Mining Top-*k* Pairs of Correlated Subgraphs in a Large Network

Arneish Prateek† Arijit Khan§ Akshit Goyal† Sayan Ranu†
†*Indian Institute of Technology, Delhi* §*Nanyang Technological University, Singapore*
arneish.p@gmail.com, arijit.khan@ntu.edu.sg, akshitgoyal5@gmail.com, sayanranu@iitd.ac.in

ABSTRACT

We investigate the problem of *correlated subgraphs mining* (CSM) where the goal is to identify pairs of subgraph patterns that frequently co-occur in proximity within a single graph. Correlated subgraph patterns are different from frequent subgraphs due to the flexibility in connections between constituent subgraph instances and thus, existing frequent subgraphs mining algorithms cannot be directly applied for CSM. Moreover, computing the degree of correlation between two patterns requires enumerating and finding distances between every pair of subgraph instances of both patterns - a task that is both memory-intensive as well as computationally demanding. To this end, we propose two 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 efficiency, we propose a top-k pruning strategy, while to reduce memory footprint, we develop a compressed data structure called $\mathcal{R}$eplica, which stores all instances of a subgraph pattern on demand. Our empirical results demonstrate that the proposed algorithms not only mine interesting correlations, but also achieve good scalability over large networks.

PVLDB Reference Format:
Arneish Prateek, Arijit Khan, Akshit Goyal, Sayan Ranu. Mining Top-k Pairs of Correlated Subgraphs in a Large Network. *PVLDB*, 13(9): 1511-1524, 2020.
DOI: https://doi.org/10.14778/3397230.3397245

1. INTRODUCTION

In this paper, we explore the problem of *correlated subgraphs mining* (CSM) in a single large network. In particular, we define a pair of subgraphs[1] 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 connections between constituent subgraph instances. To elaborate, in Figure 1, we highlight three regions inside the chemical structure of Taxol, an anti-cancer drug, where CCCH and O occur closely albeit connected in three different ways. For simplicity, we do not consider the edge types (i.e. single or double bonds) in this example. This figure illustrates that while CCCH and O form a frequently occurring correlated pair of subgraphs, the individual instances (for example, HCC(-O)C) may not be frequent patterns themselves. Therefore, existing frequent subgraphs mining techniques cannot mine such pairs of correlated subgraph patterns.

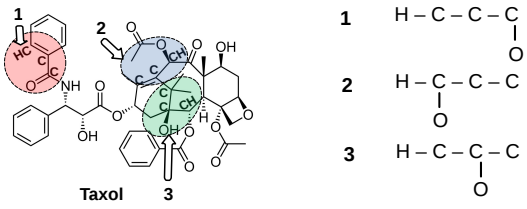


Figure 1: **Correlation between** CCCH **and** O **in Taxol, an anti-cancer drug.** CCCH **and** O **frequently occur closely but can be connected in multiple different ways.**

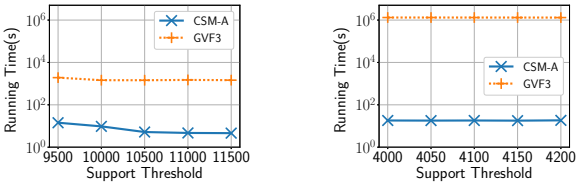
CSM will be useful for many applications. For example, it can be used in the identification of *co-operative functions in biological networks*. In a genome graph, each node represents a gene and each edge connecting two nodes denotes the interaction between the two genes. In practice, there are some combinations of dominant genes that co-occur frequently and these are more likely to express critical phenotypes of an individual. Previous studies demonstrate that pairs of dominant gene combinations can occur frequently in an individual, and that such co-occurring patterns often reflect the functionality that is needed for co-operative biochemical functioning such as chemical bonding or binding sites interactions [41]. We can use these pairs of correlated genes to predict co-operative biological functions. We produce a case study in § 5.5.

1.1 Technical Challenges and Related Work

Frequent subgraphs mining. Detecting correlated subgraphs is harder than the frequent subgraphs mining (FSM) problem [63, 28, 39, 65, 62, 12, 27, 48, 66], which already has an exponential search space (a graph with m edges can have 2^m subgraphs). Techniques have also been developed for discriminative [64, 51], statistically significant [25, 31, 53, 50, 54, 55] and representative subgraphs mining [24, 67, 46, 47, 52]. For correlated subgraphs mining (CSM), the search space is *doubly exponential* (because one needs to compute the correlation between every pair of subgraph instances). Additionally, unlike FSM, CSM neither exhibits downward closure nor upward closure (we shall demonstrate this formally in § 2.2), thereby making it difficult to directly apply *apriori*-based pruning techniques. Moreover, only mining frequent subgraphs is not sufficient for CSM. Since we call subgraph A as *correlated* to subgraph B if their *instances* are frequently located close to each other, we need to enumerate *all* instances of those

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397245

[1] Keywords *subgraph*, *pattern*, and *subgraph pattern* are used interchangeably.



(a) *MiCo* dataset (small) (b) *Citation* dataset (large)

Figure 2: **Performance of** CSM-A **against** GRAMI+VF3**.**

frequent subgraphs to compute the degree of correlation between A and B. This makes our problem more challenging from both computational and memory perspectives.

To establish this empirically, we perform frequent subgraphs mining using GRAMI [17]. For each frequent subgraph identified via GRAMI, we enumerate all its instances using the state-of-the-art VF3 algorithm [14] and finally compute the correlated pairs. Figure 2 presents the results on two datasets; the dataset description is provided in § 5.1.1. We observe that the FSM-based approach takes more than 15 days for *Citation (DBLP)*. On the other hand, the proposed approach, CSM-A, 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.

Correlated subgraphs mining in graph databases. The closest related work in the space of correlated subgraphs mining is by Ke et al. [32]. However, there are two fundamental differences:

(1) Definition of correlation: Ke et al. target the graph database scenario where there are multiple graphs: two subgraphs A and B are called correlated if the containment of A within a data graph increases the likelihood of containing B as well. In our problem, we have only one large data graph. In this graph, subgraphs A and B are defined to be correlated if the *instances* of A are frequently located in *close proximity* to the *instances* of B.

(2) Notion of proximity: Owing to the difference in the definition of correlation, there is no concept of proximity in [32]. In our problem, for each instance of a subgraph, we need to search and track instances of all other subgraphs that occur within a user-specified distance threshold. This operation is the root of the primary computational bottleneck, which does not arise in [32].

Relaxed notions of frequent subgraphs mining and search. Our problem also has similarities with various relaxed definitions for frequent subgraphs mining and search. For example, proximity patterns [36] were introduced to mine the top-k set of node labels that co-occur frequently in neighbourhoods. Correlations between node labels and dense graph structures were identified in [22, 56]. In our problem, while we allow certain flexibility in terms of how the constituent subgraph instances are connected, we still maintain fixed structures for subgraph instances.

