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

Thesis submitted by

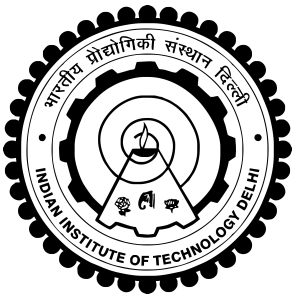
Arneish Prateek
2014CH10786

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* (CSM), 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 is memory intensive and computationally demanding.

To this end, we design two novel, holistic, best-first exploration algorithms: CSM-E (an exact method) and CSM-A (a more efficient, approximate method with near-optimal quality). To further improve their efficiency, we propose a top-k pruning strategy, while to reduce memory footprints, we develop a compressed data structure, $\mathcal{R}$eplica, that stores all instances of a subgraph pattern on demand. Our empirical results show that the proposed algorithms not only find interesting correlations, but also achieve good scalability over large networks.

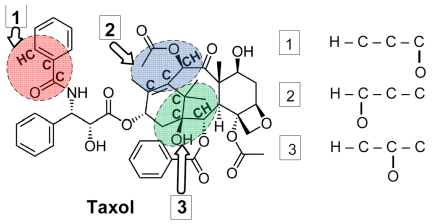


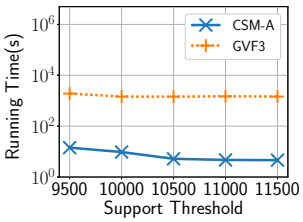
Figure 1: Correlation between CCCH and O in Taxol, an anti-cancer drug. CCCH and O occur closely for several times, but they are connected in different ways.

Contents

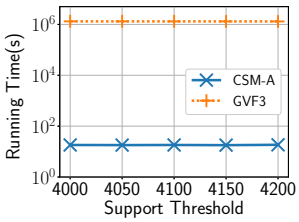
0.1 Introduction

Graphs are a ubiquitous model to represent objects and their complex relations, including protein interaction networks and genomic graphs in biology **?**, chemical compounds **?**, call graphs in program flow analysis **?**, social networks **?**, and government and medical records **?**. Efficient mining of hidden patterns play an important role in developing data-driven methodologies for analyzing 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 **??**, clustering and classification of graphs **??**, and anomaly detection **?**. Several variants of graph pattern mining have been proposed, such as frequent subgraphs **??**, approximate subgraphs **?**, discriminant and statistically significant subgraphs **???**, representative subgraphs **??**, etc.

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. To elaborate, in Figure **??**, 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, existing frequent subgraphs mining techniques would not discover such corrected patterns.



(a) *Mico* dataset (small)



(b) *DBLP Citation* dataset (large)

Figure 2: Our performance against GraMi+VF3 baseline

0.1.1 Applications

The problem of detecting correlated subgraphs from a single, large graph arises in many real-world scenarios, including biological, social, chemical, and ecological networks. We discuss two concrete examples below.

• **Co-operative functions in biological networks.** In genome graph of an individual, a node represents a gene, and each edge denotes the interaction between two genes. In practice, there are 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 indicates anomalies and diseases. Analogously, in a protein interaction network, correlated frequent subgraphs represent recurring functional units, identification of which could assist in predicting new protein functionalities.

• **Co-occurrence patterns in ecological networks.** In ecology, co-occurrence patterns are used 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.

0.1.2 Technical Challenges and Baselines

Frequent subgraphs mining. 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 subgraph instances. Additionally, unlike frequent subgraphs mining, correlated subgraphs mining is neither downward-closure, nor upward-closure (we shall demonstrate this formally in § **??**), 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. Since we call subgraph A to be correlated to B if their *instances* are frequently located close to each other, we need 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. To establish this point empirically, we perform frequent subgraphs mining using GraMi **?**, and for each frequent subgraph, we enumerate all of its instances using the state-of-the-art VF3 algorithm **?** and finally compute the correlated pairs. Fig. **??** presents the result on two datasets; the dataset description is provided in Table **??**. We observe that the frequent subgraphs mining based approach takes more than 15 days in the DBLP-Citation network dataset. On the other hand, the proposed approach, CSM, is up to 5 orders of magnitude faster. Real networks contain millions of nodes and it is desirable to obtain results within minutes. In this paper, we achieve this task with a single-step correlated subgraphs mining algorithm.

Correlated subgraphs mining in graph databases. The closest work to ours in the space of correlated subgraphs mining is by Ye et al.**?**. However, there are three fundamental differences. First, Ye et al. target the graph database scenario where there are multiple graphs and the frequency of a subgraph is defined as the number of database graphs containing the given subgraph. In our problem, we target the single large graph scenario where the frequency of a subgraph is the number instances within the single large graph. It is well known from the frequent subgraphs mining literature **??**, that the single-large-graph scenario imposes a more severe scalability challenge since subgraph enumeration is more expensive. Second, the definition of correlated patterns is different. More specifically, in **?**, two subgraphs A and B are correlated if the containment of A within a data graph increases the likelihood of containing B as well. In our problem, two subgraphs A and B are correlated if the instances of A are frequently located in *close proximity* to the instances of B. Third, we have the concept of proximity in our problem, which is absent in **?**. Owing to these fundamental differences in the formulation, the resulting algorithmic challenges are different as well.

0.1.3 Contributions and Roadmap

The main contributions of this paper are as follows:

- We formulate the problem of *correlated subgraphs mining* in a single large graph, which is defined as a pair of subgraph patterns that frequently co-occur in proximity within a single graph (§ **??**).

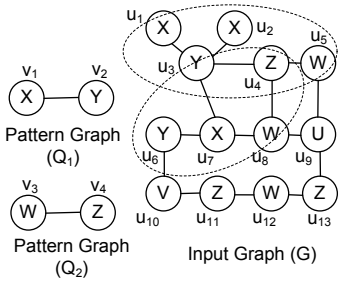


Figure 3: Subgraph isomorphism from Q to G. Correlation between Q_1 and Q_2 in G.

- The key differentiating factor in our problem compared to existing subgraph mining problems is that we not only need to identify the subgraph patterns, but also enumerate and store all of its instances. This requirement imposes a huge scalability challenge on both computation and storage. We tackle this issue by designing a novel data structure called *$\mathcal{R}$eplica*, which stores all instances of a subgraph pattern on demand in compressed manner. Using $\mathcal{R}$eplica as the data storage platform, we design a single-step, best-first exploration algorithm to detect correlated subgraph pairs efficiently (§3). We further speed up the mining process by designing a near-optimal approximtion algorithm (§4).
- We empirically demonstrate effectiveness and efficiency of our methods on seven real graphs, while also detailing concrete case studies. We establish that the proposed algorithm is up to 5 orders of magnitude faster than baseline strategies and scalable in million-sized networks (§**??**).

0.2 Preliminaries

0.2.1 Background

A data graph $G = (V, E, L)$ has a set of nodes V, a set of edges $E \subseteq V \times V$, and a label set $\mathbb{L}$ s.t. every node $v \in V$ is associated with a label, i.e., $L(v) \in \mathbb{L}$. For simplicity, we focus on bidirectional, node labeled, and unweighted graphs. However, the proposed models and algorithms can also be applied to weighted and edge labeled graphs, e.g., in our experiments (§ **??**), we demonstrate results over edge labeled graphs.

Subgraph Isomorphism. Given a data 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)** $\forall (v_1, v_2) \in E_Q, (M(v_1), M(v_2)) \in E$.

Support. To find frequent subgraphs[1] from a single, large graph, several definitions of subgraph support (σ) have been proposed, e.g., maximum independent sets (MIS) **?**, minimum image-based (MNI) **?**, and harmful overlap (HO) **?**. We adopt MNI **?** due to following reasons. **(1)** MNI can be efficiently computed from subgraph isomorphic instances; whereas the computation of MIS and

[1]We use 'subgraph' and 'subgraph pattern' interchangeably in the paper.

HO are NP-complete **??**. **(2)** MNI provides a superset of results of the two other metrics; thus MIS or HO-based results can be identified via an expensive postprocessing step **?**. **(3)** MNI (as well as two other metrics) is *downward closure*: The support of a supergraph $Q_1 \succeq Q$ is smaller (or equal) than that of its subgraph Q, i.e., $\sigma(Q_1) \leq \sigma(Q)$.

Minimum Image-based (MNI) Support ?. It is based on the number of unique nodes in G that a node of the pattern Q is mapped to. Formally,

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

Example 1. *Figure* **??** *shows subgraph isomorphism. For a subgraph isomorphic* mapping M*, 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, e.g.,* **(1)** $M_1(v_1) = u_3$*,* $M_1(v_2) = u_2$*,* $M_1(v_3) = u_7$*;* **(2)** $M_2(v_1) = u_3$*,* $M_2(v_2) = u_1$*,* $M_2(v_3) = u_2$*, etc. The MNI support of* Q *is* 1*, which is due to node* v_1*: It is mapped to only one node in* G*, i.e.,* u_3 *for all mappings. The nodes in the set* $\{M(v)\}$ *for different mappings* M *are called the* images *of* v*.*

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

0.2.2 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 data 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 the definition of MNI support.

For consistency with MNI, we group subgraph instances.

Definition 1 (Instance Grouping)**.** *Given the data 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* $\mathbb{M}(v^*) = \{M_1(v^*), M_2(v^*), \ldots, M_{\sigma(Q)}(v^*)\}$ *the images of* v^**. Notice that* $\sigma(Q)$ *is the MNI support of* Q*,* $\sigma(Q) \leq s$*, and* $M_j(v^*) \in G$ *is a mapping of* $v^* \in Q$*, 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 2. *For data graph* G *and graph pattern* Q_1 *in Figure* **??***, 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 data graph G. Clearly, $h \geq 0$ 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 patterns* Q_1 *and* Q_2 *in the data 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* $h \geq 0$*, assume that* $\sigma(Q_1) \leq \sigma(Q_2)$*. We define the correlation* $\kappa(Q_1, Q_2, h)$ *as:*

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

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, $\kappa(Q_1, Q_2, h) = \kappa(Q_2, Q_1, h)$.

