An Approximate Algorithm for Mining Top-k Correlated Subgraphs in a Large Graph

Thesis submitted by

Arneish Prateek
2014CH10786

under the guidance of

Prof. Sayan Ranu

in partial fulfilment of the requirements
for the award of the degree of

Master of Technology

Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

June 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **An Approximate Algorithm for Mining Top-k Correlated Subgraphs in a Large Graph** submitted by **Arneish Prateek**, to Indian Institute of Technology, Delhi, for the award of the degree of **Master of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. 1
Professor Sayan Ranu
Dept of CSE
IIT-Delhi, 110016

Place: New Delhi

Date: June 25, 2019

ACKNOWLEDGEMENTS

This thesis is the outcome of a joint collaborative effort with **Akshit Goyal** (Master's Candidate, IIT Delhi). The author of this thesis has attempted to demarcate the contributions of the collaborator and himself to the best of his abilities. For a complete guide to this project, this thesis must be read in conjunction with the thesis titled **An Exact Algorithm for Mining Top-k Correlated Subgraphs in a Large Graph** by **Akshit Goyal**. Appendix I contains the complementary sections from this other thesis. The following chapters are common to both theses:

Chapter 1

Chapter 2

Chapter 3: Sections 3.1,3.2,3.3,3.4

Chapter 4: Section 4.1

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.

Contents

Chapter 1

Introduction

Graphs are a ubiquitous model to represent objects and their complex relations, including protein interaction and genomic graphs in biology, chemical compound structures in chemistry, food webs and microbial taxa communities in ecology, call graphs in program flow analysis, social networks in social science, Web graphs, as well as XML documents in e-commerce, government and medical records [**?**]. Efficient mining of hidden patterns play an important role in developing data-driven methodologies for analyzing the graphs resulting from such datasets.

Research on graph mining has emerged as one of the hottest topics in recent years due to its wide applications in a variety of fields including drug discovery [**? ? ?**], molecular activity prediction [**? ?**], predicting disease susceptibility from gene expression data [**? ?**], software bug localization [**?**], clustering and classification of graphs [**? ?**], anomaly detection [**?**], and graph anonymization [**?**]. Several variants of graph pattern mining have been proposed, such as frequent subgraphs [**? ?**], approximate subgraphs [**?**], closed and maximal frequent subgraphs [**? ?**], discriminant and statistically significant subgraphs [**? ? ?**], representative subgraphs [**? ?**], etc. — their scalable algorithms were implemented; besides they were extensively surveyed and compared, e.g., in [**? ? ? ?**].

In spite of tremendous progress being made in the area of graph pattern mining, surprisingly the problem of exploring correlations between subgraphs in a single, large graph has not been investigated in the past. In particular, we define a pair of subgraphs as correlated if they co-occur frequently in proximity within a single graph. Correlated subgraphs are different from frequent subgraphs due to the flexibility in which the constituent subgraphs can be connected within different instances of a correlated pattern. In Figure 1.1, we highlight three regions inside the structure of the chemical compound Taxol, an anti-cancer drug, where CCCH and O occur closely, albeit they are connected in different ways in all three instances. For simplicity, we do not consider the edge types (i.e., single- vs. double-bond) in this example. This figure illustrates that while CCCH and O form a correlated subgraph pair, the individual instances, e.g., HCC(-O)C may not be frequent; and hence, the existing frequent subgraph mining techniques cannot be directly applied to discover such corrected patterns.

The problem of detecting correlated subgraphs from a single, large graph arises in many real-world scenarios, including biological, social, chemical, and ecological networks, such as the ones described next.

• **Co-operative functions in biological networks.** In the genome graph of an individual, a node represents a gene, and each edge denotes the interaction between two genes. In practice, there are

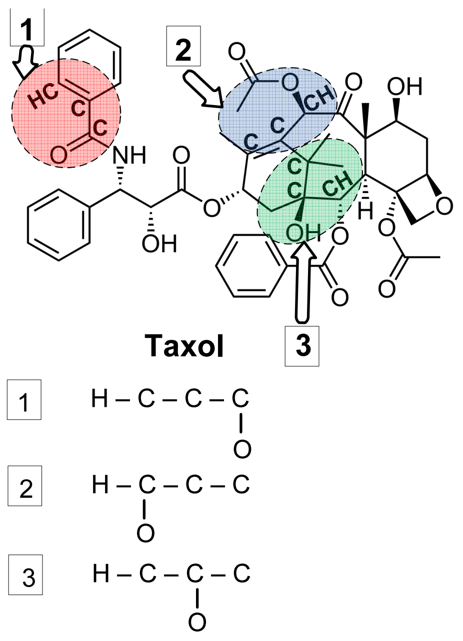


Figure 1.1: Correlation between CCCH and O in Taxol, an anti-cancer drug. CCCH and O occur closely for several times, but they are connected in different ways.

some combinations of dominant genes that occur frequently, and they are more likely to express critical phenotypes of the individual. Past studies have shown that some pairs of dominant gene combinations co-occur with each other in each individual, and such co-occurring patterns reflect the joint functionality that are needed for co-operative biological functions such as chemical bonds and binding sites [**?**]. Based on pairs of correlated genes, we can predict co-operative biological functions of an individual. Besides, the absence of such correlations could indicate anomalies and diseases. Analogously, in a protein interaction (PPI) network, correlated frequent subgraphs represent recurring functional units, identification of which could assist in predicting new protein functionalities.

• **Co-occurrence patterns in ecological networks.** Co-occurrence patterns are used in ecology to explore interactions between organisms and environmental effects on co-existence within biological communities [**?**]. Exploration of inter-taxa relationships can help identify potential biotic interactions, habitat affinities, and shared physoilogies, that could guide more focused studies and experiments. For example, Barberan et al. [**?**] analyzed over 160 000 bacterial and archaeal 16S rRNA gene sequences from 151 soil samples, collected from a wide variety of ecosystem types, in order to demonstrate the utility of network analyses and for evaluating whether soil microorganisms tend to co-occur more than expected by chance, and how these ecological categories shape network structure of inter-taxa and extra-taxa relationships. These tasks can be achieved by mining correlated subgraphs in a single large graph representing the soil microbial community.