Inexact subgraphs matching has been extensively studied [33, 58, 42, 57, 34, 35, 59, 45, 61, 19, 23, 49] (see [21] for a survey). There are several works on simulation and bisimulation-based graph pattern matching [26, 18, 43], which define relaxed subgraphs matching as a *relation* between the query and target nodes. The one-hop neighbourhood information is preserved via this relation. The CSM problem is different from graph simulation. Notice that between instances 1 and 2 in Figure 1, there exists no graph simulation: the one-hop neighbourhood information for different *C* atoms (nodes) is different in these two instances. Owing to these fundamental differences in the formulation and the resulting algorithmic challenges, we need a novel technique that is tailored for the proposed problem.

1.2 Contributions and Roadmap

(1) We formulate the problem of *correlated subgraphs mining* in a single large graph, where *correlated subgraphs* is defined as a pair of subgraph patterns that frequently co-occur in proximity within the data graph (§ 2).

(2) The key differentiating factor in our problem compared to existing subgraphs mining problems is that we not only need to identify the subgraph patterns, but also enumerate and store all its instances. This requirement imposes a huge scalability challenge on both computation and storage. We address this issue by designing a novel data structure called $\mathcal{R}$*eplica*, which stores all instances of a subgraph pattern on-demand in a 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 also discuss the novelty of the $\mathcal{R}$eplica over existing compressed structures for various graph mining tasks, as well as its potential applicability in other workloads.

(3) We further speed up the mining process by designing a near-optimal approximation algorithm (§ 4).

(4) We empirically demonstrate the effectiveness and efficiency of our methods on eight real networks while also detailing concrete case studies over biological and social networks. We establish that the proposed algorithm is up to 5 orders of magnitude faster than baseline strategies and scales to million-sized networks (§ 5).

2. PRELIMINARIES

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 $\mathbb{L}$ such that every node $v \in V$ is associated with a label, i.e., $L(v) \in \mathbb{L}$.

Definition 1 (Subgraph Isomorphism)**.** *Given a graph $G = (V, E, L)$ and a subgraph $Q = (V_Q, E_Q, L_Q)$, 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)** $\forall (v_1, v_2) \in E_Q, (M(v_1), M(v_2)) \in E$.

To quantify the frequency of a subgraph, we use the *minimum image-based (MNI) support* [13] metric.

Definition 2 (MNI Support [13])**.** *It is based on the number of unique nodes in G that a node of the pattern Q is mapped to, i.e.,*

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

MNI follows *downward closure*: the support of a supergraph $Q_1 \succeq Q$ is at most that of its subgraph Q, i.e., $\sigma(Q_1) \leq \sigma(Q)$.

Example 1. *Figure 3 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 v_1 being mapped to only one node in G (u_3) for all isomorphic mappings. The nodes in the set $\{M(v)\}$ are called the* images *of v.*

Definition 3 (Frequent Subgraphs)**.** *Given a data graph G, a definition of support σ, and a user-defined minimum support threshold Σ, the* frequent subgraphs mining (FSM) *problem identifies all subgraphs Q of G, such that $\sigma(Q) \geq \Sigma$.*

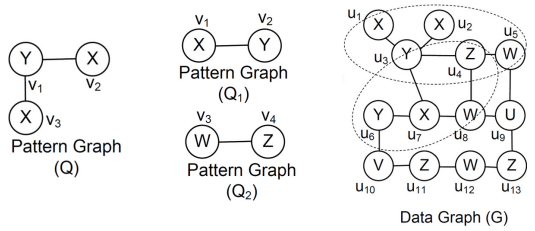


Figure 3: **Subgraph isomorphism from Q to G. Correlation between Q_1 and Q_2 in G.**

2.2 Problem Formulation

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

For consistency with MNI, we group subgraph instances.

Definition 4 (Instance Grouping)**.** *Given a data graph G, a pattern Q, and its instances in G denoted as $\mathbb{I} = \{I_1, I_2, \ldots, I_s\}$, we define by v^* the node in Q which has the minimum number of images (mappings). We denote by $\mathbb{M}(v^*) = \{M_1(v^*), M_2(v^*), \ldots, M_{\sigma(Q)}(v^*)\}$ the images of v^*. 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 having image node $M_j(v^*)$.*

Example 2. *For data graph G and pattern Q_1 in Figure 3, the instances are given by $\mathbb{I} = \{u_1u_3, u_2u_3, u_7u_3, u_7u_6\}$. However, its* MNI *support is 2, 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: $\mathbb{I}' = \{u_1u_2u_7u_3, u_7u_6\}$.*

The *distance* between two instance groups is defined as follows.

Definition 5 (Instance group distance)**.** *Let $sp(u, v)$ denote the length of the shortest path from node u to v. Then,* instance group distance *$\delta(I', J')$ between instance groups I' and J' is defined as:*

$$\delta(I', J') = \min_{\forall u \in I', \forall v \in J'} \{sp(u, v)\}$$

Definition 6 (Correlation)**.** *Let Q_1 and Q_2 be two subgraphs of data graph G with 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, we assume $\sigma(Q_1) < \sigma(Q_2)$[2]. Given a distance threshold $h \geq 0$, the* correlation *$\kappa(Q_1, Q_2, h)$ between Q_1 and Q_2 is the number of instance groups of Q_1 that occur within a distance of h from an instance group of Q_2. Mathematically,*

$$\kappa(Q_1, Q_2, h) = \sum_{\forall I' \in \mathbb{I}'} close(I', \mathbb{J}', h)$$

$$\text{where, } close(I', \mathbb{J}', h) = \begin{cases} 1 & if\ \exists J' \in \mathbb{J}',\ \delta(I', J') \leq h \\ 0 & otherwise \end{cases}$$

Note that the contribution of an instance group to the correlation count is independent of the instance group size. We adopt this approach due to two reasons. First, computing the size of an instance group, i.e., the number of unique instances within the group is *NP-hard* [13]. Furthermore, it is needed to keep the definition of correlation count consistent with the definition of MNI support.

Instead of raw counts, one may also use a *normalised* measure.

Definition 7 (Normalised correlation)**.** Normalised correlation *is (absolute) correlation normalised by support.*

$$\tilde{\kappa}(Q_1, Q_2, h) = \frac{\kappa(Q_1, Q_2, h)}{min\{\sigma(Q_1), \sigma(Q_2)\}}$$

For brevity, hereon, we assume Definition 6 for correlation value. Both the absolute and normalised correlation functions are *symmetric*, i.e., $\kappa(Q_1, Q_2, h) = \kappa(Q_2, Q_1, h)$ and $\tilde{\kappa}(Q_1, Q_2, h) = \tilde{\kappa}(Q_2, Q_1, h)$. Furthermore, the correlation of a pair can only increase with h, since any pair that satisfies the distance constraint for $h = x$ will also satisfy for $h = x + \Delta$, for any $\Delta \geq 0$.