Example 3. *Let us consider two subgraph patterns* Q_1 *and* Q_2 *in the data graph* G *(Figure* **??***), 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* $\kappa(Q_1, Q_2, h = 1) = 2$*.*

We are now ready to define our problem formally.

Problem 1. Top-k **Correlated Subgraphs Mining** (CSM)**.** *Given the data 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* $\kappa(Q_1, Q_2, h)$*, and for each subgraph pattern* $\sigma(Q_1) \geq \Sigma$*,* $\sigma(Q_2) \geq \Sigma$*.*

Analogous to the *association rule mining* over a transaction dataset **?**, we consider both: **(1)** a *minimum support threshold* Σ such that each constituent subgraph pattern is individually frequent, and **(2)** a *correlation measure* between a pair of frequent subgraph patterns so to identify the top-k pairs based on their correlations. Notice that **(i)** if Q_1 is a subgraph of Q_2, or vice versa, the correlation between them is not interesting. Thus in our methods, we remove those pairs which are related by subgraph and supergraph relationships. Similarly, **(ii)** if Q_1 and Q_2 have high correlation *only* because they are subgraphs of a frequent pattern Q_3, i.e., $Q_3 \succeq Q_1$ and $Q_3 \succeq Q_2$, such correlations are also not interesting. In our solution framework, we devise a technique to detect and eliminate such subgraph pairs from our top-k result.

0.2.3 Theoretical Characterization

The correlation metric satisfies several interesting properties.

Lemma 1. *Correlation metric $\kappa(Q_1, Q_2, h)$ is not downward closure. Specifically, consider that Q_3 is a subgraph of Q_2, i.e., $Q_2 \succeq Q_3$. Then the following $\kappa(Q_1, Q_3, h) \geq \kappa(Q_1, Q_2, h)$ does not always hold.*

The downward closure property does not always hold because as one grows the size of the pattern (e.g., from Q_3 to Q_2), the super pattern Q_2 may now have larger size instances that are closer to the instances of the other pattern Q_1. In Figure **??**, let us consider Q_3 as a single node with label W, whereas the patterns Q_1 and Q_2 are shown in the figure. Clearly, $Q_2 \succeq Q_3$. We notice that $\kappa(Q_1, Q_3, h = 1) = 1$ and $\kappa(Q_1, Q_2, h = 1) = 2$. Hence, the downward closure property does not always hold.

Lemma 2. *Correlation metric $\kappa(Q_1, Q_2, h)$ is not upward closure. In particular, consider that Q_4 is a supergraph of Q_1, i.e., $Q_4 \succeq Q_1$. Then the following $\kappa(Q_1, Q_2, h) \leq \kappa(Q_4, Q_2, h)$ does not always hold.*

The upward closure property does not always hold because as one grows the size of the pattern (e.g., from Q_1 to Q_4), the super pattern Q_4 may naturally have lower MNI support (and, hence smaller number of instance-groups). This can reduce its correlation with the other pattern Q_1. In Figure **??**, let us consider Q_4 consisting of three nodes $X - Y - X$, whereas the patterns Q_1 and Q_2 are shown in the figure. Clearly, $Q_4 \succeq Q_1$. We notice that $\kappa(Q_1, Q_2, h = 1) = 2$ and $\kappa(Q_4, Q_2, h = 1) = 1$. Hence, the upward closure property does not always hold.

Since upward and downward closure properties are widely used in frequent pattern mining problems **??????** — in particular, for early termination of the proposed algorithms, Lemma **??** and **??** demonstrate the inherent complexity of our CSM problem.

Fortunately, Lemma **??** would be useful to design an early termination criteria in our algorithm.

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

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

0.3 Exact Algorithm: CSM-E

In this section, we design an *exact* and *holistic* algorithm, referred to as CSM-E (an abbreviation for Correlated Subgraphs Mining-Exact) that follows *best-first exploration* of frequent patterns, coupled with an *early termination* strategy, to report the top-k correlated subgraph pairs in a large network.

0.3.1 Overview

To find the relevant frequent patterns and the top-k correlated pairs in a holistic manner, we consider a *pattern search tree* T, whose n-th level vertices [2] contain n-edge frequent subgraphs. In other words, the first level of the pattern search tree consists of all frequent edges from the data graph G. For any frequent subgraph in the pattern search tree, we use *subscripts to label its nodes according to their discovery order* **?**. Thus, in a frequent subgraph in T, $i < j$ indicates that v_i is discovered before v_j. If there is an edge from v_i to v_j with $i < j$, then it is called a *forward edge*; otherwise the edge is denoted as a *backward edge*. We call v_0 the root and v_n the rightmost vertex, where n is the number of nodes in that frequent subgraph. The direct path (consisting of forward edges only) from v_0 to v_n is referred to as the *rightmost path*.

To construct $(n+1)$-th layer from the n-th layer in the pattern search tree T, a vertex in the n-th layer (which is a frequent subgraph, say P, from G having n edges) is augmented with a new edge, and thus we create a new subgraph, say C. We insert C as a vertex in the $(n+1)$-th layer of T if and only if C is also a frequent subgraph in G. We refer to P as a *parent vertex* and C as its *child vertex* in T. Note that for an n-edge graph, it may have n different ways to be formed from $(n-1)$-edge graphs if we do not consider isomorphism. The generation of *duplicate graphs* is redundant and could affect the overall efficiency. To reduce the creation of duplicate graphs, we follow the strategy developed in gSpan **?**: *A new edge can only grow from nodes along the rightmost path*. In particular, **(1)** backward edges can only grow from the rightmost node v_n of P, while **(2)** forward edges can grow from nodes on the rightmost path in P. The *completeness guarantee* of generating all frequent subgraphs via the aforementioned *rightmost extension* strategy can be found in **?**, and we omit the proof for brevity.

We note that the pattern search tree T is not fully materialized, instead it is built on demand. For example, gSpan **?** explores T in depth-first manner; whereas in earlier apriori-based approaches (e.g., AGM **?** and FSG **?**) construct T in breadth-first manner. However, we find that

[2] To distinguish the nodes of the pattern search tree T from those of the data graph G, we use the notation "vertices" for the pattern search tree.

neither depth-first exploration, nor breadth-first exploration is efficient for our **Top-k Correlated Subgraphs** discovery problem. Rather, we propose a novel *best-first exploration* algorithm, details of which are given in § **??**.

Next, when a new frequent pattern C is added in the pattern search tree T, we compute C's correlation with previously explored vertices in T (i.e., frequent patterns from G that have been discovered earlier than C). In § **??**, we discuss a memory-efficient data structure, called $\mathcal{R}$eplica, to find correlation between a pair of frequent subgraph patterns.

Finally, we employ a *priority queue* $\mathcal{Q}$ to store the top-k correlation values (and the corresponding correlated subgraph pairs) found thus far. In § **??**, we further propose an *early termination* strategy to speed up search, while returning the *exact* top-k correlated pairs upon termination. This completes our algorithm.

Best-First Exploration

A pair of subgraph patterns having individual higher support values can be expected to have higher correlation between them. This is as such subgraph patterns will have many instances which would likely be closer to each other in the data graph G. Therefore, in the earlier stages of our algorithm, it is more beneficial to consider patterns with higher support values; it can be achieved via the best-first exploration of the pattern search tree. In particular, let us denote by $\mathbb{P}$ the set of frequent patterns currently in the pattern search tree T, whereas we represent by $\mathbb{C}$ the set of their children that are frequent and also *not* included in T. Among all patterns in $\mathbb{C}$, we pick the one C^* with the maximum support for inclusion in T. We implement $\mathbb{C}$ as another *priority queue* (referred to as the *search queue* in our algorithm) to quickly extract C^* from it. Formally,

$$\mathbb{C} = \{C : \exists P \in T, C \text{ is } P\text{'s child}, \sigma(C) \geq \Sigma, C \notin T\}$$
$$C^* = \arg\max_{C \in \mathbb{C}} \sigma(C)$$

If two or more patterns in $\mathbb{C}$ have the same maximum support, we add one more condition in our best-first exploration: Among them, the pattern with less number of edges is extracted earlier from $\mathbb{C}$. This rule coupled with the downward-closure property of the MNI support ensures that *all subgraphs of C^* must be included in T, before C^* is extracted from the search queue $\mathbb{C}$.*

We notice that unlike gSpan **?**, depth-first exploration of the pattern search tree is not optimal in our case since it identifies many subgraph-supergraph pairs at earlier stages, which are not useful for the **Top-k Correlated Subgraphs** finding problem. On the other hand, breadth-first exploration is more memory intensive, and based on our detailed empirical results in § **??**, it is still inefficient compared to our proposed best-first exploration strategy.

Removing duplicates. While the rightmost extension rules in gSpan **?** reduce the creation of

duplicate graphs, it cannot entirely remove them. To completely eliminate duplicated subgraphs, we further take advantage of the *minimum DFS code* **?**: whenever a subgraph C^* is extracted from $\mathbb{C}$, we compute its minimum DFS code, denoted as $MinDFS(C^*)$. We also use a hash table H to store all the minimum DFS codes for the vertices (i.e., frequent subgraphs) already in T. When a new subgraph C^* is extracted, we perform a lookup for $MinDFS(C^*)$ in H. If found, then C^* must have been discovered before, since two graphs will have the same minimum DFS code if and only if they are isomorphic **?**, therefore we do not insert C^* in T.

Removing subgraph-supergraph pairs. To avoid calculating correlations between subgraph-supergraph pairs, we check if C^* is a supergraph of already included frequent patterns in T by performing an explicit subgraph isomorphism check between C^* and every pattern Q in T. Since practically C^* and Q are small subgraphs and T is also not exceptionally large, we find this to be an inexpensive operation.