Challenges. The problem that we study is a non-trivial one.

- Detecting correlated subgraphs is more difficult than the frequent subgraphs mining problem, which already has an exponential search space, that is, a graph with m edges can have 2^m subgraphs. For correlated subgraphs mining, the search space is doubly-exponential, because one needs to compute the correlation between every pair of subgraphs.

- Additionally, unlike frequent subgraphs mining, correlated subgraphs mining is neither downward-closure, nor upward-closure (we shall demonstrate this formally in Section 2.2), thereby making it difficult to directly apply apriori-based pruning techniques.
- Last, but not least, only finding the frequent subgraphs is not sufficient for our problem. Instead, we require to enumerate all instances of those frequent subgraphs for computing the correlation between pairs. This makes our problem more challenging both from computation and memory perspective.

Our contributions and roadmap. To this end, we design a single-step, best-first exploration algorithm to detect correlated subgraphs, coupled with efficient top-k pruning and various optimization techniques. The main contributions of this thesis are as follows:

- We introduce a scheme for defining and computing the *correlation* metric between isomorphisms of two subgraph patterns
- We introduce a novel data structure called the *Replica* structure that offers a strategy to store all instances overcoming the space limitations of GROWSTORE.
- We propose an *exact* backtracking algorithm that works on the *Replica* structure to mine all ismorphisms of a pattern
- For scalability, we also propose an *approximate* backtracking algorithm that approximately computes the *replica* structure
- We empirically demonstrate effectiveness and efficiency of our methods on real-life graphs, while also detailing three concrete case studies (Section 4).

Chapter 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. 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* $M : V_Q \to V$ 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 2.1. 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 2.1 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 $Q_1 \succeq Q$ 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.

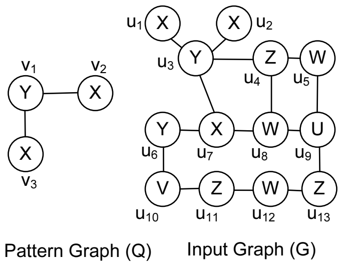


Figure 2.1: Subgraph isomorphism: $M(v_1) = u_3$, $M(v_2) = u_1$, $M(v_3) = u_2$. Two other subgraph-isomorphic mappings could be as follows: $M_1(v_1) = u_3, M_1(v_2) = u_1, M_1(V_3) = u_7$; and $M_2(v_1) = u_3, M_2(v_2) = u_2, M_2(v_3) = u_7$.

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 $\mathbf{NP}$-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,

$$\sigma(Q) = \min_{v \in V_Q} |\{M(v) : M \text{ is a subgraph-isomorphic mapping}\}| \tag{2.1}$$

In Figure 2.1, 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 2.2, 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: $\mathbb{I}' = \{u_1u_2u_7u_3, u_7u_6\}$.*

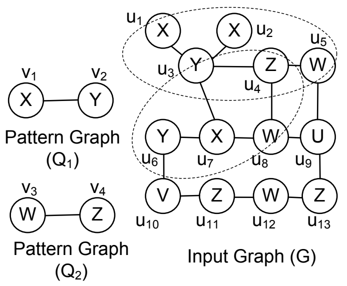


Figure 2.2: Correlation between subgraphs Q_1 and Q_2 in G

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.

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:*

$$\begin{aligned} &\tau(Q_1, Q_2, h) \\ &= |\{I' \in \mathbb{I}' : \exists J' \in \mathbb{J}', \exists u \in I', \exists v \in J', d(u, v) \leq h\}| \end{aligned} \tag{2.2}$$

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 2.2), 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.

Chapter 3

Approximate Algorithm

3.1 Overview

Observation 1. *A pair of subgraphs having higher support values can be expected to 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.

3.1.1 Search Tree

All operations including correlation computation and subgraph pattern extension are performed on patterns following an exploration order on a *search tree*. We denote this tree by T. Each node $Q \in T$ represents a subgraph pattern. We denote the set of current *leaf* nodes in T by $Leaf(T)$. At every stage, a pattern $Q \in Leaf(T)$ that has not been *operated* for correlation computation is selected. Once operated, Q is inserted into the *operated* set.

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

1) Select a node $Q \in Leaf(T)$ that has not been operated

2) Calculate $\tau(Q, Q_j, h)$, $\forall$Q$_j \in$ *operated*

3) Extend Q using possible single-edged extensions to generate the set of "children" supergraphs of Q, *children(Q)*

4) Compute $\sigma(R),\ \forall R \in children(Q)$. If $\sigma(R) \geq$ Min-Sup, branch T at Q to include R in $Leaf(T)$.

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

3.1.2 Best-first Search

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 unoperated pattern in $Leaf(T)$. As a result, we use a heuristic to determine the priority of the leaf nodes in $Leaf(T)$ for processing: Q_1 has a higher priority than Q_2 $\forall \mathrm{Q}_1, Q_2 \in Leaf(T)$ if and only if:

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

Utilizing such a best-first heuristic, if leaf node Q_k in the search tree has the highest support among all other patterns in $Leaf(T)$, i.e. $\sigma(Q_k) = \max\{\sigma(Q_i) | Q_i \in Leaf(T),\, Q_i \textit{ not operated}\}$, then Q_k has the priorty to be operated and extended.

To implement best-first exploration, we introduce a priority queue called *Search Queue* that simply queues, at any given time, all unoperated patterns in $Leaf(T)$ with the priority being accorded to patterns with higher frequencies. Also, note that as described in step 4 of the algorithm outline in Section 3.1.1, we do not consider infrequent patterns in $Leaf(T)$, therefore, *Search Queue* only orders frequent patterns.