Example 3. *Consider patterns Q_1 and Q_2 and graph G in Figure 3. 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 $v_2 \in V_{Q_1}$ in G. Similarly, the instance groups of Q_2 are given by $\mathbb{J}' = \{u_5u_4, u_8u_4, u_{11}u_{12}u_{13}\}$, with the groupings performed based on the images of node $v_3 \in V_{Q_2}$. Assuming $h = 1$, since $\sigma(Q_1) = 2 < \sigma(Q_2) = 3$, we count how many of the two instance groups of Q_1 exist within $h = 1$ hop of at least one instance group of Q_2 to get correlation between Q_1 and Q_2. For G, this gives us $\kappa(Q_1, Q_2, h = 1) = 2$ and $\tilde{\kappa}(Q_1, Q_2, h) = 2/\min\{2, 3\} = 1$.*

Problem 1 (Top-k Correlated Subgraphs Mining)**.** *Given a data graph G, a user-defined distance threshold $h \geq 0$ and a minimum support threshold Σ, find the* top-k *pairs of subgraph patterns $\langle Q_1, Q_2 \rangle$ of G having the highest correlation $\kappa(Q_1, Q_2, h)$, such that $\forall Q_1,\ Q_2$, $\sigma(Q_1) \geq \Sigma$, $\sigma(Q_2) \geq \Sigma$.*

Sanity constraint. We do not consider two patterns as correlated if one of the following conditions holds. **(i)** If Q_1 is a subgraph of Q_2 (or vice-versa), the correlation is implicit. **(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 $Q_3 \succeq Q_2$, the pair $\langle Q_1, Q_2 \rangle$ is not interesting. In our solution framework, we devise a technique to eliminate such subgraph pairs from the top-k results.

2.2.1 Parameters

There are three input parameters: k, distance threshold h, and support threshold Σ. k is easy to set and depends on how many correlated pairs one wants to examine.

The value of h dictates when we call two subgraph instances to be in close proximity. h could be set either by domain scientists based on domain knowledge or based on *statistical significance*. In the statistical significance mode, we term two instances to be close if the distance between them is significantly smaller than two randomly picked subgraph instances. To formalise this intuition, we compute the distribution of distances between random pairs of subgraph instances. h can be set such that the *probability* of $p(sp(g_1, g_2) \leq h) \leq \theta$, where θ is a probability threshold. This formulation computes the p-value.

We employ Σ to classify a subgraph as frequent. One heuristic to set Σ is to set it as $X\%$ of the support of the most frequent node label in the dataset, since MNI support of any subgraph is upper bounded by the support of the most frequent node label.

2.3 Theoretical Characterisation

Lemma 1. *$\kappa(Q_1, Q_2, h)$ does not have downward closure property. Specifically, consider a subgraph of Q_2, say Q_3 (i.e., $Q_2 \succeq Q_3$). Then, $\kappa(Q_1, Q_3, h) \geq \kappa(Q_1, Q_2, h)$ does not always hold.*

[2] If $\sigma(Q_1) = \sigma(Q_2)$, then for tie-breaking, we accord lower support to the pattern with the lower *minimum DFS code* [65].

The downward closure property does not hold because as one grows the pattern (say from Q_3 to Q_2), the supergraph (Q_2) may now map to larger-sized instances in the data graph that could result in additional spatial proximity to instances of the other pattern (Q_1) and consequently, a higher correlation. For example, in Figure 3, consider Q_3 as a single node with label W; patterns Q_1 and Q_2 are as shown. Clearly, $Q_2 \succeq Q_3$. We notice that $\kappa(Q_1, Q_3, h = 1) = 1$, however $\kappa(Q_1, Q_2, h = 1) = 2$.

Lemma 2. *Correlation metric $\kappa(Q_1, Q_2, h)$ does not have upward closure property. Specifically, consider $Q_4 \succeq Q_1$. Then, $\kappa(Q_1, Q_2, h) \leq \kappa(Q_4, Q_2, h)$ does not always hold.*

The upward closure property does not hold because as one grows the pattern (say from Q_1 to Q_4), the supergraph (Q_4) may now have lower MNI support (due to downward closure property of MNI support) that could result in a decrease in the count of instance-groups remaining in proximity to instances of the other pattern (Q_1) and consequently, a lower correlation. For example, in Figure 3, consider Q_4 as consisting of three nodes X-Y-X; patterns Q_1 and Q_2 are as shown. Clearly, $Q_4 \succeq Q_1$. We notice that $\kappa(Q_1, Q_2, h = 1) = 2$, however $\kappa(Q_4, Q_2, h = 1) = 1$.

Since downward and upward closure properties are commonly used in frequent pattern mining [10, 39, 13, 20, 40, 65], in particular for enabling early termination of mining algorithms, Lemmas 1 and 2 demonstrate the inherent complexity of our problem. Fortunately, however, Lemma 3 below provides a bound on κ that enables the design of early termination criteria for our algorithm.

Lemma 3. *Correlation $\kappa(Q_1, Q_2, h)$ between a pair $\langle Q_1, Q_2 \rangle$ cannot exceed the MNI support of either Q_1 or Q_2. That is,*

$$\kappa(Q_1, Q_2, h) \leq \min\{\sigma(Q_1), \sigma(Q_2)\}$$

Lemma 3 directly follows from Definition 6, which states that κ is computed from the instance groups of the pattern having lower support. Therefore, κ cannot exceed MNI support of the less frequent pattern. Using the same reasoning, the normalised correlation value $\tilde{\kappa}$ is bounded between 0 and 1.

3. EXACT MINING: CSM-E

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

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 m-th level vertices[3] represent m-edged frequent subgraphs. The first level of T 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* [65]. Thus, in a frequent subgraph in T, $i < j$ indicates that node v_i is discovered before v_j. An edge from v_i to v_j with $i < j$ is called a *forward edge*; if $i > j$, the edge is termed *backward edge*. We call v_0 the *root* and v_n the *rightmost vertex*, where $n + 1$ is the number of nodes in the subgraph. The direct path (consisting of forward edges only) from v_0 to v_n is called the *rightmost path*.

To construct the $(m + 1)$-th layer of T from its m-th layer, a vertex in the m-th layer (i.e., a frequent subgraph), say P, is augmented with a new edge thus creating a new subgraph, say C. We insert C as a vertex in the $(m + 1)$-th layer of T if and only if C is determined to be a frequent subgraph in G. We refer to P as a *parent vertex* of C and C as its *child vertex* in T. Note that C, having $(m + 1)$ edges, can be generated from upto $m + 1$ parent vertices in T. The generation of *duplicate graphs* in this manner is redundant and affects the overall efficiency. To reduce the creation of duplicate graphs, we take inspiration from the *rightmost extension* strategy [65]. In particular, **(1)** forward edges can grow from nodes along the rightmost path in P, while **(2)** backward edges can only grow from the rightmost node v_n of P.