Early Termination Criteria

Our objective is to mine the top-k pairs of correlated (frequent) subgraphs, and ensure that some other pair of frequent subgraphs cannot have a higher correlation (κ) than any pair in our Top-k *priority queue* $\mathcal{Q}$. A closer look at Lemma **??** mentioned in § **??** allows us to deduce the following. Consider that C^* is the current frequent subgraph extracted from $\mathbb{C}$. By Lemma **??**, $\kappa(C^*, Q, h) \leq \sigma(C^*)$, for all frequent subgraphs Q already in the pattern search tree T. Moreover, for any other pattern C_1 that would be added in T after C^*, $\sigma(C_1) \leq \sigma(C^*)$ due to best-first exploration thereby resulting in $\kappa(C_1, Q, h) \leq \sigma(C_1) \leq \sigma(C^*)$, for all Q in T. Hence, if $\sigma(C^*)$ is lower than the least correlation value (κ) currently stored in $\mathcal{Q}$, while $\mathcal{Q}$ being full, we can safely terminate our search, and report the subgraph pairs in $\mathcal{Q}$ as our exact solution set.

It is also possible that k correlated pairs do not even exist in the data graph G, given some higher minimum support threshold Σ and larger values of k. In this case, the *priority queue* $\mathcal{Q}$ will never get full, and yet there would not exist any more frequent subgraph to include in T. Thus, we terminate our algorithm.

Removing subgraph pairs with high correlation only due to a frequent supergraph. We now turn our attention to the problem of disregarding subgraph pairs Q_1 and Q_2 from our top-k priority queue when the pair has high correlation *only* because both Q_1 and Q_2 are subgraphs of another frequent pattern C^*, i.e., $C^* \succeq Q_1$ and $C^* \succeq Q_2$.

In particular, we are concerned when the supergraph C^* of both Q_1 and Q_2 has the same support as the correlation between Q_1 and Q_2, where the pair Q_1, Q_2 is currently in the top-k priority queue $\mathcal{Q}$. Since $\sigma(C^*) = \kappa(Q_1, Q_2, h)$, and the pair Q_1, Q_2 is in $\mathcal{Q}$, the termination criteria of our algorithm will not be satisfied. Next, to remove the pair Q_1, Q_2 from $\mathcal{Q}$, we incorporate the following procedure in our mining algorithm: every time a new pattern C^* is extracted from the search queue $\mathbb{C}$, if the termination criteria is not satisfied, we check if there exists a correlated pair

Algorithm 1: MINING TOP-k CORRELATED PAIRS

Input: data graph $G = (V, E, L)$, minimum support threshold Σ, distance threshold $h \geq 0$, positive integer k
Output: Top-k pairs of correlated patterns *s.t.* each pattern has support $\geq \Sigma$

```
1  Initialize Pattern Search Tree T ← ∅
2  Initialize Search Queue ℂ ← Frequent Edges in G
3  Initialize Hash Table H ← ∅
4  Initialize Priority Queue 𝒬 ← ∅ with maximum size k
5  while ℂ is non-empty do // Search
6      Pattern C* ← Best-First-Pop(ℂ) (§ ??)
7      if TERMINATION CONDITION (§ ??) is True then
8          Goto Line 19
9      if MinDFS(C*) ∉ H then
10         if ∃⟨Q1, Q2⟩ ∈ 𝒬, κ(Q1, Q2, h) = σ(C*), C* ⪰ Q1, Q2 then Remove ⟨Q1, Q2⟩ from 𝒬
11         foreach Q ∈ T do
12             if Q is not a subgraph of C* then
13                 Compute correlation κ(C*, Q, h) (§ ??)
14                 Insert κ(Q1, Q2, h) in 𝒬 (if necessary)
15         Insert C* in T, MinDFS(C*) in H
16         Find Children(C*) via Rightmost Extension (§ ??)
17         foreach Child∈ Children(C*) do
18             if σ(Child)≥ Σ then Insert Child in ℂ
19 return Top-k correlated pairs currently in 𝒬
```

Q_1, Q_2 in $\mathcal{Q}$, such that both Q_1 and Q_2 are subgraphs of C^* and $\sigma(C^*) = \kappa(Q_1, Q_2, h)$, in which case we eliminate the pair Q_1, Q_2 from $\mathcal{Q}$.

Putting Everything Together

Our pipeline is described in Algorithm **??**. It begins with finding all frequent edges in data graph G, which are then queued in the search queue $\mathbb{C}$ following a frequency-determined priority ordering (Line 2). *Search* begins and continues as long as $\mathbb{C}$ contains queued patterns and the termination condition is unsatisfied (Lines 5-18). C^*, selected as the best-first choice from $\mathbb{C}$, is processed for correlation (§ **??**) with every other previously explored pattern Q in search tree T subject to satisfaction of constraints described in § **??**, **??** (Lines 6-13). Top-k priority queue is updated based on the computed κ values (Line 14). C^* is then inserted in T and its minimum DFS Code is inserted in H (Line 15), followed by the extension of C^* to generate its *child* patterns. All frequent children of C^* are inserted in $\mathbb{C}$ for future processing and the loop continues (Lines 16-18). Upon termination, the algorithm returns Top-k pairs of correlated subgraph (Line 19).

0.3.2 Storing Subgraph Instances: The Replica

We now focus on the problem of correlation computation between two subgraph patterns which requires enumerating and finding distances between every pair of instances for both these patterns. This is memory intensive and computationally demanding due to the following reasons: **(1)** Storing all instances of all frequent subgraphs discovered so far can easily overwhelm the memory. We notice that for the frequent subgraph mining algorithms, e.g., in gSpan **?** and GraMi **?**, storing all instances of all discovered frequent patterns is not required. This illustrates an additional chal-

lenge of our problem. **(2)** Many redundant distance computations could take place: assume two instances I_1 and I_2 for patterns Q_1 and Q_2 respectively, are within h-hops in G. Consider a superpattern $Q_3 \succeq Q_1$, and I_3, which is extended from I_1, is an instance of Q_3. To compute correlation between Q_3 and Q_2, verifying the distance between their instance pairs I_3 and I_2 respectively is redundant, since it is guaranteed that they would also be within h-hops in G.

We solve both these challenges with an efficient data structure, called $\mathcal{R}$eplica that stores all instances of a subgraph pattern in a compressed manner (§ **??**). Our $\mathcal{R}$eplica structure also helps quickly generate instances of a superpattern extended from a parent pattern (§ **??**). Moreover, we design an *incremental distance indexing* method (§ **??**) that permits deletion of $\mathcal{R}$eplicas of those patterns in T for which all their frequent children patterns (obtained via rightmost extension) have also been included in T. In other words, at any point in our algorithm, we only store $\mathcal{R}$eplicas for the following patterns: **(1)** The pattern C^* that is currently extracted from the search queue $\mathcal{Q}$ (Line 6, Algorithm **??**), and **(2)** *only* those patterns in T for which they have at least one child in the search queue $\mathbb{C}$. This greatly facilitates in reducing memory footprints and redundant distance computations. Finally, we introduce the correlation computation strategy in § **??**.

$\mathcal{R}$eplica Data Structure

Given the data graph G and a pattern $Q = (V_Q, E_Q, L)$, $\mathcal{R}\text{eplica}(Q) = (V_{\mathcal{R}(Q)}, E_{\mathcal{R}(Q)}, L)$ is a subgraph of G, constructed by the *graph-union* [3] of all instances of Q in G. Since $\mathcal{R}\text{eplica}(Q)$ is a subgraph, it is stored as an adjacency list in the memory.

In addition to the aforementioned graph, we store two kinds of node mappings as follows:

Forward Node Mapping. $\forall u \in V_Q$, forward node mapping $Mapping(u, \mathcal{R}\text{eplica}(Q))$ stores all nodes $v \in V_{\mathcal{R}(Q)}$, such that v is a mapping of u in some instance I of Q within $\mathcal{R}\text{eplica}(Q)$.

Inverse Node Mapping. $\forall v \in V_{\mathcal{R}(Q)}$, inverse node mapping $Mapping^{-1}\ (v, Q)$ stores the set of all $u \in V_Q$ such that $v \in Mapping(u, \mathcal{R}\text{eplica}(Q))$.

$\mathcal{R}$eplica provides a middle-ground compared to two extreme alternatives: **(1)** *explicitly* store all instances of all generated patterns **?**, or **(2)** enumerate all instances of a pattern whenever required (in runtime) from the *large* data graph G. In contrast to these alternatives, the *$\mathcal{R}$eplica is economical from both storage and efficiency point of view: it not only avoids potential memory requirement issues by* implicitly *storing all instances of a pattern, but also lays a robust foundation for carrying out efficient correlation calculations that we shall discuss in §* **??**.

Note that both forward and inverse node mappings can be deduced from the $\mathcal{R}\text{eplica}(Q)$ subgraph via subgraph isomorphism with the pattern Q. However, we explicitly maintain these mappings for computational efficiency: Forward Node Mapping allows us to readily compute $\sigma(Q)$ and also assists in $\mathcal{R}$eplica generation, while the Inverse Node Mapping enables us to quickly

[3] The union $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2$ of graphs $\mathcal{G}_1 = (V_1, E_1, L)$ and $\mathcal{G}_2 = (V_2, E_2, L)$ is $\mathcal{G} = (V_1 \cup V_2, E_1 \cup E_2, L)$.