3.1.3 Best-first Termination Criteria

Our objective is to mine k pairs of correlated subgraphs and guarantee that any other pair of subgraphs cannot have a higher correlation (τ) than any pair in such a Top-k list. A closer look at the properties mentioned in Section 2.3 allows us to deduce the following: assume Q and Q_j are two arbitrary frequent subgraphs of the data graph. We denote $super(Q)$ as the set of all possible supergraphs of Q. Then, for all $Q' \in super(Q)$ the following conditions always hold:

$$\tau(Q', Q_j, h) \leq \sigma(Q') \quad (\text{Corollary to Lemma 3})$$

$$\sigma(Q') \leq \sigma(Q) \qquad (\text{downward-closure})$$

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

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

Consider this upperbound: if $\sigma(Q)$ is lower than the least τ value in the Top-k list, i.e. $\sigma(Q) < \tau(Q_a, Q_b, h)$ for all pairs $Q_a,\ Q_b$ among the Top-k, then all correlations containing Q $(i.e.\ \tau(Q, Q_k, h))$ cannot possibly be values in the Top-k list. Furthermore, all correlation values of every pattern R with support values such that $\sigma(R) \leq \sigma(Q)$ would also be ineligible for the list. Thus, in a

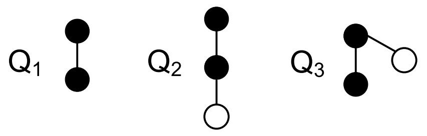


Figure 3.1: For subgraphs Q_1, Q_2, Q_3, with $\sigma(Q_1) = 5$, $\sigma(Q_2) = 2$, $\sigma(Q_3) = 3$. The current Top-k τ values are $\{4, 5, 6\}$ for $k = 3$, Min-sup$= 3$

.

best-first exploration scheme, occurrence of such a case would signal termination of the search.

Another possible termination scenario exists wherein k correlated pairs do not even exist in the data graph. In this case, the size of the Top-k list would be less than k but there wouldn't exist any frequent subgraph left in *Search Queue* to operate.

Formally, we specify the two ceasing conditions for terminating search based on a best-first heuristic:

$$(1)\ \sigma(Q) \leq min\{\tau(Q_a, Q_b, h) \mid (Q_a, Q_b) \in top\_k\},\ |top\_k| = k$$

$$(2)\ Search\ Queue = \emptyset$$

In both these cases, we cease search and report all pairs of correlated subgraphs discovered along with τ values.

Example 3. *In Figure 3.1, initially Q_1 is the only node in* Search Queue *and $\sigma(Q_1) > 4$. Suppose after τ calculations for Q_1, the minimum τ value among the* top-3 *pairs is $\tau_{min} = 4$. After this, Q_1 is extended to Q_2 and Q_3. After extension, assume $\sigma(Q_2) <$*Min-sup *and $\sigma(Q_3) =$* Min-sup*. Thus,* Search Queue *now only consists of Q_3 since Q_2 is not frequent. But since $\sigma(Q_3) < \tau_{min}$ and there are already 3 pairs in* top-k*, i.e. $|top\_k| = 3$ any correlation of Q_3 or Q_2 would not displace existing* top-k *pairs. Thus, we cease search.*

3.2 Replica-based Graph Instance Storage

As discussed in Section 2.2, computing σ and τ values requires knowledge of a pattern's *instances* (or *isomorphisms*) in the data graph. In a dense graph, the number of instances of a pattern can be prohibitively large to store for each pattern so methods like GROWSTORE have limited scope. In this section, we describe an elegant approach for performing subgraph isomorphism using the **Replica Structure**.

3.2.1 Replica Structure

Considering the large amount of overlaps of instances in dense graphs, we use a novel yet simple data structure to "store" all instances of a subgraph pattern, which not only solves potential memory issues that arise while tackling dense graphs, but also lays a robust foundation for carrying out efficient correlation calculations based on instance grouping (Section 2.2).

Instead of storing every instance of a pattern, we just create a reproduce of all occurrences of the vertices and the edges of the pattern in the data graph. We call the structure a *replica* graph. We record all the vertex identifications and the edge connections in the *replica* graph. In other words, a *replica* graph is essentially a "minimal" subgraph *G'* of data graph *G* such that every instance of a pattern Q in G and every such instance only can be found in G'.

Figures 3.2a and 3.2b illustrate a pattern and its corresponding *replica* graph. A *replica structure* consists of not just the *replica graph*, but also two maps, as defined below.

Replica Structure. Consists of the following three structures constructed for each pattern $Q \in T$ for a data graph G:

1) *Replica Graph*: occurrence graph denoted by *replica(Q)*

2) *Mappings*: $\forall u \in V(Q),\ Mappings(u, replica(Q))$ stores the set of all vertex identifications $v \in V(replica(Q))$ of u

3) *Inverse Mappings*: $\forall v \in V(replica(Q)),\ InverseMappings\ (v, Q)$ stores the set of all $u \in V(Q)$ such that $v \in Mappings(u,\, replica(Q))$

Both *Mappings* and *Inverse Mappings* can be deduced from the *replica* graph, however we still maintain these indexes for computational gains.

3.2.2 Generation of a Replica Graph

Algorithms 1 and 2 together describe the complete procedure to construct the *replica* structure for a subgraph pattern, henceforth referred to as simply *replica*. *Replica* construction for any pattern R requires knowledge of the *replica* 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* of the parent pattern can be used for constructing the *replica* of the child.