While the aforementioned search tree based mining strategy follows classical frequent graph pattern mining [65], in the following we shall introduce our three novel technical contributions: **(1)** the best-first exploration algorithm, **(2)** early-termination criteria, and **(3)** a memory-efficient structure, $\mathcal{R}$eplica, for storing all instances of a frequent pattern and thus efficiently computing correlation.

In particular, the search tree T is not fully materialised. Instead, it is built on demand. The exploration strategy varies across different graph mining algorithms. For instance, GSPAN explores T in a depth-first manner [65], while some other apriori-based approaches such as AGM [28] and FSG [39] construct T in a breadth-first manner. However, we find that for solving the **Top-k Correlated Subgraphs Mining** problem, we can perform search more efficiently than both depth-first and breadth-first exploration strategies. In § 3.1.1, we propose our novel *best-first exploration* algorithm.

Next, when a new frequent pattern C is added in T, we compute C's correlation with all previously explored vertices in T (i.e., frequent subgraphs of G that have been discovered prior to C). In § 3.2, we discuss details of the memory-efficient $\mathcal{R}$eplica structure for computing correlation.

Finally, we employ a *priority queue* $\mathcal{Q}$ for storing and updating the top-k highest correlation values and the corresponding pairs, as they are found. In § 3.1.2, 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.

3.1.1 Best-First Exploration

A pair of subgraph patterns having individual higher support values can be expected to have higher correlation between them. This is because 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 a *best-first* exploration of the pattern search tree. Let $\mathbb{P}$ denote the set of frequent patterns currently in T and $\mathbb{C}$ denote 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[4], while also removing it from $\mathbb{C}$. We implement $\mathbb{C}$ as another *priority queue* (referred to as the *search queue* in our algorithm) to easily extract C^* from $\mathbb{C}$. Formally,

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

We notice that unlike GSPAN, a depth-first exploration of the pattern search tree is not optimal since it identifies many subgraph-supergraph pairs at earlier stages, which are not useful for the CSM problem. On the other hand, breadth-first exploration is more memory intensive and based on empirical results reported in § 5, it is still less efficient compared to our proposed best-first strategy.

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

[4] If multiple patterns in $\mathbb{C}$ have the same maximum support, we break the tie based on *minimum DFS code* [65] of these patterns.

Pruning duplicate subgraphs. Following the rightmost extension rules reduces the generation of duplicate graphs but the problem is not entirely eliminated. To avoid processing a duplicate pattern, we take advantage of the *minimum DFS code* [65]. We use a hash table H to store the minimum DFS codes for all vertices (i.e. frequent subgraphs) already in T. Whenever a subgraph C^* is selected from $\mathbb{C}$, we compute its minimum DFS code, denoted by $MinDFS(C^*)$ and perform a lookup in H. If $MinDFS(C^*)$ is found, then C^* must have been discovered previously since two graphs have the same minimum DFS code if and only if they are isomorphic [65], in which case, we do not insert C^* in T.

The *completeness guarantee* of generating all frequent subgraphs via *rightmost extension* along with the aforementioned pruning strategy for eliminating duplicate patterns directly follows from guarantees proven in GSPAN; we omit the proof for brevity.

Eliminating subgraph-supergraph pairs. To avoid calculating correlations between subgraph-supergraph pairs, we perform subgraph isomorphism checks between C^* and every frequent pattern Q in T and eliminate all pairs where such a relationship exists.

3.1.2 Early Termination Criteria

Our objective is to mine the top-k pairs of correlated (frequent) subgraphs, and ensure that no other pair has a higher correlation value κ than any pair in our top-k priority queue $\mathcal{Q}$. Lemma 3 (§ 2.3) allows us to deduce the following. Assume, C^* is the current frequent subgraph selected from $\mathbb{C}$. By Lemma 3, $\kappa(C^*, Q, h) \leq \sigma(C^*)$, 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 at any stage, we have $\sigma(C^*)$ lower than the *least* correlation value 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, $\mathcal{Q}$ would not be full and yet there would not exist any more frequent subgraphs to include in T. Thus, we terminate our algorithm.

Eliminating pairs with high correlation only due to a frequent supergraph. This case arises for a subgraph pair $\langle Q_1, Q_2 \rangle$ in priority queue $\mathcal{Q}$, where $C^* \succeq Q_1$ and $C^* \succeq Q_2$, only if C^* has the same support as the correlation between Q_1 and Q_2. Since $\sigma(C^*) = \kappa(Q_1, Q_2, h)$, and $\langle Q_1, Q_2 \rangle$ is in $\mathcal{Q}$, the termination criteria of our algorithm will not be satisfied. To remove $\langle Q_1, Q_2 \rangle$ from $\mathcal{Q}$, we incorporate the following procedure in our mining algorithm: whenever a new pattern C^* is extracted from the search queue $\mathbb{C}$ and the termination criteria remains unsatisfied, we check for the existence of a correlated pair $\langle Q_1, Q_2 \rangle$ 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 $\langle Q_1, Q_2 \rangle$ and the associated correlation value $\kappa(Q_1, Q_2, h)$ from $\mathcal{Q}$.

3.1.3 Putting Everything Together

Algorithm 1 describes our pipeline. It begins by 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 remains unsatisfied (Lines 5-18). C^*, selected as the best-first choice from $\mathbb{C}$, is processed for correlation (§ 3.2.4) with every other previously explored pattern Q in search tree T subject to satisfaction of constraints described in § 3.1.1, 3.1.2 (Lines 6-13). Top-k priority queue $\mathcal{Q}$ is updated based on the computed κ values (Line 14). C^* is then inserted in T and its minimum DFS Code in H (Line 15), followed by the extension of C^* to generate its *child* patterns (Line 16). All frequent children of C^* are queued in $\mathbb{C}$ for processing (Lines 17-18) and the loop continues. Upon termination, the algorithm returns top-k pairs of correlated subgraphs (Line 19).

Algorithm 1: MINING TOP-k CORRELATED PAIRS

```
   Input: data graph G = (V, E, L), parameters Σ, h ≥ 0, k
   Output: Top-k pairs of correlated patterns s.t. each pattern has support ≥ Σ
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 size-bound k
5  while ℂ is non-empty do // Search
6      Pattern C* ← BEST-FIRST-POP(ℂ) (§ 3.1.1)
7      if TERMINATION CONDITION (§ 3.1.2) 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) (§ 3.2)
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 (§ 3.1)
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
```

3.2 Storing Subgraph Instances

Computing correlation κ between two subgraph patterns requires enumerating and finding distances between every pair of instances for both these patterns. This is a challenging problem due to the following reasons: **(1)** storing all instances of all frequent subgraphs explored by our algorithm can easily overwhelm the memory. Note that frequent subgraphs mining algorithms such as GSPAN [65] and GRAMI [17] do *not* require all instances of all discovered frequent subgraphs to be stored. In particular, GRAMI employs a constraint satisfaction problem (CSP) that only finds the *minimal* set of instances needed to meet the frequency threshold and avoids the costly enumeration of *all* instances. Therefore, such a requirement for CSM presents an additional challenge. **(2)** Moreover, many redundant distance computations could take place. For example, assume instances I_1 and I_2, corresponding to patterns Q_1 and Q_2 respectively, occur in data graph G within h-hops. Consider a supergraph $Q_3 \succeq Q_1$ and its instance I_3. Assume that $I_3 \succeq I_1$ in G. To compute correlation between Q_3 and Q_2, evaluating the distance between their corresponding instance pair I_3 and I_2 is redundant since it is guaranteed that they would also exist within h-hops of each other.