Algorithm 2: BUILD REPLICA FOR A NEW PATTERN

Input: data graph $G = (V, E, L)$, parent pattern $Q = (V_Q, E_Q, L)$, $\mathcal{R}\text{replica}(Q) = (V_{\mathcal{R}(Q)}, E_{\mathcal{R}(Q)}, L)$, child pattern $R = (V_R, E_R, L)$, extending node: $u \in V_Q$, extending edge: $(u, v) \in E_R$
Output: $\mathcal{R}\text{replica}(R)$

1 $DFS\ List(Q) \leftarrow$ get edge list via rooted DFS in Q with u as root
2 Initialize $instance \leftarrow \emptyset, \mathbb{I} \leftarrow \emptyset$
3 **foreach** $u' \in Mapping(u, \mathcal{R}\text{replica}(Q))$ **do**
4 $\quad instance.\text{add}(u \mapsto u')$
5 $\quad$ **foreach** edge $(u', v') \in E_G$ that maps to the extending edge $(u, v) \in E_R$ **do**
6 $\quad\quad instance.\text{add}(v \mapsto v')$
7 $\quad\quad \mathbb{I} \leftarrow$ FINDALLINSTANCES(G, Q, $\mathcal{R}eplica(Q)$, R, $DFS\ List(Q)$, $instance$, $\mathbb{I}$)
8 $\quad\quad$ UPDATEREPLICA(R, $\mathcal{R}eplica(R)$, $\mathbb{I}$)
9 $\quad\quad instance.\text{delete}(v \mapsto v')$
10 $\quad instance.\text{delete}(u \mapsto u')$
11 **return** $\mathcal{R}\text{replica}(R)$

Algorithm 3: FINDALLINSTANCES:EXACT METHOD

Input: data graph $G = (V, E, L)$, parent pattern $Q = (V_Q, E_Q, L)$, $\mathcal{R}\text{replica}(Q) = (V_{\mathcal{R}(Q)}, E_{\mathcal{R}(Q)}, L)$, child pattern $R = (V_R, E_R, L)$, $DFS\ List(Q)$, partial isomorphism of R: $instance$, $\mathbb{I}$
Output: $\mathbb{I}$: set of all instances of R in G consistent with input partial isomorphism $instance$

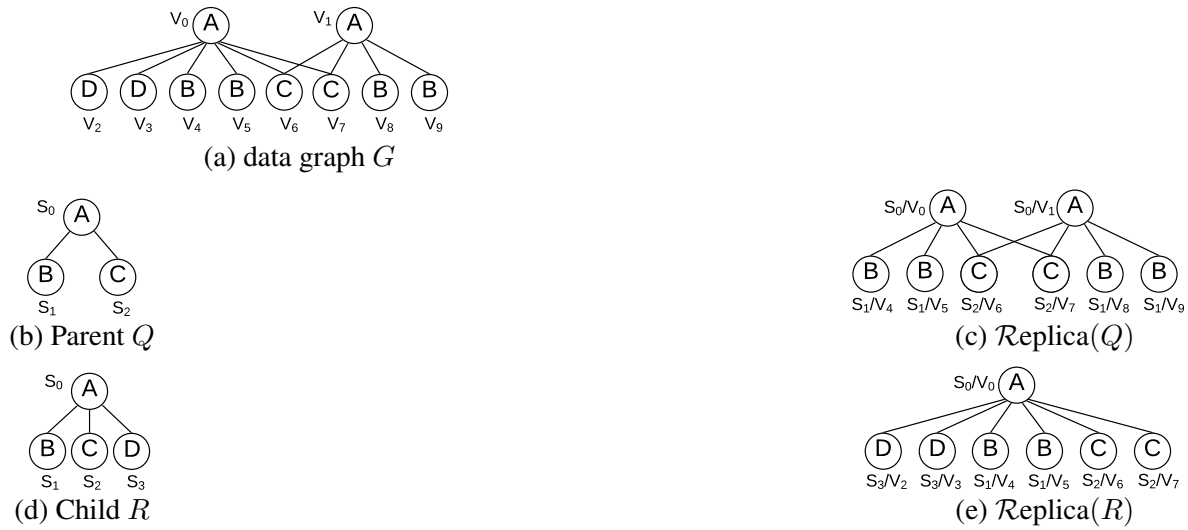
1 **if** $instance$ is Found **then**
2 $\quad$ **return** $\{instance\}$
3 **else**
4 $\quad e = (p, c) \leftarrow$ NEXTQUERYEDGE($DFS\ List(Q)$)
5 $\quad v' \leftarrow instance(p)$
6 $\quad$ **if** e is a backward edge **then**
7 $\quad\quad$ **if** an edge $(v', instance(c))$ exists in $E_{\mathcal{R}(Q)}$ **then**
8 $\quad\quad\quad \mathbb{I} \leftarrow \mathbb{I} \cup$ FINDALLINSTANCES(G, Q, $\mathcal{R}eplica(Q)$, R, $DFS\ List(Q)$, $instance$, $\mathbb{I}$)
9 $\quad\quad$ **else**
10 $\quad\quad\quad$ **return** $\emptyset$
11 $\quad$ **else**
12 $\quad\quad P_c \leftarrow$ FILTERCANDIDATES(v', c, Q, $\mathcal{R}eplica(Q)$)
13 $\quad\quad$ **foreach** $w' \in P_c$ *s.t.* w' is not matched in $instance$ **do**
14 $\quad\quad\quad instance.\text{add}(c \mapsto w')$
15 $\quad\quad\quad \mathbb{I} \leftarrow \mathbb{I} \cup$ FINDALLINSTANCES(G, Q, $\mathcal{R}eplica(Q)$, R, $DFS\ List(Q)$, $instance$, $\mathbb{I}$)
16 $\quad\quad\quad instance.\text{delete}(c \mapsto w')$
17 $\quad$ **return** $\mathbb{I}$

prune candidate nodes of $\mathcal{R}\text{replica}(Q)$ for matching with a node in Q while performing isomorphism. The utility of using these mappings will become apparent in § **??**.

On-Demand Replica Generation for a New Pattern

We create $\mathcal{R}$replicas in an incremental manner: given $\mathcal{R}\text{replica}(Q)$ of a parent pattern Q, we generate $\mathcal{R}\text{replica}(R)$ of its child pattern R. Our exact approach for replica extension is given in Algorithm **??**, which essentially describes a backtracking procedure for subgraph isomorphism similar to the *Ullman's algorithm* **?**.

Algorithm **??** begins with a depth-first search (DFS) procedure (Line 1) executed on parent Q, selecting $u \in V_Q$ at which the *extending edge* (u, v) is grown as the *root*. We call u the *extending node*. Both forward and backward edges encountered in the DFS starting at u are recorded in an ordered list called the *DFS List*, which guides the isomorphism performed subsequently. For every mapping u' of u in $\mathcal{R}\text{replica}(Q)$, the algorithm attempts to enumerate all instances of child R in G fixing $u \mapsto u'$ (Lines 3-10). It does so by matching v next, the newly extended node to all



(a) data graph G (b) Parent Q (c) $\mathcal{R}$eplica(Q) (d) Child R (e) $\mathcal{R}$eplica(R)

Figure 4: $\mathcal{R}$**eplica generation for a new subgraph pattern**

nodes $v' \in V$ adjacent to u', such that edge $(u', v') \in E$ is a valid mapping for $(u, v) \in E_R$ (Lines 5-6). It then invokes FINDALLINSTANCES (Algorithm **??**) which uses $\mathcal{R}$eplica(Q) to find every instance of R such that $u \mapsto u'$ and $v \mapsto v'$ (Line 7).

Algorithm **??**, as invoked above, recursively enumerates all instances of R in a depth-first manner following *DFS List(Q)*. In the general case (Lines 4-17), the algorithm begins by invoking NEXTQUERYEDGE which returns one edge at a time from E_Q in the order of *DFS List(Q)*. Edge $e = (p, c)$, thus returned, connects nodes $p, c \in V_Q$ such that c is the pattern node to be matched next; p is already matched to $v' \in V_{\mathcal{R}(Q)}$. (The first call to Algorithm **??** in every iteration of the inner loop in Algorithm **??** always has p matched with u'). If, however, e is a backward edge, c is already matched in which case the algorithm checks whether an edge exists in $\mathcal{R}$eplica(Q) connecting v' and $instance(c)$: if it does exist, it proceeds with the search for the next node matching, but returns unsuccessfully if it does not. If e is a forward edge, the algorithm calls FILTERCANDIDATES to compute the candidates set P_c for storing all nodes $w' \in V_{\mathcal{R}(Q)}$ for matching c such that: **(1)** w' is a neighbor of v' in $\mathcal{R}$eplica(Q); **(2)** c exists in the Inverse node mapping set for w' in $\mathcal{R}$eplica(Q). That is, $c \in Mapping^{-1}(w', Q)$; Next, for every node $w' \in P_c$ such that w' has not already been matched in the current *instance*, the algorithm attempts the match $c \mapsto w'$ in *instance* and recursively calls FINDALLINSTANCES to match remaining pattern nodes following the edges in *DFS List*. *Base case* (Lines 1-2) occurs when the algorithm finds an *instance* of R, which it simply returns.

The set of all instances $\mathbb{I}$ thus found is returned by Algorithm **??** (Line 17) and is recorded by Algorithm **??**. In UPDATEREPLICA (Algorithm **??**, Line 8), the algorithm updates $\mathcal{R}$eplica(R) by performing a graph union with all instances in $\mathbb{I}$. It also updates the *Mappings* and *Mappings*$^{-1}$ indices to record new mappings found for every $instance$ in $\mathbb{I}$.

Example 4. *Initially, Q and $\mathcal{R}eplica(Q)$ are shown in Figs.* **??** *and* **??**, *respectively. Child R, extended from Q at* extending node s_0 *using the* extending edge (s_0, s_3) *is given in Figure* **??**.