Algorithm 1 essentially describes a backtracking procedure to find every instance (also referred to as *isomorphism* or *mapping*) of the child pattern in the data graph G using the mappings of the parent pattern and thus obtain its *replica*. The algorithm begins with a depth-first search (DFS) procedure (line 1) executed on the parent Q, selecting the vertex u from which the *candidate edge*

Algorithm 1: GETREPLICA

```
Input: Graph $G$, parent $Q$, $replica(Q)$, child $R$, extending vertex: $u \in V(Q)$, 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      $instance \leftarrow \{(u, u')\}$
5      foreach edge $e(u', v') \in E(G)$ that maps to $candidate\ edge(u, v)$ do
6          $instance \leftarrow instance \cup \{(v, v')\}$
7          $\mathbb{I} \leftarrow$ FINDALLINSTANCES($R$, $instance$, $\mathbb{I}$, $DFS\ List$, ...)
8          UPDATEREPLICA($replica(R)$, $\mathbb{I}$, ...)
9          $instance \leftarrow instance \setminus \{(v, v')\}$
10     $instance \leftarrow instance \setminus \{(u, u')\}$
11 return $replica(R)$
```

is extended as the *root*. We call this vertex the *extending vertex* in Q. The *edges* encountered in the DFS starting at u are recorded in an ordered list called the *DFS List*, which helps guide the isomorphism procedure performed subsequently. The algorithm iterates over all mappings of u in *replica(Q)* (line 3) and attempts to find all (if any) instances of child R in G one-by-one. More specifically, for every vertex $m \in V(replica(Q))$ that maps to the *extending vertex* u, the algorithm iterates over its adjacent edges $e \in E(G)$ that map to the *candidate edge* (line 5) and invokes FINDALLINSTANCESAPPROX (Algorithm 2) to find every instance of R from *replica(Q)* that contains e.

Exact *replica* construction (Appendix I) 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 (Appendix I), 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 A.1 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 2 describes an approximate method 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 2 is a recursive algorithm. In the general (recursive) case (lines 5-30), 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 $w \in P_{v,c}$.

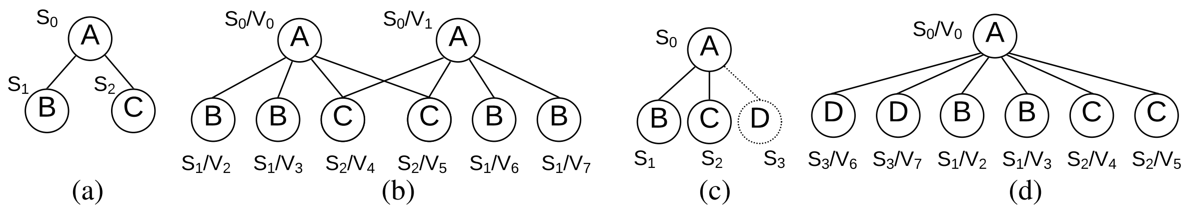


Figure 3.2: (a) parent Q (b) $replica(Q)$ (c) child R showing edge extension (S_0, S_3) (d) $replica(R)$

(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 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.

This replica-based instance 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(R)* just by counting all the sizes of the $Mappings(v)$, where $v \in V(Q)$.

Example 4. *In Figure 3.2, originally* Q *and* replica(Q) *are as shown in 3.2a and 3.2b respectively. Child* R*, extended from* Q *at* extending vertex S_0 *using the* candidate edge (A, D) *is as shown in 3.2c. Assume* DFS List *starting at* S_0 *records edges* (S_0, S_1) *and* (S_0, S_2) *in that order. To construct* replica(R)*, the algorithm jumps to* $Mappings(S_0)$ *in* replica(Q)*, i.e.* V_0 *followed by* V_1*. Assume,* $(V_0, V_6), (V_0, V_7) \in E(G)$ *map to* (A, D) *but no such mapping edges exist incident to* V_1 *in G. All instances of R thus found would result in* replica(R) *as shown in Fig.3.2d*

Algorithm 2: 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 $(u',v')$
   $Base\ Case$ :
1  if $|instance| = |V(R)|$ then
2      Update $E_{l'}$ for each $l' \in Leaves(Q)$
       $[[E_{l'} \leftarrow E_{l'} \cup \{instance(l')\}, \forall l' \in Leaves(Q)]]$
3      return $instance$
4  $Recursive\ Case$ :
5  $e(p,c)$ :− NEXTQUERYEDGE($DFS\ List$, ...)
   $[[\ (p,c) \in E(Q),\ c \in children(p)]]$
6  $v$ :− Mapping of $p$ in $instance$
   $[[\ v \in V(replica(Q)) \wedge (p,v) \in instance]]$
7  if $P_{v,c}$ does not exist then
8      $P_{v,c}$ :− FILTERCANDIDATES($instance$, $v$, $c$, ...)
       $[[\forall w \in P_{v,c}\ ((w \in V(replica(Q)) \wedge (w \in adjList(v)) \wedge (c \in$
       $InverseMap(w)) \wedge (L(v,w) = L(p,c)))]]$
9      $C_{v,c}$ :− $\emptyset$
10 $boolean\ InstanceFound$ :− False
11 foreach $w \in P_{v,c}$ such that $w$ is not yet matched do
12     $instance \leftarrow instance \cup \{(c,w)\}$
13     $\mathbb{I}' \leftarrow$ FINDALLINSTANCESAPPROX($R$, $e$, $instance$, $DFS\ List$)
14     if $\mathbb{I}' \neq \emptyset$ then
15         $\mathbb{I} \leftarrow \mathbb{I} \cup \mathbb{I}'$
16         $InstanceFound \leftarrow$ True
17     if $w \in E_c$ then
18         $P_{v,c} \leftarrow P_{v,c} \setminus \{w\}$
19         $C_{v,c} \leftarrow C_{v,c} \cup \{w\}$
20     $instance \leftarrow instance \setminus \{(c,w)\}$
21 foreach $w' \in C_{v,c}$ such that $w'$ is not yet matched and $InstanceFound$ is False do
22     $instance \leftarrow instance \cup \{(c,w')\}$
23     $\mathbb{I}' \leftarrow$ FINDALLINSTANCESAPPROX($R$, $e$, $instance$, $DFS\ List$)
24     if $\mathbb{I}' \neq \emptyset$ then
25         $\mathbb{I} \leftarrow \mathbb{I} \cup \mathbb{I}'$
26         $InstanceFound \leftarrow$ True
27     $instance \leftarrow instance \setminus \{(c,w')\}$
28 if $\forall c' \in children(p),\ P_{v,c} = \emptyset$ then
29     $E_p \leftarrow E_p \cup v$
30 return $\mathbb{I}$
```

3.3 Subgraph Extension

3.3.1 Extension Rule

We assign pattern vertex indices for all $V_i \in V(Q)$ to identify their order of discovery.

Definition 3 (Vertices Subscripting). *For all $Q \in T$, apart from vertex identification V_i, we use another convention to label the vertices in Q by the* **order** *of discovery leading to pattern Q, denoted as $S_Q = (S_0, S_1, ..., S_n)$, $n = |V_Q|$. Thus, $\forall S_i, S_j$, if $i < j$, then the vertex V_i is discovered earlier than the vertex V_j.*

Taking advantage of this convention, it is easy to get the right-most path of a subgraph. Q is then extended only from vertices along the right-most path in single-edged extensions to generate $children(Q)$ which continue the search. Algorithm 3 captures this procedure.

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      foreach $v' \in Mappings(v, replica(Q))$ do
5          $E \leftarrow$ set of all edges $(v', w') \in E(G)$ extending $Q$
6          $Ex(Q) \leftarrow Ex(Q) \cup E$
7  return $Ex(Q)$
```

