau(Q_1, Q_k, h)$, for all $Q_k \in Cor(T)$.

3.4.3 Replica Structure Transformation. It is more convenient to implement an efficient collection in a hierarchical structure, like tree. As a result, to initialize a collection, we first transform the search space of the collection to a particular tree.



(a) Subgraph Pattern Q_1 (b) Occurrence of Q_1 (c) Replica of Q_1

(d) Subgraph Pattern Q_2 (e) Collection Tree of Q_2

(f) Occurrence of Q_2 in G

Figure 5: Figure 5(a) is the subgraph pattern of Q_1, Figure 5(c) is a replica of the occurrence of Q_1 in Figure 5(b) without any backward edges (the dotted lines are the backward edges in Q_1). Figure 5(e) is the same subgraph pattern Q_2 with Figure 5(d), with collection tree rooted at the group center s_2. Figure 5(f) is the occurrence of Q_2 in data graph G.

Prior to the detailed operations, we first define the group center and the collection tree of a pattern.

Definition 5 (Group Center and Center Subscript). *Given a subgraph Q, and the instance-groups of Q, $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q)}\}$, a group center u of I'_i is the vertex having the minimum MNI support among all $v \in I'_i$, denoted as $u = center(I_i)$, a center subscript is the vertex subscript has the MNI support of the images, denoted as $centerSubscript(Q)$. i.e. $centerSubscript(Q) = i$, where $M(s_i) = \sigma(Q)$.*

Definition 6 (Collection Tree). *Given a subgraph Q, a collection tree of Q, $CT(Q)$ is tree transfromed from the tree structure replica, and rooted at $centerSubscript(Q)$.*

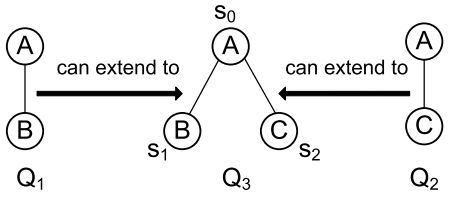


Figure 6: **Subgraph Q_1 and Q_2 are the subgraphs of Q_3.**

Then, we consider the group center as the root of the tree. During the collection, the collection is initiated from the root of the tree.

Prior to the correlation computations for Q, we first update the distance index of proximity patterns using the data of proximity vertices. That is, for all $u \in Q$, for each $v \in Cor(u)$, $CorP(v) = CorP(v) \cup Q$.

LEMMA 4. *Let Q be a subgraph, for all v in data graph G, if $v \in \cup CorV(u), u \in Q$, then there must exists $Q \in CorP(v)$.*

After the update of the distance index, we operate the correlation calculation of subgraph Q. This process includes two phases.

• **Collection Phase:** For each group center $c \in Center(Q)$, all instances of Q including c are enumerated in a depth-first manner similar to the recursive instances-enumeration technique of Algorithm 2. A set union of patterns contained in $CorP(u)$ is performed across every vertex u of an instance group. At the same time, every vertex v in the *proximity vertices map* of u ($CorV(u)$) records the proximity of pattern Q in its *proximity patterns map* ($CorP(v)$).

$$Collect(c, Q) = \cup\{CorP(v) | v \in instancegroup\} \quad (4)$$

• **Computation Phase:** The correlation between patterns Q_1 and Q_2 is easily stated by calculating the count of instance groups of Q_1 that are positive instance group of Q_2.

Clearly, a instance group I'_i of Q_1 is a positive instance group Q_2, i.e. $P(I'_i, Q_2, h) = 1$ if and only if

$$u = groupCenter(I'_i), Q_2 \in Collect(u, Q_1) \quad (5)$$

Then, we sum all the results to get the correlation.

$$\tau(Q_1, Q_2, h) = \sum_{i}^{\sigma(Q_1)} P(I'_i, Q_2, h) \quad (6)$$

Algorithm 4 summarises the steps described above.

3.4.4 Avoiding Subgraph/Supergraph Correlation. As we mentioned in Section 2.2, we do not want to consider the correlation of Q_1, Q_2 if Q_1 is a subgraph or a supergraph of Q_2. If Q_2 is extended from Q_1, we could easily know Q_2 is the supergraph of Q_1 and skip their correlation calculation. However, there are more troublesome conditions.