To address both these challenges, we propose an efficient data structure called $\mathcal{R}$eplica, defined as follows.

3.2.1 The Replica Data Structure

Given a data graph G and pattern $Q = (V_Q, E_Q, L)$, the corresponding $\mathcal{R}\text{eplica}(Q) = (V_{\mathcal{R}(Q)}, E_{\mathcal{R}(Q)}, L)$ is a subgraph of G, constructed by the *graph-union*[5] of all instances of Q in G. Since $\mathcal{R}\text{eplica}(Q)$ is a subgraph, it is stored as an adjacency list.

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

[5]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 $\mathcal{G} = (V_1 \cup V_2, E_1 \cup E_2, L)$.

Forward Node Mapping. $\forall u \in V_Q$, we define a *forward* node mapping $\mathbb{M}_{\mathcal{R}(Q)}(u)$ that stores the set of 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. $\forall v \in V_{\mathcal{R}(Q)}$, we define an *inverse* node mapping $\mathbb{M}_Q^{-1}(v)$ that stores the set of all nodes $u \in V_Q$ such that $v \in \mathbb{M}_{\mathcal{R}(Q)}(u)$.

$\mathcal{R}$eplica provides a middle ground between two extreme strategies, i.e. **(1)** explicitly *storing* all instances of all generated patterns [40], or, **(2)** *enumerating* all instances of a pattern on-demand (during runtime) in the entire data graph G. *In contrast to these alternatives, the* $\mathcal{R}$eplica *is economical from the viewpoints of both storage and efficiency. It not only avoids a potential out of memory (OOM) state 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 § 3.2.4.*

Note that both Forward and Inverse Node Mappings can be deduced from the $\mathcal{R}\text{eplica}(Q)$ subgraph via subgraph isomorphism with pattern Q. However, we explicitly maintain these mappings for computational efficiency since Forward Node Mapping allows us to readily compute MNI support $\sigma(Q)$ and also assists in $\mathcal{R}$eplica generation, while the Inverse Node Mapping enables us to quickly prune candidate nodes of $\mathcal{R}\text{eplica}(Q)$ for matching with a node in Q while performing subgraph isomorphism (§ 3.2.2).

3.2.2 Building On-Demand Replica for New Patterns

We create $\mathcal{R}$eplicas in an incremental manner. Given $\mathcal{R}\text{eplica}(Q)$ of a parent pattern Q, we generate $\mathcal{R}\text{eplica}(R)$ of its child pattern R. Our approach for *exact* $\mathcal{R}$eplica extension is given in Algorithm 2, which essentially describes a backtracking procedure for subgraph isomorphism similar to the *Ullman's algorithm* [60].

Algorithm 2 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 $\mathcal{R}\text{eplica}(Q)$, 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 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 3) which uses $\mathcal{R}\text{eplica}(Q)$ to find every instance of R such that $u \mapsto u'$ and $v \mapsto v'$ (Line 7).

Algorithm 2: BUILD REPLICA FOR A NEW PATTERN

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

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** $u' \in \mathbb{M}_{\mathcal{R}(Q)}(u)$ **do**
4 $instance$.add$(u \mapsto u')$
5 **foreach** edge $(u', v') \in E$ that maps to the extending edge $(u, v) \in E_R$ **do**
6 $instance$.add$(v \mapsto v')$
7 $\mathbb{I} \leftarrow$ FINDALLINSTANCES$(G, Q, \mathcal{R}eplica(Q), R, DFS\ List(Q), instance, \mathbb{I})$
8 UPDATEREPLICA$(R, \mathcal{R}eplica(R), \mathbb{I})$
9 $instance$.delete$(v \mapsto v')$
10 $instance$.delete$(u \mapsto u')$
11 **return** $\mathcal{R}\text{eplica}(R)$

Algorithm 3: FINDALLINSTANCES: EXACT METHOD

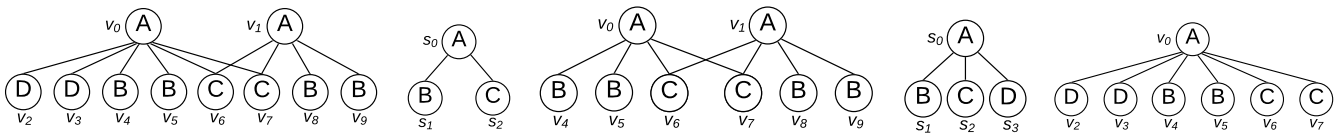
Input: data graph $G = (V, E, L)$, parent pattern $Q = (V_Q, E_Q, L)$, $\mathcal{R}\text{eplica}(Q) = (V_{\mathcal{R}(Q)}, E_{\mathcal{R}(Q)}, L)$, child pattern $R = (V_R, E_R, L)$, $DFS\ List(Q)$, partial isomorphism of R: $instance$, $\mathbb{I}$
Output: $\mathbb{I}$: set of all instances of R in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

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

Algorithm 3, 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 $p' \in V_{\mathcal{R}(Q)}$. (The first call to Algorithm 3 in every iteration of the inner loop in Algorithm 2 always has p matched with u', i.e., $instance(p) = u'$ since $p = u$). If e is a backward edge, c is already matched in which case the algorithm checks whether an edge exists in $\mathcal{R}\text{eplica}(Q)$ connecting p' 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 candidate nodes $c' \in V_{\mathcal{R}(Q)}$ for matching c such that: **(1)** c' is a neighbor of p' in $\mathcal{R}\text{eplica(Q)}$, and, **(2)** c exists in the Inverse Node Mapping set for c' in $\mathcal{R}\text{eplica(Q)}$, i.e., $c \in \mathbb{M}_Q^{-1}(c')$; Next, for every node $c' \in P_c$ such that c' has not *already* been matched in the current *instance*, the algorithm attempts the match $c \mapsto c'$ 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 after matching every edge in $DFS\ List(Q)$, which it returns.