SUBGRAPHEDGEEXTENSIONS iterates over every mapping $v' \in V(replica(Q))$ for every $v \in rmpath$(Q) and attempts to extend Q using a single-edge extension. The set of all valid candidate edge extensions, $Ex(Q)$ thus obtained is returned.

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 constructed we compute its minimum DFS code, denoted as $DFS\ Code(Q)$.

We use a dictionary $\mathbb{D}$ to store all the minimum $DFS\ codes$ we have discover. When a subgraph Q is discovered, we perform a lookup for $DFS\ Code(Q)$ in the dictionary. If $DFS\ Code(Q) \in \mathbb{D}$, 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.*

$$\exists J' \in \mathbb{J}, \exists u \in I', \exists v \in J', d(u,v) \leq h \tag{3.1}$$

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 $Q_k \in Cor(T)$.

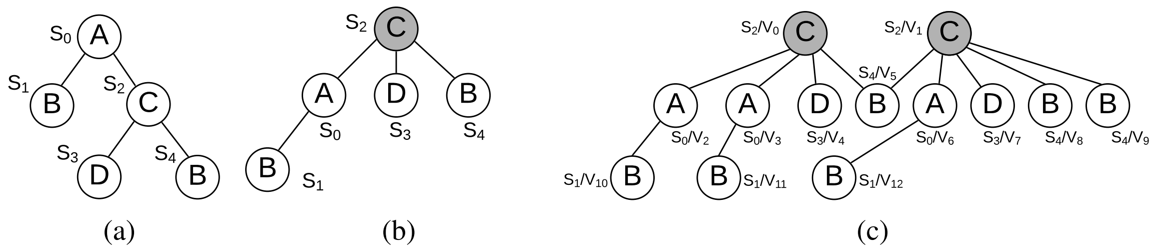


Figure 3.3: Figure 3.3a is the subgraph pattern of Q_1, Figure 3.3c is a replica of the occurrence of Q_1 in Figure 3.3a without any backward edges. Figure 3.3b is the same subgraph pattern Q_2 with Figure 3.3a, with collection tree rooted at the group center s_2. Figure 3.3c is the occurrence of Q_2 in data graph G.

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.

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 $centerSubscript(Q)$.*

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 $proximity\ patterns\ map$ ($CorP(v)$).

$$Collect(c, Q) = \cup\{CorP(v) | v \in instancegroup\} \tag{3.2}$$

• **Counting 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

$$u = groupCenter(I_i'), Q_2 \in Collect(u, Q_1) \tag{3.3}$$

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

$$\tau(Q_1, Q_2, h) = \sum_{i}^{\sigma(Q_1)} P(I_i', Q_2, h) \tag{3.4}$$

Algorithm 4 summarises the steps described above.

```
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 ($\tau$) values
9 operated $\leftarrow$ operated $\cup \{Q\}$
```

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.

Example 5. *In Figure 3.4, 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.*

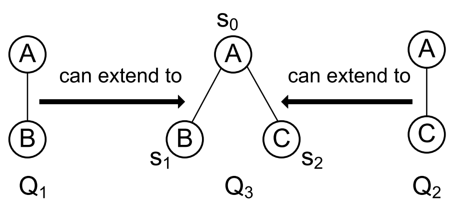


Figure 3.4: Subgraph Q_1 and Q_2 are the subgraphs of Q_3.

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 $\mathsf{Op}(Q)$ 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 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

```
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 $F\_edges,\ CorV,\ G$
```

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($Q', u, e, \ldots$)
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 *Search 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 3.1.2). *Search 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 3.1.3) remains unsatisfied, the subgraph pattern at the front of *Search 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 SUBGRAPHEDGEEXTENSIONS (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 GETREPLICA 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 *Search Queue* (line 12) to be (possibly) processed in a later iteration of the outer loop. *DFS Code(*Q'*)* is recorded in $\mathbb{D}$ (line 13).