Algorithm 4: OPERATE

Input: Graph G, Q, $replica(Q)$, hop h, $CorV$, $CorP$
Output: $\tau(Q, Q_k, h)$, updated Top k order
1 **foreach** vertex $m \in Mappings(center, replica(Q))$ **do**
2 $\mathbb{I} \leftarrow$ set of all instances I such that $(center, m) \in I$
3 **foreach** $u \in V(replica(Q))$ constituing an *instance* in $\mathbb{I}$ **do**
4 $\forall v \in CorV(u), CorP(v) \leftarrow CorP(v) \cup \{Q\}$
5 $Collect(m, Q) \leftarrow Collect(m, Q) \cup CorP(u)$
6 **foreach** pattern Q_k in set operated **do**
7 $\tau(Q, Q_k, h) \leftarrow |\{m \mid m \in Mappings(center, replica(Q)) \wedge Q_k \in Collect(m, Q)\}|$
8 Update Top k order with computed correlation (τ) values
9 operated $\leftarrow$ operated $\cup \{Q\}$

EXAMPLE 4. *In Figure 6, suppose Q_1 extended to Q_3 and Q_3 knows that Q_1 is its subgraph. However, during* Op(Q_3)*, Q_3 must avoid the correlation $\tau(Q_3, Q_2, h)$ because Q_2 is also the subgraph of Q_3.*

Obviously, we can not use subgraph isomorphism to check this relationship of Q_1, Q_2 before the correlation calculation since it is too expensive. We use following approach to rapidly get the answer.

We first add one more rule to best-first-search, if $\sigma(Q_1) = \sigma(Q_2)$, Q_1 has higher priority to be operated if and only if:

$$|V_{Q_1}| < |V_{Q_2}|$$

According to this rule, together with downward-closure, we could always guarantee that Q_1 is extended before Q_2.

For each subgraph Q, it maintain a subgraph set $SubRec(Q)$, recording all of its subgraphs. Under our assumption, Q_1 can extend to Q_2 so that Q_2 knows that Q_1 is the subgraph of Q_2 and all the subgraphs of Q_1 are also the subgraphs of Q_2. As a result, if Q_1 can extend to Q_2, then we operate $SubRec(Q_2) = SubRec(Q_2) \cap Q_1 \cap SubRec(Q_1)$. As Op($Q$) is occurring, if $Q_i \in SubRec(Q)$, we skip $\tau(Q, Q_i, h)$.

3.5 Mining Algorithm

The complete mining algorithm consists of an initialization step and the search algorithm to compute top-k pairs of correlated subgraph patterns. The details of these procedures are described in the following subsections.

3.5.1 Initialization. Algorithm 5 specifies the step-by-step operations of the initialization procedure.

The algorithm begins with a brute-force search to obtain all frequent edges in the data graph (line 2). Once the set of frequent edges (stored in variable *F_edges*) is computed, the algorithm executes a breadth-first search (BFS) procedure (line 4) for every vertex u that constitutes an edge in *F_edges*

Algorithm 5: INITIALIZATION

Input: Graph G, Min-Sup: min_sup, hop value: h
Output: frequent edges set: F_edges, proximate vertices index: $CorV$, modified data graph: G

1 $F\_edges \leftarrow \{e \mid e \in E(G),\ \sigma(e) \geq min\_sup\}$
2 **foreach** $u \in V(G)$ such that $(u, u') \in F\_edges$ **do**
3 BFS from u to all $v \in V(G)$, such that min distance $d(u, v) \leq h$
4 $CorV(u) \leftarrow$ set of all v obtained above
5 $NF\_edge \leftarrow E(G) \setminus F\_edge$
6 Remove NF_edge from $E(G)$
7 **return** F_edges, $CorV$, G

to obtain all vertices satisfying the hop constraint with respect to u. The set of these vertices is stored in the $CorV$ dictionary mapped to u (line 5). Thus, the set of proximate vertices for every vertex constituting a frequent edge is obtained and stored. Finally, the algorithm deletes the set of infrequent edges from G to obtain the modified data graph (lines 7-8). Infrequent edges are removed since these have no bearing on the algorithm hereafter and doing so accelerates the search procedure.

3.5.2 Search Steps. Following the initialization, the search algorithm as specified in Algorithm 6 is executed.

Algorithm 6: Search

Input: Graph G, Min-Sup: min_sup, frequent edges set: F_edges, $CorV$, Generated patterns dictionary: $\mathbb{D}$
Output: top_k pairs of correlated subgraph patterns

1 Initialize Search Queue with F_edges
2 **while** CEASINGCONDITION is not satisfied **do**
3 subgraph $Q \leftarrow$ Search Queue.Pop()
4 Execute OPERATE(Q)
5 $Ex(Q) \leftarrow$ SUBGRAPHEDGEEXTENSIONS(Q)
6 **foreach** candidate edge $e(u, v) \in Ex(Q)$ **do**
7 child $Q' \leftarrow Q$ extended with e
8 **if** $DFS\ Code(Q') \notin \mathbb{D}$ **then**
9 $replica(Q') \leftarrow$ GETREPLICA($Q', u, e, \dots$)
10 Compute $\sigma(Q')$ from $replica(Q')$
11 **if** $\sigma(Q') \geq min\_sup$ **then**
12 Push Q' into SEARCHQUEUE
13 Record $DFS\ Code(Q')$ in $\mathbb{D}$
14 **else**
15 continue
16 **return** top_k correlated pairs