Assume DFS List *for* DFS *starting at* s_0 *records edges* (s_0, s_1) *and* (s_0, s_2) *in that order. To construct* $\mathcal{R}eplica(R)$*, the algorithm iterates over* $Mapping(s_0, \mathcal{R}eplica(Q))$*, i.e. set* $\{v_0, v_1\}$*. With* $s_0 \mapsto v_0$*, the algorithm then considers vertices in* G *(Fig.* **??***) for matching* s_3*. Thus, for each of* $s_3 \mapsto \{v_2, v_3\}$*, the algorithm makes recursive invocations to map* s_1 *and* s_2 *following the* DFS List *thus successfully enumerating instances. With* $s_0 \mapsto v_1$ *however, no mappings for matching* s_3 *exist. Graph union of all instances of* R *thus enumerated results in* $\mathcal{R}eplica(R)$ *depicted in Fig.* **??**.

Indexing to Facilitate $\mathcal{R}$eplica Deletion of Old Patterns

We build and maintain two distance-based indexes, which permit us deleting $\mathcal{R}$eplicas of old patterns once all their frequent children patterns (via rightmost extension) have been included in the pattern search tree T.

Proximity Nodes. For each frequent node u in data graph G, we store proximity nodes of u that are also frequent, denoted as $CorV(u)$: for each node $v \in CorV(u)$, it holds that $d(u, v) \leq h$.

Proximity Patterns. For each frequent node u in G, we store patterns that are already included in T and are within distance h from u. This is denoted as $CorP(u)$. For each pattern $Q \in CorP(u)$, **(1)** $Q \in T$, and **(2)** given the instance-groups $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q)}\}$ of Q, $\exists I' \in \mathbb{I}'$, $\exists v \in I'$, it holds that $d(u, v) \leq h$.

The *proximity nodes index* ($CorV$) is constructed before our mining process starts, while the *proximity patterns index* ($CorP$) is updated incrementally every time a new pattern Q is inserted in the pattern search tree T. The details of $CorP$ maintenance will be discussed in § **??**. Since we already have h-hop proximity information about all existing patterns in T, it allows us to delete their $\mathcal{R}$eplica structures, thereby reducing the memory footprint of our technique.

Correlation Computation

We recall that when a pattern C is extracted from the search queue $\mathcal{Q}$, its correlation is calculated with every pattern already in the pattern search tree T. In the following, we focus on the computation of correlation $\kappa(C, Q, h)$ between C and some $Q \in T$. Due to our best-first exploration strategy, $\sigma(C) \leq \sigma(Q)$, thus we need to verify for every instance-group I' of C, if there is some instance-group of Q within distance h.

First, to find all instance-groups of C, our novel data structure $\mathcal{R}$eplica(C) would be immediately useful: Forward node mapping allows us to readily compute the MNI support of C, and the node v in C having the minimum MNI support. For every $u \in Mapping(v, \mathcal{R}\text{eplica}(C))$, we enumerate from $\mathcal{R}$eplica(C) all instances of C having u (Algorithm **??**). This generates the instance-group of C containing u. In this way, one can also efficiently enumerate all instance-

groups of C.

Given an instance-group I' of C, we find if any instance-group of Q is within distance h from I' as demonstrated below. Consider $CorP(w)$ for all node $w \in I'$. If $\cup_{w \in I'} CorP(w)$ contains Q, this essentially indicates that there is some instance-group of Q within distance h from I'. Ultimately, we count, out of all $\sigma(C)$ instance-groups of C, how many of them are close to at least one instance-group of Q. This count is reported as the correlation $\kappa(C, Q, h)$. In this way, our *proximity nodes index* ($CorP$) aids in efficient correlation computation.

Incrementally Updating Proximity Patterns Index. Finally, we incrementally update the $CorP$ index as follows. If $CorV(w)$ contains u for some $w \in I'$, we include the new pattern C in $CorP(u)$. Thus, we also ensure that at any point in our algorithm, $CorP(u)$ would contain all patterns in T which are within distance h from u.

0.3.3 Complexity Analysis

For simplicity, we use the notations below: The data graph G has n nodes and m edges, among them n_f nodes and m_f edges are frequent based on the input minimum support threshold. We denote by n_h and m_h the maximum number of nodes and edges in the h-hop of a node in G. The pattern search tree T has at most n_T vertices, with n'_T vertices having at least one frequent child in the search queue $\mathcal{C}$, during the execution of our algorithm. Let the maximum number of nodes and edges in a pattern Q mined by our method be n_Q and m_Q, and those in a replica be $n_{\mathcal{R}}$ and $m_{\mathcal{R}}$, respectively. Finally, also assume that the maximum size of the forward node mapping set be M for some node in a frequent pattern mined by our algorithm.

Time Complexity

We start with the complexity analysis of *replica creation* for a new pattern Q (§ **??**). For every mapping of the *extending edge* in G, we enumerate all possible instances of Q using the replica subgraph of its parent pattern. There can be $\mathcal{O}(m)$ matches for the extending edge, whereas the complexity of enumerating all possible instances of Q using the parent's replica is bounded by $\mathcal{O}(M^{n_Q})$. Hence, the time complexity of building the replica of Q is $\mathcal{O}(mM^{n_Q})$.

The creation of *proximity nodes index* (§ **??**) requires h-hop BFS from every frequent node in G, and hence it has total time complexity: $\mathcal{O}(n_f(n_h + m_h))$. In contrast, whenever a new frequent pattern is mined, one requires updating the *proximity patterns index*. This is performed by traversing every node in the replica of the new pattern, and then finding their h-hop frequent nodes using the proximity nodes index — this has time complexity $\mathcal{O}(n_{\mathcal{R}} n_f)$. Since the pattern search tree T has at most n_T vertices, total time complexity due to updating of proximity patterns index is given by $\mathcal{O}(n_{\mathcal{R}} n_f n_T)$.

Correlation computation (§ **??**) between the new pattern Q extracted from the search queue $\mathbb{C}$ and an existing pattern in T is essentially bounded by the complexity of enumerating all instances of Q in its replica, i.e., $\mathcal{O}(M^{n_Q})$. Since T has at most n_T patterns, one performs $\mathcal{O}(n_T^2)$ correlation computations, which has time complexity $\mathcal{O}(n_T^2 M^{n_Q})$.

Considering all above, the time complexity of our exact algorithm, CSM-E [4] is: $\mathcal{O}(M^{n_Q}(n_T^2 + m) + n_{\mathcal{R}} n_f n_T + n_f(n_h + m_h))$. Notice that *we enumerate all instances of a pattern using its replica (or, the replica of its parent), thereby significantly improving the efficiency in comparison with naïvely enumerating all instances over the entire data graph* G. Nevertheless, an exact enumeration of all instances, as given in Algorithm **??**, is still expensive when the replica size is too large (thereby larger M). In § **??**, we shall introduce an approximate method for instances enumeration, which further improves the efficiency, without significantly affecting the accuracy of our solution.

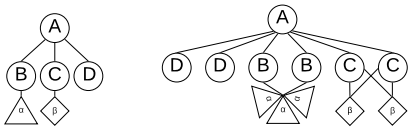
Space Complexity

The space complexity is bounded by the *replica size* and the *size of our index sets*: *forward node mapping* and *inverse node mapping*, as well as *proximity nodes* and *proximity patterns indexes*. Since we only store the replica of $\mathcal{O}(n'_T + 1)$ patterns, this requires $\mathcal{O}((n_{\mathcal{R}} + m_{\mathcal{R}})n'_T)$ space. Both forward and inverse node mappings for a pattern Q has size $\mathcal{O}(n_Q M)$. Finally, proximity nodes and proximity patterns indexes require $\mathcal{O}(n_f^2)$ and $\mathcal{O}(n_f n_T)$ space, respectively. Hence, the overall space complexity is given by: $\mathcal{O}((n_{\mathcal{R}} + m_{\mathcal{R}} + n_Q M)n'_T + n_f^2 + n_f n_T)$.

0.4 Approximate Algorithm: CSM-A

The $\mathcal{R}$eplica construction algorithm (§ **??**) for a subgraph pattern enumerates *all* its isomorphisms in the data graph. This computation is expensive since subgraph isomorphism is *NP-hard*. Moreover, the number of instances of a pattern generally grows exponentially with increasing density and size of the data graph. As a result, Algorithm **??** does not scale well to large query patterns or large and dense data graphs. To address this need for better scalability, in this section, we develop a near-optimal approximation algorithm for efficiently constructing the $\mathcal{R}$eplica. The approximate algorithm does not consider enumerating instances that are likely redundant.

[4] We neglected times required for finding the best pattern from search queue $\mathbb{C}$, verifying its subgraph relationships with existing patterns in T, and computing its rightmost extensions, due to relatively smaller sizes of frequent patterns mined by our algorithm while finding the top-k correlated pairs.



(a) (b)

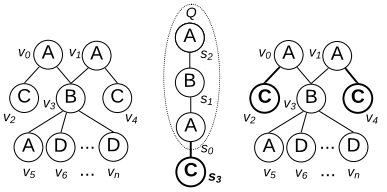
Figure 5: Subgraph Isomorphism. (a) pattern R'. (b) $\mathcal{R}$eplica(R').

0.4.1 Redundant Computations

In many cases, to obtain a $\mathcal{R}$eplica, the identification of *all* instances of the query pattern may not be necessary. To illustrate, let us revisit pattern Q and $\mathcal{R}$eplica(Q) in Figure **??**. To obtain $\mathcal{R}$eplica(R) for Q's child R via the exact approach, edges $e_1 = (v_0, v_2)$ and $e_2 = (v_0, v_3)$ in G are tried as mappings for the *extending edge* $(s_0, s_3) \in E_R$. To record e_1 as a valid *extension edge* mapping, we enumerate all instances of R using $\mathcal{R}$eplica(Q) that are consistent with $(s_0, s_3) \mapsto e_1$. Now, while considering the second edge $e_2 = (v_0, v_3)$ as a mapping of (s_0, s_3), the same enumeration scheme is exactly repeated. Here, we observe that e_2 is *symmetric* to e_1 since both of them share the extending vertex v_0 and both v_2 and v_3 map to s_3. Consequently, any instance mapping that is applicable for e_1 is likely to be applicable for e_2. Thus, enumeration of *all* instances as in Algorithm **??** may be redundant.