Chapter 4

Experiments

In this section, we benchmark the proposed algorithms (CSM-Exact and CSM-Approximate, denoted as CSM-E and CSM-A respectively) and establish that:

- The pruning strategies employed in CSM are effective in making it scale on million-sized datasets. Furthermore, CSM-A is a 10 orders of magnitude faster than baseline approaches built using the state-of-the-art algorithms.
- CSM-A is consistently near-optimal in quality across all datasets.
- CSM unearths useful patterns that would not be discovered using existing techniques such as frequent subgraphs mining.

4.1 Experimental Setup

All of our algorithms have been implemented in C++ and compiled using gcc 7.4.0. All experiments are conducted on a Linux(Ubuntu 18.04) 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 the baseline algorithms.

4.1.1 Datasets

We use the seven real datasets, summarized in Table 4.1, to benchmark CSM.

•*Chemical*[]. This graph dataset represents the structure of a chemical compound in the MCF7 Dataset of X.F. Yan[**?**].Each node represents a chemical atom and two atoms are connected by an edge if they share a chemical bond.

•*Citeseer*[]. Each vertex is a publication and its label categorizes the area of research. 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.

•*Yeast*[]. This dataset contains the protein-protein interaction network in Yeast. The node labels denote their functional category.

•*Mico*[]. Mico models Microsoft co-authorship information. Each vertex is an author and the label is the field of interest. Edges represent collaboration between two authors and the edge label is the number of coauthored papers.

Table 4.1: Datasets and characteristics

Datasets	Nodes	Edges	Domain
Chemical	207	205	Biological
Yeast	4K	79K	Biological
MiCo	100K	1M	Collaboration
LastFM	1.1M	5.2M	Social Network
DBLP Coauthor	1.7M	7.4M	Collaboration
DBLP Citation	3.2M	5.1M	Collaboration
Citeseer	3K	4.5K	Collaboration

•*LastFM*[]. This dataset represents the FM social network where a node label represents the most frequent singer or music band that the corresponding user listens to.

•*DBLP coauthor*[]. is a co-authorship network in which two authors (nodes) have an edge between them if they have collaborated on at least one paper together. The label of a node is the conference in which that author has most published the most.

•*DBLP citation*[]. Each vertex is a publication and the 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.

4.1.2 Parameters

There are three different input parameters to our problem: MNI 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 each of these input parameters. When not specifically mentioned, the default values of k and h are set to 20 and 1 respectively.

4.1.3 Baselines

We compare CSM-E and CSM-A with two baselines.

• **GrowStore:** GROWSTORE employs the same algorithm as CSM-E except one critical difference; instead of using the replica to summarize all instances, we store each of the instances individually. More simply, comparing with GrowStore allows us to precisely quantify the benefit obtained due to using the replica data structure.

• **GraMi-VF3:** In this approach, we first use GRAMI[] to mine frequent subgraphs. Next, we employ the VF3[] to enumerate all instances of the frequent subgraphs and compute the correlation among them to compute the top-k answer set. Both GRAMI and VF3 are the state-of-the-art algorithms for the specific tasks of mining frequent subgraphs and subgraph enumeration respectively.

4.2 Efficiency

In this section, we compare the efficiency of CSM-E and CSM-A against the baseline techniques. Since the baseline algorithms are exorbitantly slow, we restrict the mining process to only patterns containing at most 5 vertices. As the pattern size increases, the cost of enumerating their instances increases exponentially, since it involves performing subgraph isomorphism tests.

4.2.1 *Running time:*

Figure 4.1 presents the running times of the benchmarked algorithms across different datasets against the support threshold. As expected, CSM-A is the fastest algorithm followed by CSM-E, and then GrowStore. Note that we do not show the running time of GraMi-VF3 since it fails to terminate even after 7 days.

Several key insights can be derived from Figure 4.1. First, as evident from GraMi-VF3, using frequent subgraph mining to solve the proposed problem is not scalable. Second, the slow running time of GrowStore shows that the replica data structure is effective in reducing the computation cost. Finally, CSM-A is 100 times faster than CSM-E on average, with the difference being more significant on smaller values of support threshold. This is expected since as the support threshold is lowered, the search space increases exponentially, and the impact of avoiding repetitive computation magnifies.

4.2.2 *Memory footprint.*

Our goal in this experiment is to understand the reduction obtained in memory footprint due to the replica data structure. Replica provides an efficient mechanism to store all instances of a subgraph pattern in a compact manner. In contrast, GROWSTORE stores all instances separately. This is a significant bottleneck since, in the worst case, the number of instances of a subgraph could be exponential. In figure 4.2, we compare the memory footprint of CSM-A with GROWSTORE. On average, CSM-A consumes more than 100 times lower memory than GrowStore. This stark difference in the memory consumption of CSM-A and GrowStore clearly highlights that replica is not only critical to reduce running time, but also the memory footprint. Note that CSM-E has the same memory requirements as CSM-A and hence we do not plot CSM-E in figure 4.2. In the plot of LASTFM, we have just one point for GROWSTORE because its memory consumption exceeds $256GB$ for lower supports. The same happened in the lowest support threshold for a small dataset like Citeseer as well.

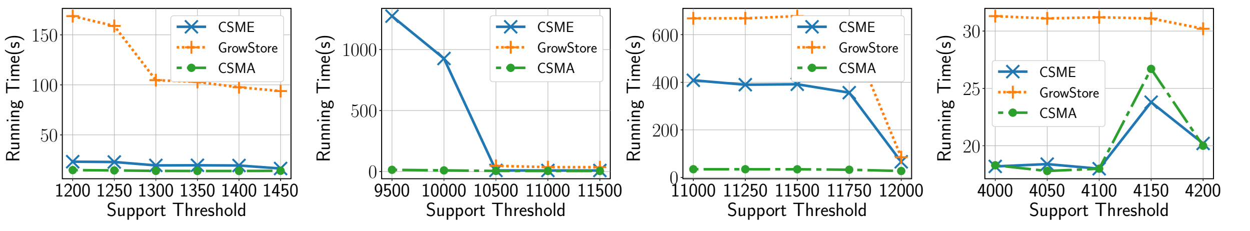

(a) Hop-1, K-20 in *LASTFM* (b) Hop-1, K-20 in *Mico* (c) Hop-1, K-20 in *DBLP Coauthor* (d) Hop-1, K-20 in *DBLP Citation*

Figure 4.1: Baseline Running Time Comparisons.