The algorithm begins with the initialization of a priority queue called $Processing\_Queue$ (line 1) that stores subgraph patterns scheduled for correlation computation with the property that a pattern with a higher MNI-support is accorded a higher priority following the best-first search strategy (Section **??**). $Processing\_Queue$ is initialized with the set of frequent edges (queued in the order of decreasing support values). During search, as long as the *ceasing condition* (Section **??**) remains unsatisfied, the subgraph pattern at the front of $Processing\_Queue$ is selected for correlation computation and extension. Correlation computation takes place in method Operate (Algorithm 4) wherein the top-k set can also be updated. This is followed by the computation of all possible one edge extensions in Q in method Subgraph Edge Extensions (line 5). For every candidate extension in $Ex(Q)$, the DFS Code of the resulting child subgraph pattern Q' is tested for presence in dictionary $\mathbb{D}$. A match in $\mathbb{D}$ indicates that Q' has already been generated previously, so the algorithm proceeds with the MNI-support computation only if there is no match (line 8). MNI-support computation for Q' requires the construction of its *replica* structure, which the algorithm computes and stores through the invocation of Get Replica method described (line 9). Note that the *replica* structure for a child pattern not only establishes the MNI-support but also allows correlation computation in method Operate. If the child pattern's MNI-support value exceeds the threshold min_sup, it is pushed into $Processing\_Queue$ (line 12) to be (possibly) processed in a later iteration of the outer loop. $DFS\_Code(Q')$ is recorded in $\mathbb{D}$ (line 14).

4 APPROXIMATE ALGORITHM

Exact *replica* construction (Section 3.2.2) for a subgraph pattern requires searching for every instance of the pattern in the data graph using the *replica* of its parent pattern. In FINDALLINSTANCES (Algorithm 2), we identify all instances one-by-one in a depth-first guided search. However, performing this computation can be expensive since subgraph isomorphism is known to be an *NP-hard* problem and thus the computational complexity is worst-case exponential in the pattern size. Moreover, the number of instances of a pattern generally grows exponentially with increasing density and size of the data graph. As a result, a *complete* enumeration of instances such as in Algorithm 2 does not scale well to dense or large graphs. This establishes the need for a faster strategy for *replica* construction.

We now make an important observation: to match a vertex $v \in V(G)$ and appropriate edges incident to v in $V(replica(R))$ and $E(replica(R))$ respectively for a pattern R, the identification of *all* instances of R containing v might not be necessary. In other words, by identifying more instances containing v than might be necessary, we do not gain any additional information about the *replica* graph structure. This suggests that

repeated traversals on vertices during the course of identifying instances can perhaps be reduced without losing structural information about the *replica*.

Algorithm 7 describes an alternative strategy to Algorithm 2 to implement the idea for accelerated *replica* construction by reducing repeated traversals on "already-explored" vertices. We attempt to avoid enumerating those instances during construction that do not provide new information about the *replica* structure as a strategy to reduce computational complexity. Before discussing this method, we define some additional data structures for the *replica*:

• **Potential children set** $P_{v,c}$
(defined for each $v \in V(replica(Q))$ mapped to $p \in V(Q)$ for each child $c \in children(p, Q) \in V(Q)$): stores every replica vertex $w \in V(replica(Q))$ such that $c \in children(p, Q)$ can potentially map to w (i.e. w is a candidate for c).

• **Enumerated set** E_v
(defined for each $v \in V(Q)$): stores every vertex $w \in replica(Q)$ that maps to v such that: $\forall c \in children(v, Q),\ P_{v,c} = \emptyset$. In other words, E_v stores all its mappings that have been "fully-explored" for all child mappings.

• **Confirmed children set** $C_{v,c}$
(defined for each $v \in V(replica(Q))$ mapped to $p \in V(Q)$ for each child $c \in children(p, Q) \in V(Q)$): stores every replica vertex $w \in V(replica(Q))$ such that: (1) there exists at least one instance including the mappings (p, v) and (w, c), and (2) $w \in E_c$