The impact of redundant computations in symmetric extensions can be further appreciated from the example presented in Fig. **??**. In Fig. **??**, R' is a subgraph pattern and we are again considering an ("A'', "D'') extension. However, unlike Fig. **??**, where "B'' and "C'' are leaf nodes, here two arbitrary subgraphs α and β are attached. To obtain $\mathcal{R}$eplica(R'), the exact algorithm would enumerate every instance of R', which means the same three mappings of α in $\mathcal{R}$eplica(R') would be visited multiple times through each "B'' for the two ("A'', "D'') mappings; worse, for each α, both the mappings of β would be enumerated twice - once through each "C''. Clearly, these redundant computations can be avoided if we have the ability to identify symmetric edges in the replica. For instance, the two ("A'', "D'') edges in $\mathcal{R}$eplica(R') are symmetric and hence while considering the second ("A'', "D'') edge, we can simply re-use the instance enumerations of the first ("A'', "D''). Even more importantly, while enumerating the instances corresponding to the first ("A'', "D'') edge, we can observe that the two ("B'', α) edges are symmetric and therefore enumerating the three α subgraphs twice can be avoided by reusing the instance mappings from the first ("B'', α) edge. The same applies to the ("C'', β) edges as well.

Armed with this intuition, our goal, therefore, is as follows: (**1**) Identify if two edges in a replica are symmetric to each other, (**2**) For a group of symmetric edges, enumerate all instance mappings for only one edge from the group and then re-use the mappings for the remaining symmetric edges.



(a) data graph G (b) pattern Q and extension to R (c) Extending $\mathcal{R}$eplica(Q) to $\mathcal{R}$eplica(R)

Figure 6: $\mathcal{R}$eplica generation for a new subgraph pattern

0.4.2 Algorithm

First, we define when two edges in a replica are *symmetric*.

Definition 3 (Symmetric Edges)**.** *Edges $e_1 = (v_a, v_b)$ and $e_2 = (v_c, v_d)$ in the replica are symmetric if (**1**) $a = c$, and (**2**) both v_b and v_d are mapped to the same node in the subgraph pattern.*

We now describe the approximate algorithm in place of the exact version (Algorithm **??**). Recall, that edges in the replica are processed in the DFS order. While processing edges in this order, we check if it is symmetric to one of the already enumerated edges. If not, the algorithm proceeds in exactly the same manner as in the exact version except one difference: all of the successful mappings of a replica edge (or node) are stored. On the other hand, if the extension is symmetric, we do not recompute from scratch. Rather, we reuse the mappings that were identified in the previously enumerated symmetric edge, and among these mappings, we check if there exists at least one instance to which the candidate edge for extension can be mapped to. If one such instance is found, the candidate edge is added to the replica.

Example 5. *Consider Fig.* **??***, where edge (s_0, s_3) is extended from pattern Q to get child R. To obtain $\mathcal{R}eplica(R)$ from $\mathcal{R}eplica(Q)$,* CSM-A *first attempts to enumerate instances of R in G by mapping edge (v_0, v_2) to (s_0, s_3). Following this mapping, CSM-A, like CSM-E, enumerates all isomorphisms by mapping $(s_0, s_1) \mapsto (v_0, v_3)$ and $(s_1, s_2) \mapsto \{(v_3, v_5), (v_3, v_1)\}$. Thus, nodes v_1 and v_5 are considered valid mappings for s_2. Next, when* CSM-A *attempts to enumerate instances after mapping $(s_0, s_3) \mapsto (v_1, v_4)$, it recognizes the symmetric relation between (v_1, v_3) and (v_0, v_3) and thus only traverses the already-*confirmed *mappings set $\{v_1, v_5\}$ instead of all the neighbors of v_3. Furthermore, even while traversing the set of confirmed mappings, we stop as soon as one instance is found, which further reduces the computation cost.*

0.4.3 Properties

To understand the *approximation* in the above algorithm, let us revisit Example **??**. The symmetric extension (v_1, v_4) searches only within the confirmed mappings v_1 and v_5 to match s_2. However, s_2 can also be mapped to v_0 to constitute an instance with $s_0 \mapsto v_1$, $s_3 \mapsto v_4$ and $s_1 \mapsto v_3$, which CSM-A will fail to enumerate. To generalize, CSM-A may miss a mapping if some node in the

data graph can be mapped to multiple nodes of the subgraph pattern. In Fig. **??**, this occurs, where v_0 (or v_1) may be mapped to either s_2 or s_0. It can be guaranteed, however, that there are no false positives since any edge that we add to the replica, corresponds to at least one instance from the subgraph. Consequently, we can state the following.

Lemma 4. *For any pair of subgraphs $Q, R \in T$, consider $\sigma(Q)$ to be the* MNI-support *and $\kappa(Q, R, h)$ the correlation value between Q and R at h-hop separation computed via* CSM-A*, and $\sigma^*(Q)$, $\kappa^*(Q, R, h)$ be the respective values computed via* CSM-E*. Then: (1) $\sigma(Q) \leq \sigma^*(Q)$, and (2) $\kappa(Q, R, h) \leq \kappa^*(Q, R, h)$.*

PROOF: The support and correlations counts obtained through CSM-A is a lower bound to CSM-E since CSM-A does not return any false positives during instances enumeration. In other words, the set of pattern instances enumerated in CSM-A is always a subset of the (exhaustive) set found by CSM-E. □

0.5 Experimental Results

We benchmark the proposed algorithms CSM-Exact (CSM-E) and CSM-Approximate (CSM-A) and establish that:

- Replica is effective in enumerating and storing subgraph instances in a compact and cost-efficient manner. By utilizing replica and best-first search, our algorithms scale on million-sized datasets. Furthermore, CSM-A is up to 5 orders of magnitude faster than baseline approaches.
- CSM-A is near-optimal in quality.
- Our algorithms unearth useful patterns and correlations that would not be discovered using existing techniques such as frequent subgraphs mining.

Our implementation is available at `https://github.com/CSM2019/correlated-subgraphs-mining`.

0.5.1 Experimental Setup

All algorithms have been implemented in C++ and compiled using gcc $7.4.0$ in a Linux (Ubuntu 18.04) machine with a single core running at 2.1GHz and 256GB RAM. Our machine uses a large RAM to accommodate the memory requirements of the baseline algorithms.

Datasets

We use seven real networks (Table **??**).

Table 1: Datasets and characteristics

Datasets	Nodes	Edges	#Node labels	# Edge labels	Domain
Chemical	207	205	4	2	Biological
Yeast	$4K$	$79K$	41	1	Biological
Citeseer	$3K$	$4.5K$	6	5	Collaboration
MiCo	$100K$	$1M$	30	106	Collaboration
LastFM	$1.1M$	$5.2M$	$83K$	1	Social Network
Coauthor(DBLP)	$1.7M$	$7.4M$	$11K$	1	Collaboration
Citation(DBLP)	$3.2M$	$5.1M$	$11K$	1	Collaboration

•*Chemical* **?**. This graph represents the structure of an anti-breast tumor compound in the MCF7 Dataset **?**. Each node represents a chemical atom, and two atoms are connected by an edge if they share a chemical bond.

•*Citeseer* **?**. Each node is a publication, and its label categorizes the area of research. Two nodes 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. The labels are in the range of $[0, 5]$ with 0 indicating the top 20 percentile similarity.

•*Yeast* **?**. This dataset contains the protein-protein interaction network in Yeast. The node labels denote their gene ontology tags **?**, which capture their biological functions and properties.

•*Mico* **?**. Mico models Microsoft co-authorship information. Each node 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.

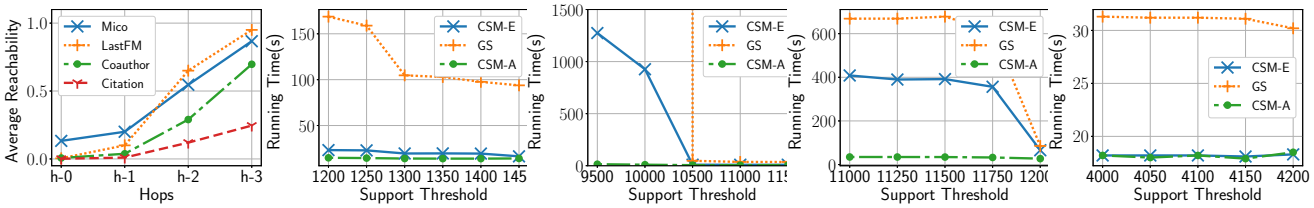
•*LastFM* **?**. This dataset represents the LastFM social network where a node label represents the most frequent singer or music band that the corresponding user listens to.

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

•*DBLP citation* **?**. Each node is a publication, and the label is the publication venue. Two nodes have an edge if one of the two papers is cited by the other.

Parameters

There are three input parameters to our problem: MNI support threshold Σ, the k value of our Top-k results, and the hop-constraint value h, which defines when two instances are distance-wise "close" in the network. We vary each of these parameters and study their impact on the performance. When not specifically mentioned, the default values of k and h are set to 20 and 1, respectively. Here, we point out that $h \geq 3$ is not meaningful since in such a scenario most sub-graph patterns are close to each other and therefore every pair of pattern is correlated. To provide empirical evidence, in Figure **??**, we vary h, and measure the average proportion of frequent pattern instances that are reachable within h hops from a randomly chosen frequent pattern instance. Notice that, at $h = 3$, except for the *Citation* dataset, the reachability is higher than 75%. Even in



(a) (b) *LASTFM* (c) *Mico* (d) *DBLP Coauthor* (e) *DBLP Citation*

Figure 7: (a) Growth of reachability against h. (b-e) Growth of running time against support threshold.