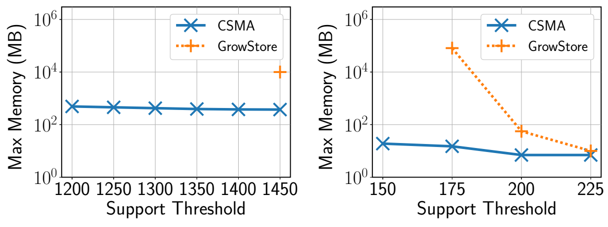

(a) Hop-1, K=20 in *LastFM* (b) Hop-1, K-20 in *Citeseer*

Figure 4.2: Baseline Memory Comparison.

4.3 Performance.

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

Running Time Comparison. Figure 4.3 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 4.4 shows results of our approximation algorithm with varying the K parameter. As K increases, it is expected 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

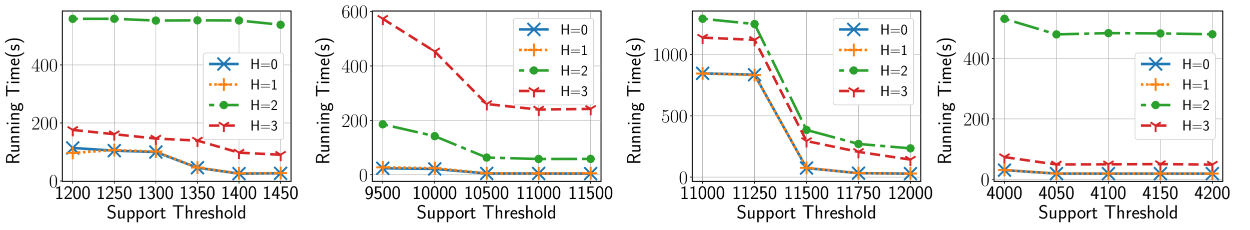
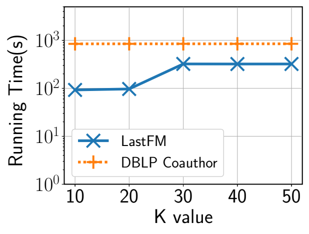

(a) K-20 in *LastFM* (b) K-20 in *Mico* (c) K-20 in *DBLP Coauthor* (d) K-20 in *DBLP Citation*

Figure 4.3: CSMA Running Time Results.



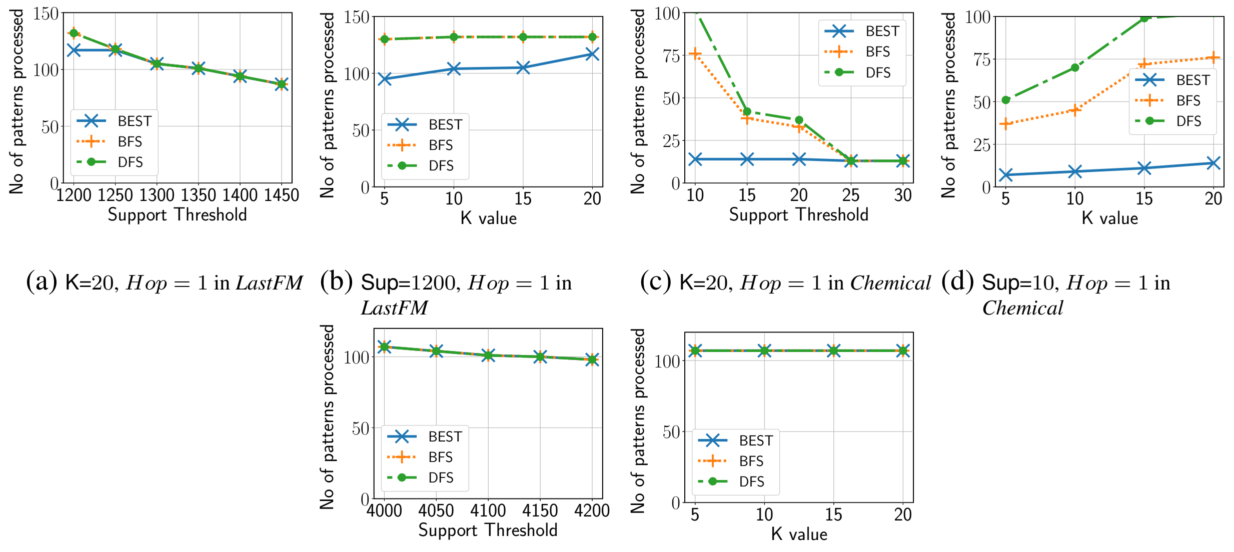
(a) Hop-1 in *LastFM*

Figure 4.4: CSMA results with varying k, LASTFM σ=1200, DBLP Coauthor σ=11000.

a chosen support, the program is terminated after all the patterns are processed and there is no pattern remaining in the Search Queue which means we cannot get more correlated patterns if we increase k. The time in such cases would stay the same after a certain K as can be seen in the case of $DBLPCoauthor$.

Comparison with other search strategies We provide the comparison of our $Best\ First\ Search$ strategy with other search strategies, $Breadth\ First\ Search(BFS)$ and $Depth\ First\ Search(DFS)$. We count the number of patterns processed in all the three strategies by counting the number of patterns which are popped from the Search Queue. Figure 4.5 shows the results of these experiments on two datasets by varying $min - sup$ and K.