Algorithm 7 is a recursive algorithm. In the general (recursive) case (lines??-??), it begins by invoking NEXTQUERYEDGE which returns one edge at a time from $E(Q)$ in the order of the rooted *DFS List*. Edge $e(p, c)$ thus returned connects vertices $p, c \in V(Q)$ such that $c \in children(p)$ as per the *DFS List* and c is the pattern vertex to be matched next (p is matched to $v \in V(replica(Q))$). The algorithm then checks for the existence of the potential children set for storing candidate vertices for matching c, given that p is mapped to v. If this set $P_{v,c} (\subseteq V(replica(Q)))$ does not exist, it is computed by invoking FILTERCANDIDATES which stores all vertices $w \in V(replica(Q))$ such that: (1) w is contained in the *replica* adjacency list of v; (2) c exists in the inverse mapping list of w. That is, $\forall w \in P_{v,c}$, $c \in InverseMap(w)$; (3) The edge label of $(v, w) \in E(replica(Q))$ matches that of $(p, c) \in E(Q)$. Thus, set $P_{v,c}$ is constructed once when a match for c is being sought for the first time with p having been mapped to v, and indexed. Similarly, confirmed children set $C_{v,c}$ is initialized as an empty set and indexed. In the event of any subsequent visit, the algorithm simply considers candidates remaining in $P_{v,c}$ for matching c. Next, for every vertex $w \in P_{v,c}$ such that w has not already been mapped to some other pattern vertex in the current *instance*, the algorithm attempts the mapping (c, w) in *instance* and recursively calls FINDALLINSTANCESAPPROX to match remaining pattern vertices following the *DFS List* of edges. It sets a boolean *InstanceFound* to *True* if it finds at least one *instance* in the recursive call and appends the set of all found instances to $\mathbb{I}$. During the recursive call, if w gets inserted into the enumerated set for c (E_c), it is transferred from $P_{v,c}$ to $C_{v,c}$. Finally, (c, w) is removed from *instance* and the next valid candidate in $P_{v,c}$ tried as a possible matching for c in the following iteration. Suppose, the algorithm fails to find any *instance* even after considering every $w \in P_{v,c}$. (A possible scenario where this could happen is if $P_{v,c} = \emptyset$, after all original candidates have been confirmed and transferred to $C_{v,c}$). In such a scenario, *InstanceFound* would remain false after the first loop. The algorithm would then start iterating over $C_{v,c}$, the set of confirmed children till the time an *instance* is found for some $w' \in C_{v,u}$ and *InstanceFound* is set to *True*. Finally, if the candidates set $P_{v,c'}$ for **every** $c' \in children(p)$ is exhausted, v is appended to the enumerated set for $p(E_p)$. This means that *all* possible candidate child mappings have been traversed and confirmed. Note that in lines ??-??, child mapping w if inserted into E_c gets tranferred from $P_{v,c}$ to $C_{v,c}$. This is because any subsequent searches for possible mappings of c at v can ignore all $w' \in C_{v,c}$ since such a w' has been completely "enumerated" and thus matching it to c would result in repeated traversals over already explored portions of the graph. The *Base Case* (lines 1-3) occurs when the algorithm finds an instance of the child pattern R (i.e., $|instance| = |V(R)|$). The algorithm records the mappings of every $l' \in Leaves(Q)$ in the corresponding enumerated sets, since there cannot exist any further child mappings below leaves. Finally, the *instance* is returned.

5 ANALYSIS

5.1 Complexity Analysis

We denoted the number $Op(Q)$ occurs for all the subgraphs Q as f and the average degree of the vertices is d, the number of the vertices in the data graph is $|V|$.

• **Time Complexity.** For f subgraphs, suppose Q is the subgraph selected, i.e. Op(Q) is occurring. Suppose there are n group centers, the time cost of each collection process is bounded by $|V|d$, and the cost of proximity pattern set union is f. And the event of Found(Q) cost most $|V_Q||V|d$, which is dominated by the collection cost. Thus, the total time complexity is bounded by $O(|V|ndf^2)$.

• **Space Complexity.** As mentioned in Section **??**, we remove the replica of Q if Op(Q) occurs so that there would be no space cost of Q anymore. Thus, the only storage which domainates the space cost is the storage of global index. For at most $|V|$ vertices in the data graph, each of them needs to store a proximity vertex set, bounded by d^h, , commonly $h \leq 2$ (The correlation is not interesting if h is not very small). Besides, the proximity pattern set of each vertex is

Algorithm 7: FINDALLINSTANCESAPPROX

Input: Graph G, parent: Q, $replica(Q)$, child: R, parent edge: $(u, v) \in E(G)$ mapped to $(u', v') \in E(R)$, *DFS List* of Q rooted at *extending index*, *instance*, $\mathbb{I}$

Output: relevant *instances* $\mathbb{I}$ of pattern R in G such that $\forall I \in \mathbb{I}$, $(u, v) \in E(I)$ as a mapping of (u', v')

Base Case :

if $|instance| = |V(R)|$ **then**
  Update $E_{l'}$ for each $l' \in Leaves(Q)$ $[[E_{l'} \leftarrow E_{l'} \cup \{instance(l')\}, \forall l' \in Leaves(Q)]]$
  return *instance*

Recursive Case :

$e(p, c) \coloneqq$ NEXTQUERYEDGE(*DFS List*, ...) $[[(p, c) \in E(Q),\ c \in children(p)]]$

$v \coloneqq$ Mapping of p in *instance* $[[v \in V(replica(Q)) \wedge (p, v) \in instance]]$

if $P_{v,c}$ does not exist **then**
  $P_{v,c} \coloneqq$ FILTERCANDIDATES(*instance*, v, c, ...) $[[\forall w \in P_{v,c}\ ((w \in V(replica(Q)) \wedge (w \in adjList(v)) \wedge (c \in InverseMap(w)) \wedge (L(v, w) = L(p, c)))]]$
  $C_{v,c} \coloneqq \emptyset$