The set of all instances $\mathbb{I}$ thus found is returned by Algorithm 3 (Line 17) and recorded by Algorithm 2. In UPDATEREPLICA (Line 8, Algorithm 2), the algorithm updates $\mathcal{R}\text{eplica}(R)$ by performing a graph union with all the mined instances in $\mathbb{I}$. It also updates the $\mathbb{M}_{\mathcal{R}(R)}$ (Forward) and $\mathbb{M}_R^{-1}$ (Inverse) Node Mapping indices to record new mappings found for every $instance$ in $\mathbb{I}$. Thus, we obtain the $\mathcal{R}$eplica of a child pattern using the $\mathcal{R}$eplica of its parent.

Example 4. *Consider data graph G, pattern Q and the corresponding* $\mathcal{R}eplica(Q)$ *as shown in Figures 4a, 4b and 4c respectively. Pattern Q is extended at the* extending node s_0 *using the* extending edge (s_0, s_3) *to generate child R, shown in Figure 4d. Assume* DFS List *for* DFS(Q) *starting at* root s_0 *records edges* (s_0, s_1) *and* (s_0, s_2) *in order. To construct* $\mathcal{R}eplica(R)$*, the algorithm iterates over* $\mathbb{M}_{\mathcal{R}(Q)}(s_0)$*, i.e., the set* $\{v_0, v_1\}$*. With* $s_0 \mapsto$



(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 using the $\mathcal{R}$eplica of its parent pattern**

v_0, the algorithm then considers nodes in G for matching the newly extended node s_3. *Clearly, the* candidate nodes *for matching s_3 are v_2 and v_3. Next, for each of $s_3 \mapsto \{v_2, v_3\}$, the algorithm makes recursive invocations to enumerate valid mappings for nodes s_1 and s_2 following the* DFS List*, thus successfully enumerating instances of R. With $s_0 \mapsto v_1$, however, no mappings exist for matching s_3 in G. Graph union of all instances of R thus enumerated results in $\mathcal{R}eplica(R)$ as depicted in Figure 4e.*

3.2.3 Replica Deletion for Parent Patterns

We build and maintain the following two distance-based *global* indices, which allow the deletion of $\mathcal{R}$eplicas of parent patterns once all their frequent child patterns (generated via rightmost extension) have been included in the pattern search tree T.

Proximate-Nodes Index. For each frequent node u in graph G, we store the set $CorV(u) = \{v \in V \mid sp(u, v) \leq h,\ v \text{ is frequent}\}$. We store this information as a *hashmap* where the key is the node ID and the value is the $CorV$ set for that node. This index is constructed only once before our mining process starts by performing a breadth-first search from each frequent node.

The memory consumption significantly depends on the distance threshold h. As h increases, more nodes satisfy the distance constraint and therefore need to be stored in $CorV(u)$. While in our empirical evaluation, we never encounter the case where the size of this index exceeds the main memory capacity, we discuss two strategies to handle such a situation should it arise. First, if the fraction of nodes satisfying the distance constraint is above 50% on average, which is typically the case for $h \geq 3$ (see Figure 7a), then one may instead store the complement set, i.e. nodes that do *not* satisfy the distance constraint. This set encodes the same information while being smaller in size. Second, one could adopt a *hybrid* approach. Specifically, for each frequent node u, we can store all frequent nodes within a distance threshold $h' < h$ such that this information can be stored in main memory. To compute if $sp(u, v) \leq h$, we first fetch the set of nodes $\mathbb{U}_1$ within h' from u in constant time. If $v \notin \mathbb{U}_1$, we recursively fetch $\mathbb{U}_2 = \{u_2 \mid sp(u_1, u_2) \leq h', u_1 \in \mathbb{U}_1\}$ and check if $v \in \mathbb{U}_2$. This process continues iteratively for i iterations till $i \times h'$ exceeds h.

Proximate-Patterns Index. For each frequent node u in data graph G, this index, denoted by $CorP(u)$, stores the set of all frequent patterns that are already included in T and whose instance(s) exist within h distance of u. Mathematically, $\forall Q \in CorP(u)$, **(1)** $Q \in T$, and **(2)** given the instance groups $\mathbb{I}' = \{I'_1, I'_2, \ldots, I'_{\sigma(Q)}\}$ of Q, $\exists I' \in \mathbb{I}'$, $\exists v \in I'$ such that $sp(u, v) \leq h$. $CorP$ index is incrementally updated each time a new pattern Q is inserted in T. The details of $CorP$ maintenance will be discussed in § 3.2.4.

3.2.4 Correlation Computation

We recall that when a pattern C^* is extracted from the search queue $\mathbb{C}$, 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 whether for every instance group I' of C^*, there exists an instance group of Q within distance h of I'.

First, to find all instance groups of C^* in data graph G, our novel data structure $\mathcal{R}eplica(C^*)$ can be immediately used. The Forward Node Mapping $\mathbb{M}_{\mathcal{R}(C^*)}$ allows us to readily compute the MNI support of C^*, and also the node v in C^* having the minimum MNI support. For every node $v' \in \mathbb{M}_{\mathcal{R}(C^*)}(v)$, we enumerate from $\mathcal{R}eplica(C^*)$ all instances of C^* having $v \mapsto v'$ (Algorithm 3). This generates the instance group of C^* where $v \mapsto v'$. In this way, we efficiently enumerate *all* instance groups of C^*.

Given an instance group I' of C^*, we evaluate if any instance group of Q exists within distance h of I'. Consider $CorP(w)$ for each node $w \in I'$. *If $\cup_{w \in I'} CorP(w)$ contains Q, it means that there exists some instance group of Q within distance h of I'.* Finally, we count, out of all $\sigma(C^*)$ instance groups of C^*, how many instance groups are in proximity to at least one instance group of Q. This count is reported as the correlation $\kappa(C^*, Q, h)$. In this way, the *Proximate-Patterns Index* ($CorP$) aids in efficient correlation computation.

Incrementally Updating Proximate-Patterns Index. For each node $w \in I'$, if $CorV(w)$ contains node u, we include the new pattern C^* in $CorP(u)$. *In other words, for every node $u \in \cup_{w \in I'} CorV(w)$, the Proximate-Patterns Index $CorP(u)$ is updated to include pattern C^*.* Thus, we ensure that at any point in our algorithm, $CorP(u)$ would contain all patterns in T that are within distance h of u.

3.2.5 Novelty and Usefulness of Replica

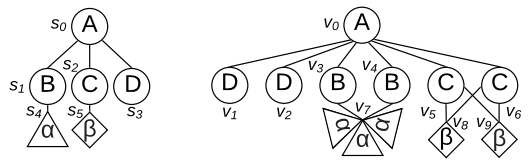
We explained in § 3.2 the usefulness of $\mathcal{R}$eplica for CSM. We conclude this section by briefly discussing its novelty in regards to existing compression methods for graph mining, as well as pointing out other workloads where $\mathcal{R}$eplica could be beneficial.