In DFS, if we go to a branch where all patterns are frequent till a greater depth, it will lead to a lot of patterns being processed. BFS processes all the patterns at a level as we can only stop processing patterns when all the patterns at a given depth have support less than the least correlation value in top k. However in best first search, there is no such restriction and 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. In addition, the problem setting is such that patterns with higher frequency tend to have a higher correlation with other patterns and the best first strategy takes full advantage of this. However, if program gets terminated after all the patterns are processed and there are no patterns remaining in the Search Queue, best first will have exactly same number of patterns explored as BFS and DFS.



(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*

(e) K=20, $Hop = 1$ in *DBLP Citation* (f) Sup=10, $Hop = 1$ in *DBLP Citation*

Figure 4.5: Comparison with BFS and DFS.

4.4 Quality

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

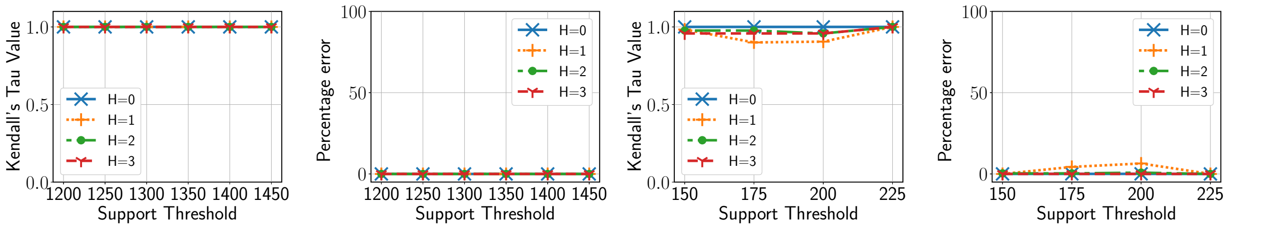
Jaccard Coefficient. We compare the patterns that we get in top-K in both the algorithms and report the fraction of pairs that we have in the approximate output which are there in the ground truth as well. The fraction comes out to be 1 for all the datasets which means we get the exact same results as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

Kendall's Tau. This is a measure of relationships between ranked data and focusses on the ordering of the results. It gives a value between 0-1 where 0 is no relationship and 1 is perfect relationship. It is calculated as

$$KT = \frac{C - D}{C + D}, C = |concordant\ pairs|, D = |disconcordant\ pairs|$$

Concordant pairs are how many larger ranks are below a certain rank and disconcordant pairs are how many smaller ranks are below a certain rank. We keep $K - 20$ and show results of varied $hops$ at different supports for $LastFM$ in Figure 4.6a.

Percentage error in τ values. This is a measure of the error in τ values introduced by the approximate version. We calculate the difference in the τ values for all the $top - K$ patterns common



(a) K=20 in *LastFM* (b) K=20 in *LastFM* (c) K=20 in *Citeseer* (d) K=20 in *Citeseer*

Figure 4.6: Qualitative Analysis.

Table 4.2: Quality metrics

Datasets	Jaccard Coeff	Kendall's Tau	Percentage Error
Chemical	1.0	1.0	0
Yeast	1.0	0.75	0
MiCo	1.0	1.0	0
LastFM	1.0	1.0	0
DBLP Coauthor	1.0	0.98	2.26
DBLP Citation	1.0	1.0	0
Citeseer	1.0	0.98	0

in the approximate and the exact set and report the mean error(in percentage). Figure 4.6b shows this error at $K - 20$ and varied *hops* at different supports for $LastFM$.

We report quality metrics for the datasets at $K - 20$, $Hop - 1$ in Table 4.2. The support for every dataset is taken as the minimum support for the support ranges chosen for that dataset for the experiments.

Chapter 5

Conclusions

In this thesis, we presented a novel approach to compute *correlations* between two subgraphs in a single large graph, where we defined *correlations* on the basis of a distance metric. We introduced a novel method to group instances and defined the *correlation value* (τ) as the count of such *instance groups* for a pattern in proximity to those of another pattern. For computing all instances, we proposed a TREESEARCH backtracking algorithm. The algorithm constructs what we call the *Replica* structure - an occurrence graph of a pattern in a data graph. However in a dense or large graph, computing the *replica* is computationally intensive since subgraph isomorphism is an *NP-hard* problem and the number of instances of a pattern generally increases exponentially with increasing graph density. To solve this problem, we proposed an **Approximation** scheme that constructs the *Replica* approximately. We observed good performance of the Approximation algorithm on large datasets, as reported by the *Kendall-Tau's coefficient*, *Jaccard coefficient* and *Percentage error in τ values* - which were close to perfect values for almost all datasets we tested on.

Appendix A

The Exact Algorithm for Correlated Subgraph Mining

The following algorithm is reported verbatim from the thesis titled **An Exact Algorithm for Mining Top-k Correlated Subgraphs in a Large Graph** by **Akshit Goyal**. For a detailed analysis of the exact strategy, the reader can refer to Akshit Goyal's thesis.

```
A.1: FINDALLINSTANCESEXACT()
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      return $instance$
3  else
4      $e(p,c)$ :− NEXTQUERYEDGE($DFS\ List, ...$)
5      $P_c$ :− FILTERCANDIDATES($instance, c, ...$)
6      foreach $w \in P_c$ such that w is not yet matched do
7          $instance \leftarrow instance \cup \{(c,w)\}$
8          $\mathbb{I} \leftarrow \mathbb{I} \cup$ FINDALLINSTANCES($R, instance,$ ...)
9          $instance \leftarrow instance \setminus \{(c,w)\}$
10     return $\mathbb{I}$
```

```
A.2: OPERATEEXACT() ;
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 $(center, m) \in I$
       [[Found Using FINDALLINSTANCESEXACT]]
3      foreach $u \in V(replica(Q))$ constituting 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 ($\tau$) values
9  operated $\leftarrow$ operated $\cup \{Q\}$
```