Citation, which is a sparser dataset, the reachability is $> 25\%$, indicating that high values of h do not yield semantically interesting results.

Baselines

We compare CSM with two baselines.

• **GraMi+VF3 (GVF3):** This baseline approach has already been discussed in details in § **??**.

• **GrowStore (GS):** 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.

0.5.2 Efficiency

We compare the efficiency of CSM-E and CSM-A against baselines. Since baselines are exorbitantly slow, we restrict the mining process to only patterns containing at most 5 nodes.

Running time.

Recall that the running time of CSM-A has already been compared with **GVF3** in the Figure **??**. As visible, CSM-A is 5 orders of magnitude faster than **GVF3** in the *Citation* dataset. The gap in performance is narrower in *MiCo* since *MiCo* is a much smaller dataset. In Table **??**, we compare the running times of CSM-E, CSM-A, and **GVF3** at the largest possible support threshold such that there is at least one correlated pair. As clearly visible, even when the support thresholds are set at

Table 2: Running Time Efficiency(s)

Datasets	CSM-A	CSM-E	**GVF3**
Chemical	0.1	0.1	2.5
MiCo	4.5	8.9	1521
LastFM	14	16.3	346000
Coauthor(DBLP)	27.8	64.3	503015
Citation(DBLP)	18.2	18.2	1311474
Citeseer	0.1	0.1	10

the highest possible values, **GVF3** is unable to scale in large datasets. For example, *in the* Citation *dataset, **GVF3** terminates after* 15 *days. In contrast,* CSM-A *and* CSM-E *finish in* 18 *seconds. Overall, both the proposed approaches are orders of magnitude faster.*

Next, we compare CSM-A and CSM-E with **GS**, to clearly understand the impact of replica. Figures **??**-**??** present the running times across various datasets. *As expected,* CSM-A *is the fastest algorithm followed by* CSM-E*, and then **GS**. This result shows that replica is effective in reducing the computation cost.* We also observe that CSM-A is 10 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.

Memory footprint.

Our goal in this experiment is to understand the reduction obtained in memory footprint due to replica. To fully understand the impact of replica, we remove the restriction of mining only patterns of size up to 5. In Figure **??**, we compare the memory footprint of CSM-A with **GS** in *Citeseer*. Since CSM-E has the same memory requirements as CSM-A, we do not show CSM-E in Figure **??**.

Without the size restriction of 5 nodes, **GS** runs out of memory on large datasets, e.g., in *LastFM*, *Citation* and *Coauthor*, even at high support thresholds. In *Citeseer*, which is a small dataset of $4\,500$ edges, **GS** consumes more than $10GB$ of main memory at support threshold of 175; and at the lowest support threshold of 150, it exceeds $256GB$. *In contrast, the memory footprint of* CSM-A *is* 100 *times lower on average*. The memory consumption grows with reduction in support thresholds since more subgraphs become frequent. Overall, the stark difference in memory consumption between CSM-A and **GS** highlights the benefit of replica on memory footprint.

0.5.3 Approximation Quality

We evaluate the approximation quality of CSM-A by comparing it with the top-k answer set of CSM-E (ground-truth). The accuracy of CSM-A is quantified with three metrics:

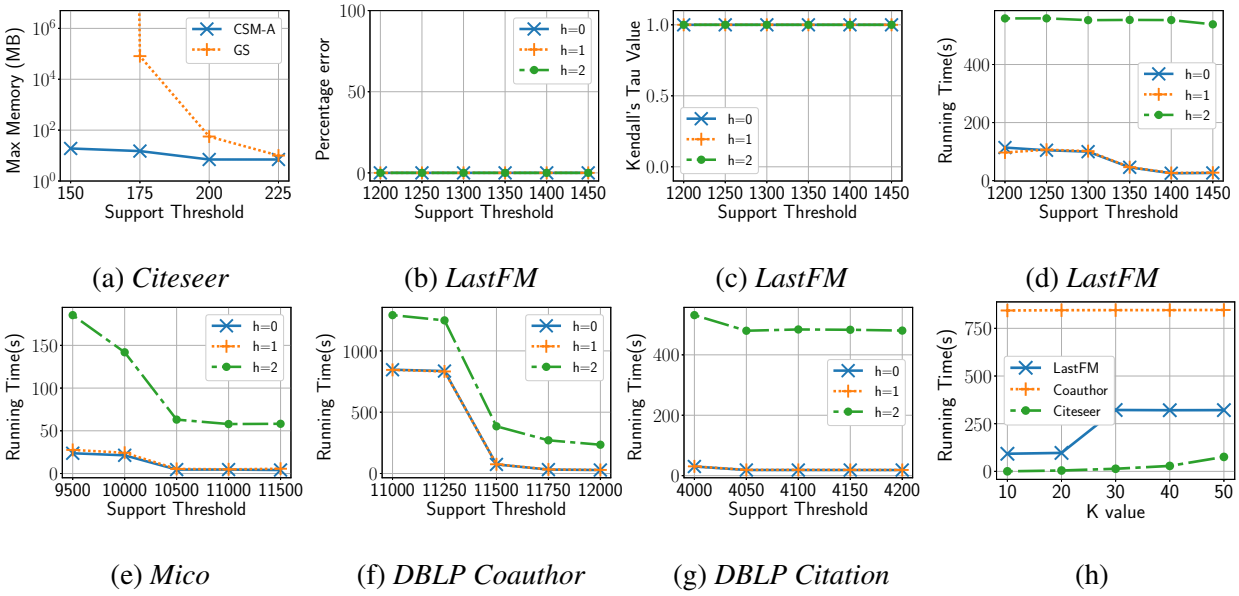
• **Jaccard Coefficient ?**: Jaccard coefficient measures the similarity of the approximate answers with the ground truth.

• **Kendall's Tau ?**: Kendall's Tau measures how accurately the ranking of the patterns in the ground truth is preserved in the top-k answer set of CSM-A. It ranges from 0 to 1, with 1 indicating perfect preservation of the ranking order.

• **Percentage error in correlation count.** This metric measures the error in correlation count introduced by the approximate version. The percentage error for a given pattern p is $\frac{\kappa_p^* - \kappa_p}{\kappa_p^*} \times 100$, where κ_p is the correlation count for the pair of subgraphs p in CSM-A and κ_p^* is the exact value in

Table 3: Quality evaluation of CSM-A

Datasets	Jaccard Coeff	Kendall's Tau	Percentage Error	Support
Chemical	1.0	1.0	0	10
Yeast	1.0	1.0	0	300
MiCo	1.0	1.0	0	9500
LastFM	1.0	1.0	0	1200
Coauthor(DBLP)	1.0	0.98	2.26	11000
Citation(DBLP)	1.0	1.0	0	4000
Citeseer	1.0	1.0	0	150



(a) *Citeseer* (b) *LastFM* (c) *LastFM* (d) *LastFM*

(e) *Mico* (f) *DBLP Coauthor* (g) *DBLP Citation* (h)

Figure 8: (a) Growth of memory consumption against support threshold. (b-c) Approximation quality of CSM-A against support threshold. (d-e) Growth rate of CSM-A's running time against: (d-g) support threshold and (h) k. For (h) the support thresholds are as follows: *LastFM*: 1200, *DBLP Coauthor*:11000, *Citeseer*: 150.

CSM-E for this same pair. From Lemma **??**, we are guaranteed that $\kappa_p^* \geq \kappa_p$ for any pattern p. We compute the percentage error across all common top-k patterns in the approximate and the exact set and report the mean error (in percentage).

We report the accuracies across all datasets in Table **??**. *As can be seen,* CSM-A *produces near-optimal results*. To further analyze how the quality changes with support threshold, in Figures **??** and **??**, we plot the percentage error and Kendall's Tau against support for various values of h. *Consistent with previous observations, the results remain near-optimal throughout*. We do not include Jaccard Coefficient in this experiment since it is always 1. Due to space limitations, we show the result only in *LastFM*. Similar trends are observed across all datasets.

0.5.4 Impact of Parameters

After establishing that CSM-A is accurate and fast, we measure its scalability on large graphs against various parameters.

Running Time.

In § **??**, we had to limit the mining process only to patterns of size up to 5 due to the non-scalability of the baseline algorithms. In the next set of experiments, we remove this constraint and measure the growth rate of running time against the support threshold. In addition, we also plot the running times across all values in the range $h \in [0, 2]$. Figures **??**-**??** present the results in the four largest datasets listed in Table **??**. As expected, the running time decreases with increasing support. *What is more noteworthy is that even at* $h = 2$*,* CSM-A *terminates within* 20 *minutes on million-scale datasets*. As we increase h, time and space required to compute proximate nodes' index ($CorV$) increases and as a result, the time taken for correlation computation also increases due to more correlations obtained.

Figure **??** analyzes the growth rate of running time against k. As k increases, it is expected that more patterns will be processed and hence the running time should increase. The increase, however, is minimal since in proportion to the number of subgraphs in the search space, k is very small.

Memory footprint.

Figure **??** analyzes the memory footprint against support threshold for various values of h. Two key trends emerge. First, *as support decreases, the increase in memory consumption of CSM-A is minimal. This follows from the property that CSM-A maintains only replicas at any point of time*. The small increase in memory results from storing more subgraph patterns (but not their instances). As h increases, the memory required to store the proximate nodes index, which stores all frequent nodes within h hops from each frequent node, goes up.

Search Strategy.

We adopt $Best\text{-}First\ Search(BEST)$ strategy to explore the search space. How would the performance be if we adopt $Breadth\text{-}First\ Search(BFS)$ or $Depth\text{-}First\ Search(DFS)$? Our next experiment answers this question. More specifically, we explore the search space using each of these strategies and track their pruning power by counting the number of patterns popped from the Search Queue. We use this methodology instead of comparing the running time since this is more precise and hardware-independent.