Given a graph database consisting of multiple graphs, Chen et al. developed Summarize-Mine [15] that first summarises the original graphs, which are then mined for frequent patterns. In contrast, Cook et al. [16] and Maneth et al. [44] recursively replace frequent *substructures* in graphs, which minimises the minimum description length (MDL) cost, with a meta-node. A pattern that compresses a large portion of most graphs in the mining set, but that occurs less frequently, could be preferred over a less complex but more frequent pattern. $\mathcal{R}$eplica has a different objective: it enumerates all instances of a frequent pattern for a single large graph in a compressed and efficient manner.

We envision that $\mathcal{R}$eplica could also be useful in interactive and iterative search and exploration scenarios [30, 29], for instance, when a user reformulates a past query by adding more constraints. Since $\mathcal{R}$eplica enables an efficient computation of instances of a supergraph pattern extended from its parent (§ 3.2.2), it could be beneficial in such interactive query processing workloads.

3.3 Complexity Analysis

Consider data graph G with n nodes and m edges of which n_f nodes and m_f edges are frequent based on the input minimum sup-



(a) pattern R' (b) $\mathcal{R}$eplica(R')

Figure 5: **Repeated traversals during subgraph isomorphism**

port threshold Σ. We denote by n_h and m_h, the maximum number of nodes and edges in the h-hop neighbourhood of any node in G, respectively. 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 $\mathbb{C}$, during the execution of our algorithm. Let the maximum number of nodes and edges in any pattern Q mined by our method be n_Q and m_Q and those in a $\mathcal{R}$eplica be $n_\mathcal{R}$ and $m_\mathcal{R}$ respectively. Also, assume the maximum size of Forward Node Mapping Set is M for some node in a frequent pattern mined by our algorithm.

Time Complexity. Consider $\mathcal{R}$eplica generation for a new pattern Q (§ 3.2.2). Building $\mathcal{R}$eplica(Q) requires performing subgraph isomorphism for Q in G. Hence, the associated time complexity is $\mathcal{O}(M^{n_Q})$. Next, the construction of the Proximate-Nodes Index (§ 3.2.3) requires h-hop BFS from every frequent node in G, hence it has a time complexity of $\mathcal{O}(n_f(n_h + m_h))$. In contrast, the Proximate-Patterns Index is updated whenever a new frequent pattern is mined. It requires traversing all nodes in the corresponding $\mathcal{R}$eplica subgraph and accessing their $CorV$ indices, costing $\mathcal{O}(n_\mathcal{R} n_f)$ time. Since T has at most n_T vertices, the total complexity of updating $CorP$ is $\mathcal{O}(n_\mathcal{R} n_f n_T)$. Finally, the complexity of correlation computation (§ 3.2.4) between a new pattern Q and any existing pattern in T is essentially determined by the cost of performing subgraph isomorphism for Q in $\mathcal{R}$eplica(Q), i.e., $\mathcal{O}(M^{n_Q})$. Since there can be $\mathcal{O}(n_T^2)$ correlation computations, the total time complexity is $\mathcal{O}(n_T^2 M^{n_Q})$. Overall, the time complexity of CSM-E[6] is $\mathcal{O}(M^{n_Q} n_T^2 + n_\mathcal{R} n_f n_T + n_f(n_h + m_h))$.

Incremental Top-k Computation. Assume that a user starts with a low value for k and increases k incrementally based on how many more pairs they want to explore. It is desirable that the computation is not restarted. Rather, the information already mined should be reused and any additional computation should only be performed to identify additional pairs. Our strategy supports this incremental computation mode as the answer set of top-k is a proper subset of top-$(k+1)$. Following the previous discussion, the complexity of computing the $(k+1)$-th correlated pair, assuming that the top-k pairs were already mined, is: $\mathcal{O}(M^{n_Q} + n_\mathcal{R} n_f)$.

Space Complexity. The space complexity is bounded by the size of $\mathcal{R}$eplica structures and all indices i.e., $\mathbb{M}_{\mathcal{R}(\mathbb{Q})}$ and $\mathbb{M}_Q^{-1}$ Mappings (corresponding to each $\mathcal{R}$eplica), as well as $CorV$ and $CorP$ global indices. Since we only store the $\mathcal{R}$eplica of $\mathcal{O}(n'_T)$ patterns, this requires $\mathcal{O}((n_\mathcal{R} + m_\mathcal{R})n'_T)$ space. Both $\mathbb{M}_{\mathcal{R}(\mathbb{Q})}$ and $\mathbb{M}_Q^{-1}$ mappings require $\mathcal{O}(n_Q M)$ memory. Finally, $CorV$ and $CorP$ require $\mathcal{O}(n_f^2)$ and $\mathcal{O}(n_f n_T)$ space respectively. The overall complexity is $\mathcal{O}((n_\mathcal{R} + m_\mathcal{R} + n_Q M)n'_T + n_f^2 + n_f n_T)$.

4. APPROXIMATE MINING: CSM-A

The $\mathcal{R}$eplica construction algorithm (§ 3.2.2) for a subgraph 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 3 might not scale well. For better scalability, we develop a near-optimal approximation algorithm that skips enumerating instances that are likely *redundant* (§ 4.1).

4.1 Identifying Redundancy

In many cases, to construct 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 Example 4 (Figure 4). To build $\mathcal{R}$eplica(R) for Q's child pattern 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 enumeration scheme is exactly repeated. Here, we observe that e_2 is *symmetric* to e_1 since both of them share the extending node v_0 and both v_2 and v_3 map to s_3. Consequently, any instance mapping that is applicable for e_1 is also likely to be applicable for e_2. Thus, enumeration of *all* instances may be redundant.

The impact of redundant computations in symmetric extensions can be further appreciated from the following example. In Figure 5, R' is a subgraph pattern and $\mathcal{R}$eplica(R') is its $\mathcal{R}$eplica subgraph. As in Example 4, we are again considering edge (s_0, s_3) as the extending edge. However, unlike Figure 4 where s_1 and s_2 are leaf nodes, here two arbitrary subgraphs α and β are attached at s_1 and s_2 respectively. To construct $\mathcal{R}$eplica(R'), the exact algorithm would enumerate every instance of R', which means that the same three mappings of α in $\mathcal{R}$eplica(R') would be visited multiple times through nodes v_3 and v_4 for the two extending edge mappings $(s_0, s_3) \mapsto e_1 = (v_0, v_1)$, followed by $(s_0, s_3) \mapsto e_2 = (v_0, v_2)$; worse, for each α, both the mappings of β would be enumerated twice — once each through nodes v_5 and v_6. Clearly, these repeated traversals over the *same* portion of the graph do not yield new information. These redundant traversals can be avoided if we have the ability to identify symmetric edges in the $\mathcal{R}$eplica. For instance, edges e_1 and e_2 in $\mathcal{R}$eplica(R') are symmetric and hence while considering the second extending edge e_2, we can simply *reuse* the instance enumerations mined with the first edge e_1. Moreover, while enumerating the instances corresponding to $(s_0, s_3) \mapsto e_1$, we observe that edges (v_3, v_7) and (v_4, v_7) are symmetric and therefore enumerating the three α subgraphs twice can be avoided by *reusing* the instance mappings obtained with e_1, for instances mining with e_2. The same applies to edges (v_5, v_8) and (v_5, v_9) as well. Armed with this intuition, our objective, therefore, is as follows: (**1**) identify if two or more edges in a $\mathcal{R}$eplica are symmetric to each other, and, (**2**) for a group of symmetric edges, enumerate all instance mappings for only one edge from the group and reuse these mappings for remaining symmetric edges.