boolean InstanceFound $\coloneqq$ False

foreach $w \in P_{v,c}$ such that w is not yet matched **do**
  $instance \leftarrow instance \cup \{(c, w)\}$
  $\mathbb{I}' \leftarrow$ FINDALLINSTANCESAPPROX(R, e, *instance*, *DFS List*)
  if $\mathbb{I}' \neq \emptyset$ **then**
    $\mathbb{I} \leftarrow \mathbb{I} \cup \mathbb{I}'$
    $InstanceFound \leftarrow$ True
  if $w \in E_c$ **then**
    $P_{v,c} \leftarrow P_{v,c} \setminus \{w\}$
    $C_{v,c} \leftarrow C_{v,c} \cup \{w\}$
  $instance \leftarrow instance \setminus \{(c, w)\}$

foreach $w' \in C_{v,c}$ such that w' is not yet matched and *InstanceFound* is False **do**
  $instance \leftarrow instance \cup \{(c, w')\}$
  $\mathbb{I}' \leftarrow$ FINDALLINSTANCESAPPROX(R, e, *instance*, *DFS List*)
  if $\mathbb{I}' \neq \emptyset$ **then**
    $\mathbb{I} \leftarrow \mathbb{I} \cup \mathbb{I}'$
    $InstanceFound \leftarrow$ True
  $instance \leftarrow instance \setminus \{(c, w')\}$

if $\forall c' \in children(p),\ P_{v,c} = \emptyset$ **then**
  $E_p \leftarrow E_p \cup v$

return $\mathbb{I}$

bounded by f. Thus, the total space complexity is bounded by $O(|V|(d^h + f))$.

5.2 Search Strategy Analysis

• **vs. Depth-first-search.** Suppose Min-sup is small and k in Top-k is also small, then we just need to find a few correlations with high τ value to get the result. However, if DFS goes to a subgraph Q with lower support $\sigma(Q)$, then events Found(Q) and Op(Q) would still occurs. Until $\sigma(Q)$ is smaller than the small value Min-sup can we go to another branch. Obviously, we have already calculated a large amount of insignificant correlations.

• **vs. Breath-first-search.** BFS seems to perform better than DFS. However, it is still not good enough. Consider a subgraph Q with small subgraph size $|V_Q|$ and small value of $\sigma(Q)$, events Found(Q) and Op(Q) would still occurs and it just need not to go as deep as DFS. Still, we have calculated some of insignificant correlations.

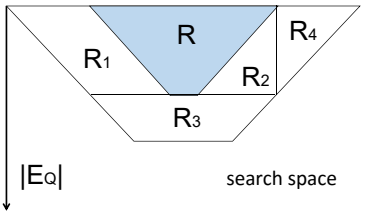


Figure 7: In the whole search space, the smaller trapezoid R is the solution space, and the larger trapezoid, $R + R_1 + R_2 + R_3 + R_4$ is the regions in the search space where $\sigma(Q) \geq$ Min-sup.

EXAMPLE 5. *In Figure 7, to explore the solution region R, if expanding by DFS strategy, the region we explore finally will be $R + R_1 + R_2 + R_3$. If expanding by BFS, the region we explore will be $R + R_1 + R_2 + R_4$. If we expanding by best-first-search, the region will fit the solution region R as much as possible.*

• **DFS lexicographic order edge discovery.** Following the order which is correspond to DFS lexicographic order[**?**] during the new edge discovery, we could guarantee that, the if Q is the first time being discovered, then there must exist $C(Q) = Z(Q)$. Both DFS and BFS strategy of search space expanding could take advantage of this property. What we should guarantee is for any two subgraphs Q_1, Q_2, if they have the same edge number, i.e. $|E_{Q_1}| = |E_{Q_2}|$, and Q_1 is discovered earlier than Q_2, then Q_1 must have a smaller DFS lexicographic order. In [**?**], this property is utilized by carrying out a DFS strategy.

EXAMPLE 6. *In Figure 8, $C(Q_3) = Z(Q_3)$ and $C(Q_4) \neq Z(Q_4)$. To achieve the guarantee above, we should guarantee that Q_3 is discovered earlier than Q_4. If we expand the search*

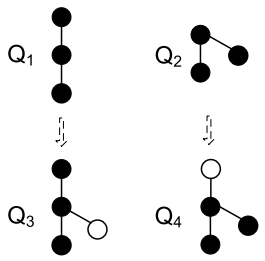


Figure 8: Subgraph Q_1 and Q_2 are generated by DFS lexicographic order, i.e. $C(Q_1) = Z(Q_1)$, $C(Q_2) = Z(Q_2)$. Q_1 extends to Q_3, Q_2 extends to Q_4, Q_3, Q_4 are isomorphic. If follow the DFS lexicographic order, Q_1 is discovered earlier than Q_2, Q_3 is discovered earlier than Q_4.

space by DFS, the discovery order is Q_1, Q_3, Q_2, Q_4, which satisfies the guarantee. If we expand the search space by BFS, the discovery order is Q_1, Q_2, Q_3, Q_4, which satisfies the guarantee. If we expand the search space by best-first-search, since there may exist $\sigma(Q_2) > \sigma(Q_1)$, so that Q_4 may be discovered earlier than Q_3, which break the guarantee.