Figure **??** presents the results against k. *BEST is built on the observation that subgraphs with higher frequency tend to have a higher correlation with other frequent subgraphs. This prioritization scheme yields better results across most datasets*. Specifically, the results obtained in *LastFM* and *Chemical* reflect the trends in other datasets as well; we omit them due to space limitations.

0.5.5 Application

To demonstrate the utility of mining correlated subgraph pairs, we present insights derived from mining correlated pairs on the *Yeast* **?** dataset. The node labels (Gene Ontology tags) in this Protein-Protein Interaction (PPI) network (§ **??**) indicate their biological activity. In Figure **??** we present two of the top-20 correlated pairs mined at $\Sigma = 300$. Figures **??** (Q_1) and **??** (Q_2) present the first pair. Q_1 is constituted entirely of units specialising in *ion binding*, while Q_2 is composed of those specialising in *transferase activity*. Keike et al.**??** show that transferases gain enzymatic activity following the binding of ionic ligands, undergoing conformational changes to insulate the ligands from surrounding water molecules. The frequent co-occurrence of these subgraph patterns specializing in two complementary biological functions shows that the proposed model is effective in discovering semantically meaningful associations. Furthermore, this is a pattern that frequent subgraphs mining techniques would not identify.

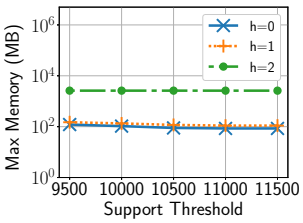
We highlight another pair (Q_3, Q_4) in Figures **??** and **??**. Q_3 is made up of genes that handle responses to *chemicals*, while Q_4 is associated with *transcription*. This co-occurrence is not a coincidence. Specifically, the Gene Ontology**?** database describes in detail the involvement of positive transcription regulation in cellular response to chemical stimulus.

0.6 Related Work

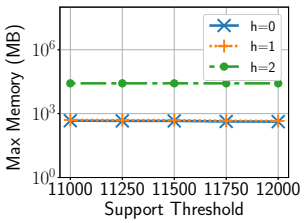
We categorize related work as follows.

• **Frequent subgraphs mining.** To mine a set of graphs, efficient frequent subgraph mining algorithms were proposed, e.g., AGM **?**, FSG **?**, gSpan **?**, PathJoin **?**, MoFa **?**, FFSM **?**, GASTON **?**, SPIN **?**, etc. Techniques were also developed to mine maximal **??**, closed **?**, discriminative **??**, statistically significant **??**, and representative subgraph patterns **??**. These methods adopt subgraph isomorphism as a way to count the support of graph patterns in multiple graphs. For a survey, we refer to **??**.

In the area of mining single massive graphs, **????** developed techniques to calculate the support of graph patterns. The state-of-the-art technique for mining frequent subgraphs in a single-large graph is GRAMI **?**. GRAMI uses MNI **?** as subgraph frequency, and models the problem



(a) *Mico*



(b) *Coauthor(DBLP)*

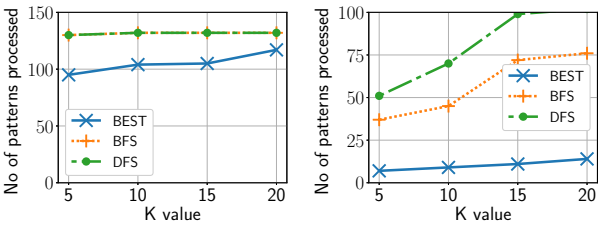
Figure 9: Memory usage of CSM-A against support threshold at different values of h.

of subgraph frequency evaluation as a constraint satisfaction problem. Algorithms for statistically significant graph patterns **?** and discriminative graph patterns **??** over a single graph were also designed.

As noted earlier, correlated subgraphs are different from frequent subgraphs due to the flexibility in which the constituent subgraph instances are connected. Moreover, correlation computation between two subgraph patterns require enumerating and finding distances between every pair of subgraph instances of both these patterns, thereby making our problem more memory intensive and computationally demanding.

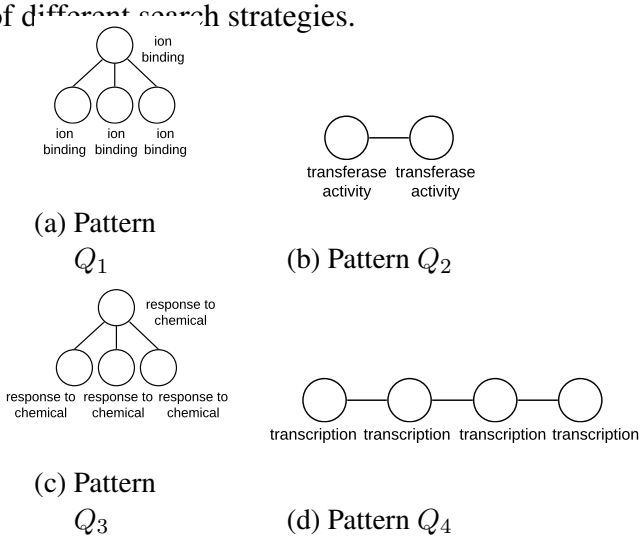
• **Approximate subgraphs mining.** To tolerate certain structural and label differences in two graphs, approximate pattern mining frameworks were developed in **???**, which can find patterns that are missed by exact mining algorithms. Proximity patterns **?** were introduced to mine the top-k set of node labels that co-occur frequently in neighborhoods. Correlations between node labels and dense graph structures were identified in **??**. In the CSM problem, while we allow certain flexibility in terms of how the constituent subgraph instances are connected, we still maintain fixed structures for subgraph instances. Hence, CSM is different from existing works on approximate subgraph mining.

• **Correlation mining in graph databases.** All prior works on correlated graph mining **??????** considered graph databases consisting of multiple graphs. In particular, **????** developed effi-



(a) Sup=1200, $h = 1$ in *LastFM* (b) Sup=10, $h = 1$ in *Chemical*

Figure 10: Performance of different search strategies.



(a) Pattern Q_1 (b) Pattern Q_2

(c) Pattern Q_3 (d) Pattern Q_4

Figure 11: Correlated Subgraph Pairs (Q_1, Q_2) and (Q_3, Q_4)

cient algorithms for searching both the top-k and threshold-based "correlative" subgraphs in the database, which share *similar occurrence distributions with a given query graph*. The incremental and streaming versions of the top-k correlative subgraph search problem were studied in **?** and **?**, respectively. Moreover, Ke et al. **?** designed mining algorithms for automatically finding the top-k frequent correlated subgraph pairs, where two subgraphs are correlated if they share similar occurrence distributions in the graph database. Our problem is significantly different and more complex than these existing works: **(1)** We consider mining over a single, large graph, while these works consider searching and mining in a graph database having several small and medium-scale graphs. **(2)** Their "correlation" measures simple co-occurrence, i.e., if two constituent subgraphs occur in the same set of graphs from the graph database. In our "correlation" computation, we need to enumerate every pair of instances of both these subgraphs in a single, large graph, and then verify their pairwise distances. Hence, our CSM problem is computationally more challenging.

• **Correlation mining in other domains.** Correlation mining has drawn extensive attention in diverse applications due to its advantages in uncovering underlying dependencies, for example, in market transactions **???**, sequence databases **?**, sequences of sets **?**, quantitative databases **?**, time series data **??**, and even in spatial domain **?**. To the best of our knowledge, our work is the first application of correlation mining in a single, large graph.

0.7 Conclusions

A large body of work exists on mining recurring structural patterns among a group of nodes in the form of frequent subgraphs. However, *can we mine recurring patterns among the frequent subgraphs itself?* In this paper, we answer this question by mining correlated pairs of frequent subgraphs. Unlike frequent subgraphs mining, we not only need to identify if a subgraph is frequent, but also enumerate, maintain, and compute distances among *all* instances of all frequent subgraphs. Managing instances imposes a severe scalability challenge on both computation and storage. We tackle this challenge by designing a data structure called *Replica*, which stores all instances in a compact manner. Furthermore, Replica allows us to design a near-optimal approximation scheme to enumerate and identify instances in a highly efficient manner. Through extensive evaluation across a series of real datasets, we demonstrate that the proposed mining algorithm *CSM* scales to million-sized networks, imparts up to 5 orders of magnitude speed-up over baseline techniques, and discovers patterns that existing techniques fail to reveal. Overall, our work initiates a new line of research by mining higher-level patterns from the pattern space itself.

For future work, we propose to mine arbitrary-sized groups of correlated subgraphs instead of being restricted to pairs.

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 $instance$

```
if |instance| = |V(R)| then
    return instance
else
    e(p, c) :− NEXTQUERYEDGE(DFS List, ...)
    P_c :− FILTERCANDIDATES(instance, c, ...)
    foreach w ∈ P_c such that w is not yet matched do
        instance ← instance ∪ {(c, w)}
        I ← I ∪ FINDALLINSTANCES(R, instance, ...)
        instance ← instance \ {(c, w)}
    return I
```

A.2: OPERATEEXACT() ;
Input: Graph G, Q, $replica(Q)$, hop h, $CorV$, $CorP$
Output: $\tau(Q, Q_k, h)$, updated Top k order

```
foreach vertex m ∈ Mappings(center, replica(Q)) do
    I ← set of all instances I such that (center, m) ∈ I
    [[Found Using FINDALLINSTANCESEXACT]]
    foreach u ∈ V(replica(Q)) constituing an instance in I do
        ∀v ∈ CorV(u), CorP(v) ← CorP(v) ∪ {Q}
        Collect(m, Q) ← Collect(m, Q) ∪ CorP(u)
foreach pattern Q_k in set operated do
    τ(Q, Q_k, h) ← |{m | m ∈ Mappings(center, replica(Q)) ∧ Q_k ∈ Collect(m, Q)}|
Update Top k order with computed correlation (τ) values
operated ← operated ∪ {Q}
```