Besides, we also consider the disadvantage of best-first-search. If we strictly follow the DFS lexicographic order to generate the subgraph patterns, there is no need to create a dictionary $\mathbb{D}$ because we could prune a subgraph Q directly if $C(Q) \neq Z(Q)$. However, compared with the time cost of generate the minimum DFS code, the search in the dictionary cost only constant time. As a result, the additional time cost of the dictionary is small enough to be ignored. Thus, best-first-search outperform both DFS and BFS without doubt.

6 EXPERIMENTAL RESULTS

In this section, we provide the experimental result of our algorithm, noted as CSM. We also provide the instance-based method, noted as GROWSTORE, for comparison of frequent subgraph mining. In addition, we use GRAMI-VF3 as a comparison metric for subgraph isomorphism. GraMi is the state of the art method for frequent subgraph mining and VF3 is used for getting all instances of the frequent patterns. The algorithm is implemented in C++11 and compiled using gcc 7.4.0. All experiments are conducted on a Linux(Ubuntu 18) machine with 96 cores running at 2.1GHz with 251GB RAM and 8.5TB disk. Our experimental machine used a large memory size to accommodate the memory requirements of GROWSTORE.

6.1 Experimental Setup

Datasets - Our experiments are based on the following real graph datasets. Their main characteristics are listed in Table 1.

Chemical. This dataset contains the undirected graph of a chemical compound in MCF7 Dataset of X.F. Yan[**?**] with 207 vertices, 215 edges, 4 distinct node labels. Each node represents a chemical atom.

Table 1: Datasets and characteristics

Datasets	Nodes	Edges	Distinct node labels
Chemical	207	205	4
Yeast	4000	79000	4
MiCo	100,000	1,080,298	26
LastFM	1,092,145	5,237,032	82,526
DBLP Coauthor	1.7M	7.4M	11K
DBLP Citation	3.2M	5.1M	11K
Citeseer	3312	4591	5

Citeseer. Each vertex is a publication and label is the area of research in which that paper is published. Two vertices have an edge if one of the two papers is cited by the other and the edge label is the similarity measure between the two papers with a smaller label denoting stronger similarity. It has 3312 vertices, 4591 edges, ?? distinct node labels and 100 distinct edge labels(0-100) in it's original form but we scaled down the edge labels to 5.

Yeast. This dataset contains the undirected graph of proteins using the function as label, with 4K vertices, 79K edges and 26 distinct node labels.

Mico. It models Microsoft co-authorship information. Each vertex is an author and label is the field of interest. Edges represent collaboration between two authors and the edge label is the number of coauthored papers. It has 100K vertices, 1.08M edges and ?? distinct node labels.

LastFM. Each vertex is a user in the FM social network and label is the most frequent singer or music band that this user listens to. Two users who interact with each other will have a edge for connection. It has 1.1M vertices, 5.2M edges and 83K distinct node labels.

DBLP coauthor. Each vertex is an author and label is the conference in which that author is most publishe. Two vertices have an edge if the two authors have co-authored in at least 1 paper. It has 1.7M vertices, 7.4M edges and 11K distinct node labels.

DBLP citation. Each vertex is a publication and label is the conference in which that paper is published. Two vertices have an edge if one of the two papers is cited by the other. It has 3.2M vertices, 5.1M edges and 11K distinct node labels.

Parameters. There are 3 different input parameters in the problem of correlated subgraph mining. The minimum image based support Min-sup, the k value of our Top-k results, and the hop-constraint value h. We show the results of our experiments by varying the values of these input parameters. Increasing the value of Min-sup results in a decrease of frequent subgraphs, increasing the value of k results in an increase of output number, increasing the value of h results in more correlated subgraph pairs.

Baselines We compare our approximate algorithm with three baselines. The first one is the exact algorithm - CSM-EXACT (Section 3) . The second one is GROWSTORE which is an

instance based approach for frequent subgraph mining. The third one is GRAMI-VF3 in which we use GRAMI(??) for frequent subgraph mining and VF3(??) for subgraph isomorphism of all the frequent patterns which we get as an output from GRAMI . We do not do correlation computation for *GraMi plus VF3* as doing subgraph isomorphism for all the frequent patterns takes a significant amount of time which is larger than our entire approximate algorithm. GROWSTORE shows the advantage that we have in terms of memory for storing all the instances as discussed later.

Time Comparison. The baselines are very slow for large and dense datasets so we compare the running times of our approximate algorithm with these baselines with an additonal constraint of size bound in patterns where we do not explore patterns which have more than 5 vertices. We do this because as the pattern size increases, the cost of computing the instances for that pattern grows exponentially. Figure 9 shows comparison of running times with different baselines for different datasets when support is varied at $k = 20$, $h = 1$.

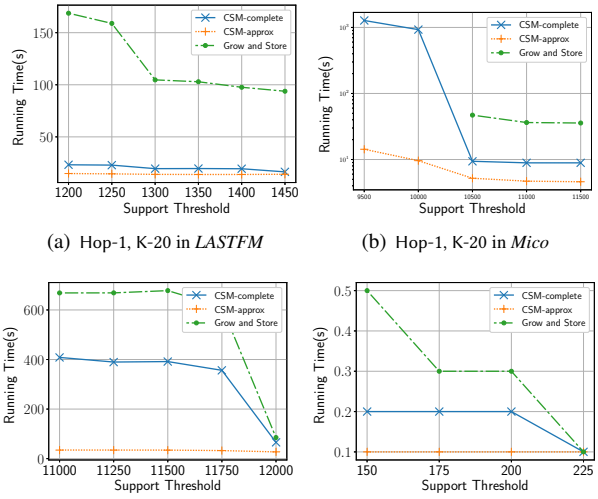
Memory Comparison and Usefulness of Replica. Replica provides an efficient way to store all the instances of a subgraph pattern whereas GROWSTORE stores all the instances. So in a frequent subgraph mining setting, GROWSTORE does work for small and less dense graphs but as the graph size increases or the graph becomes dense, the number of instances grow exponentially and thus the storage cost becomes too high. In figure 10, we provide comparison of the memory requirements of our approach - CSM and GROWSTORE and it can be clearly seen that our approach helps in saving memory by a huge margin. In the plot of LASTFM, we have just one point for GROWSTORE because it gave memory error on a 256*GB* machine for lower supports. The same happened for a smaller dataset like Citeseer as well for lower support.

6.2 Performance.

We provide the experimental results of our algorithm on large datasets from now on.

Running Time Results. Figure 11 shows results of our approximation on four datasets at different hops and varied support. The running time decreases with increasing support. As we increase hops, time and space required to compute proximate vertices index($CorV$) increases and as a result, the time taken for correlation computation also increases due to more correlations obtained. However, this can sometimes lead to early termination as the top K set gets filled early. If the decrease in time due to this dominates the increase in time which is there, time may also decrease with increasing hops.

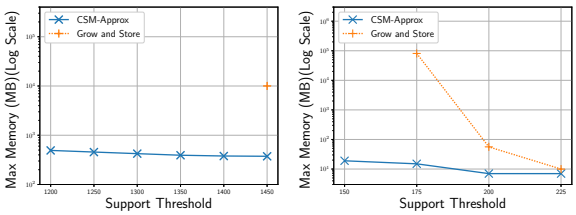
Figure 12 shows results of our approximation algorithm with varying the K parameter. As K increases, it is exppected that more number of patterns will be processed and hence the running time should increase which can be seen in case of $LastFM$. However, it might happen that for the chosen support if ceasing condition is achieved with the condition



(a) Hop-1, K-20 in *LASTFM* (b) Hop-1, K-20 in *Mico*

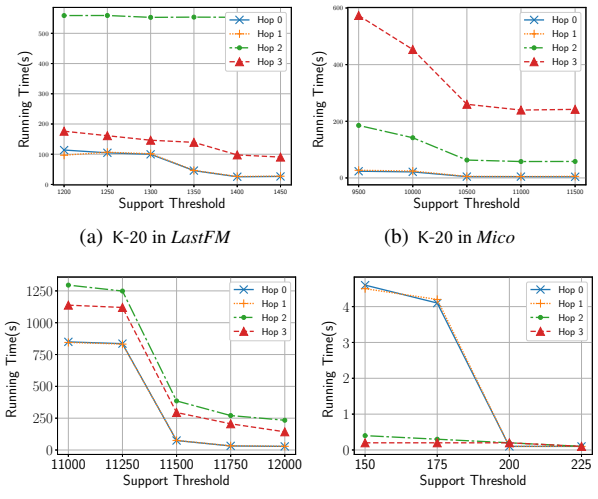
(c) Hop-1, K-20 in *DBLP Coauthor* (d) Hop-1, K-20 in *Citeseer*

Figure 9: Baseline comparisons of datasets *(4)(5)(6)(7)*.



(a) Hop-1, K=20 in *LastFM* (b) Hop-1, K-20 in *Citeseer*

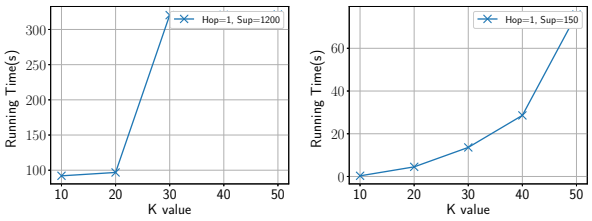
Figure 10: **Memory comparison.**



(a) K-20 in *LastFM* (b) K-20 in *Mico*

(c) K-20 in *DBLP Coauthor* (d) K-20 in *Citeseer*

Figure 11: **Approx algorithm results on datasets *(4)(5)(6)(7)*.**



(a) Hop-1 in *LastFM* (b) Hop-1 in *Citeseer*

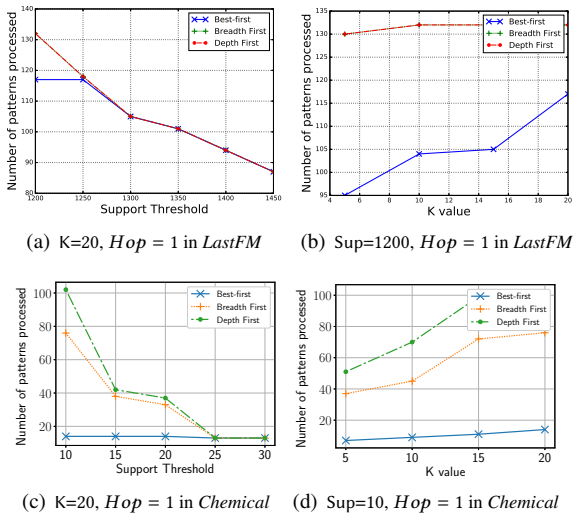
Figure 12: Approx algorithm results with varying k.

of leaf getting empty which means all patterns have been processed and we cannot get more correlated patterns, then the time would stay same with increasing K as can be seen in the case of *DBLPCoauthor*.

However, users may not be interested in some correlation like Figure **??**. Due to its small size and symmetric structure, we can infer that if $h = 1$, the subgraph pattern in Figure **??** is very frequent. This job could already be done by frequent subgraph mining technique. Some users may also not interested in the correlation between just single edges. However, by simply limit the output correlation subgraph size, she can filter those small ones and reach her destination, like Figure **??**.

• **vs. BFS & DFS** We provide the comparison between best-first-search and the other two search strategies, BFS and DFS for the approximate version of out Top-k mining. We count the number of patterns processed in all the three strategies by counting the number of patterns which are popped from the leaf. Depth first search is expected to have the maximum number of patterns processed because if it goes on a branch where all patterns are frequent till a large depth, it will process a large number of patterns. Breadth first search processes all the patterns at a level and so it will also process more patterns than best first search as we can only stop when all the patterns in a level have support less than the least correlation value in top k. However in best first search, we can stop as soon as we find a pattern with support less than the least correlation value in top k as all the other patterns will have a support less than or equal to the current pattern because of the best first strategy. However, if leaf gets completely exhausted in best first strategy, best first will have exactly same number of patterns explored as breadth first and depth first.

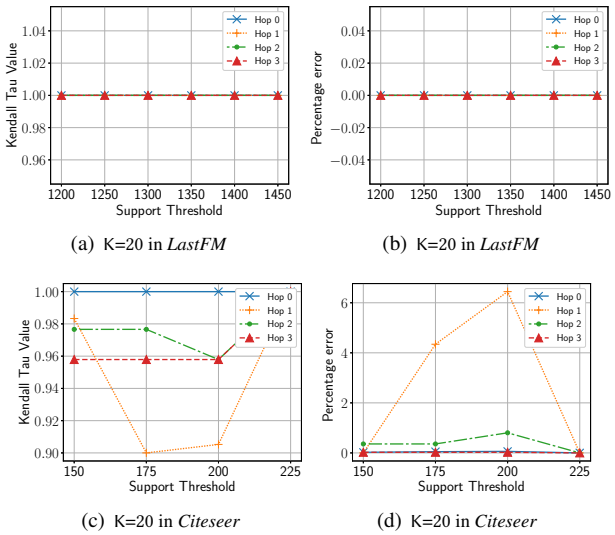
• **Qualitative Analysis** We show the quality of our approximation algorithm by comparing the results of top k of our approximate algorithm with the complete version, both run with a size bound of maximum 5 sized patterns. We take the output of the complete version as the ground truth. The following 3 experiments are done to show quality of our approximate version:



(a) K=20, $Hop = 1$ in *LastFM* (b) Sup=1200, $Hop = 1$ in *LastFM*

(c) K=20, $Hop = 1$ in *Chemical* (d) Sup=10, $Hop = 1$ in *Chemical*

Figure 13: Comparison with BFS and DFS.



(a) K=20 in *LastFM* (b) K=20 in *LastFM*

(c) K=20 in *Citeseer* (d) K=20 in *Citeseer*

Figure 14: Qualitative Analysis.

(1) *Jaccard coefficient* - We compare the patterns that we get in top-K in both the algorithms and report the fraction of pairs that we have in the approximate output which are there in ground truth as well. The fraction comes out to be 1 for all the datasets which means we get the exact same results as the complete version.

(2) *Kendall Tau* - In this experiment, we focus on the ordering of the top K results. We keep $K - 20$ and show results of varied hops for different supports.

(3) *Percentage error in correlation values* - In this ecperiment, we calculate the average error we have in correlation values for all the patterns in $top - K$. We plot this error values at $K - 20$ and varied hops at different supports.

7 RELATED WORK

- **Frequent Itemset Mining.**
- **Frequent Subgraph Mining.**
- **Frequent Subgraph Mining in Single Graph.**
- **Subgraph Isomorphism/Subgraph Querying.**

8 CONCLUSIONS