4.2 Algorithm

Definition 8 (Symmetric Edges)**.** *Given pattern Q and $\mathcal{R}$eplica(Q), edges $e_1 = (v_a, v_b) \in E_{\mathcal{R}(Q)}$ and $e_2 = (v_c, v_d) \in E_{\mathcal{R}(Q)}$ are defined to be* symmetric *if (**1**) $a = c$, and (**2**) both v_b and v_d map to the same node in pattern Q.*

We now describe the approximation algorithm called CSM-A (Correlated Subgraphs Mining - Approximate), which uses symmetric edges to reduce repeated traversals. Recall, $\mathcal{R}$eplica edges are mapped to pattern edges following the DFS order. While processing a $\mathcal{R}$eplica edge following this order, we check if it is symmetric to one of the already enumerated $\mathcal{R}$eplica edges (for the

[6]We neglect the times required for maintaining search queue $\mathbb{C}$, verifying supergraph-subgraph relationships between Q and existing patterns in T, computing minimum DFS codes and rightmost extensions due to relatively smaller sizes of frequent patterns mined by our algorithm.

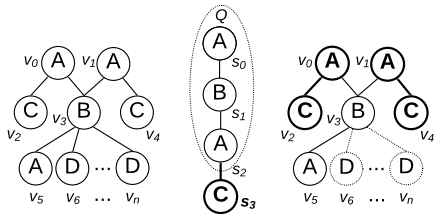


Figure 6: $\mathcal{R}$**eplica generation. (a) Data graph** G**. (b) Subgraph** R **extended from** Q**. (c) Extending edge mappings in** G**.**

same pattern edge). If it is not, the algorithm proceeds in exactly the same manner as in CSM-E (Algorithm 3) but for one difference: now, all the newly mined $\mathcal{R}$eplica edges are indexed and stored. In contrast, if the candidate $\mathcal{R}$eplica edge is determined to be symmetric, we do not recompute all mappings. Rather, we reuse the $\mathcal{R}$eplica mappings that have been indexed in a previously enumerated symmetric edge and among those mappings check if there exists at least one instance. If one such instance is found, this candidate edge is also added to the $\mathcal{R}$eplica graph and indexed. Thus, we reduce repeated traversals over explored mappings.

Example 5. *Consider data graph G and patterns Q and R as shown in Figure 6. With* extending edge (s_2, s_3)*, pattern R is extended from parent Q. To generate $\mathcal{R}eplica(R)$ from $\mathcal{R}eplica(Q)$,* CSM-A *attempts to enumerate instances of R in G by first mapping extending edge $(s_2, s_3) \mapsto (v_0, v_2)$. Following this mapping,* CSM-A*, like* CSM-E*, mines all isomorphisms by mapping $(s_2, s_1) \mapsto (v_0, v_3)$ and $(s_1, s_0) \mapsto (v_3, v_5)$ and (v_3, v_1), in* DFS *order. Thus, nodes v_5 and v_1 are considered valid mappings for s_0. Next, when* CSM-A *attempts to enumerate instances after mapping $(s_2, s_3) \mapsto (v_1, v_4)$, it recognises the symmetricity between edges (v_1, v_3) and (v_0, v_3) and so in order to map edge (s_1, s_0), it only traverses the already* confirmed mappings *set $\{(v_3, v_5), (v_3, v_1)\}$ instead of all the edges adjacent to v_3 again. Furthermore, even while traversing the set of confirmed mappings, we stop as soon as one instance is found, which further reduces the computation cost.*

4.3 Properties

To understand the approximation in the above algorithm, let us revisit Example 5. While enumerating instances with the extending edge mapped to (v_1, v_4), due to the symmetric relation between (v_1, v_3) and (v_0, v_3), the algorithm searched only within the confirmed edges (v_3, v_5) and (v_3, v_1) to map (s_1, s_0). However, (s_1, s_0) could also be mapped to (v_3, v_0) to constitute an instance with $s_3 \mapsto v_4$, $s_2 \mapsto v_1$, $s_1 \mapsto v_3$ and $s_0 \mapsto v_0$, which CSM-A failed to enumerate. To generalise, CSM-A may miss a mapping if some node in the data graph can be mapped to multiple nodes of the subgraph pattern. In Figure 6, 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 $\mathcal{R}$eplica corresponds to at least one valid instance from the subgraph. Consequently, we can state the following theorem.

Theorem 1. *For any pair of subgraphs $Q, R \in T$, consider $\sigma(Q)$ to be the* MNI *support of Q and $\kappa(Q, R, h)$ the correlation between Q and R at separation h 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)** $\kappa(Q, R, h) \leq \kappa^*(Q, R, h)$ hold true.*

Table 1: **Datasets and characteristics**

Dataset	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
Memetracker	$96.6M$	$418.2M$	$24.75M$	1	Web

5. EXPERIMENTAL EVALUATION

In this section, we benchmark the proposed exact (CSM-E) and approximation (CSM-A) algorithms. The implementation is available at https://github.com/idea-iitd/correlated-subgraphs-mining.

5.1 Experimental Setup

All algorithms have been implemented in C++11 using GCC 7.4.0 compiler on a system running Ubuntu 18.04 with a 2.1 GHz Intel® Xeon® Platinum 8160 processor and 256 GB RAM.

5.1.1 Datasets

We use eight real networks (Table 1).

***Chemical* [3]**. This graph represents the structure of an anti-breast tumor compound in the *MCF7 Dataset* [65]. Each node represents a chemical atom, and two nodes are connected by an edge if they share a chemical bond.

***Citeseer* [4]**. Each node is a publication with its label categorising the area of research. Two nodes are connected if one of the two papers is cited by the other. The edge label is an integer in the range $[0, 5]$, representative of the similarity between the two papers such that a smaller label denotes stronger similarity (label 0 indicates similarity among the top 20 percentile, and so on).

***Coauthor (DBLP)* [5]**. A coauthorship network in which two authors (represented by 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.

***Citation (DBLP)* [5]**. Each node is a publication and the label is the publication venue. Two nodes share an edge if one of the two papers is cited by the other.

***LastFM* [6]**. *LastFM* is a social network of users where the node label represents the most frequent music band that the corresponding user listens to.

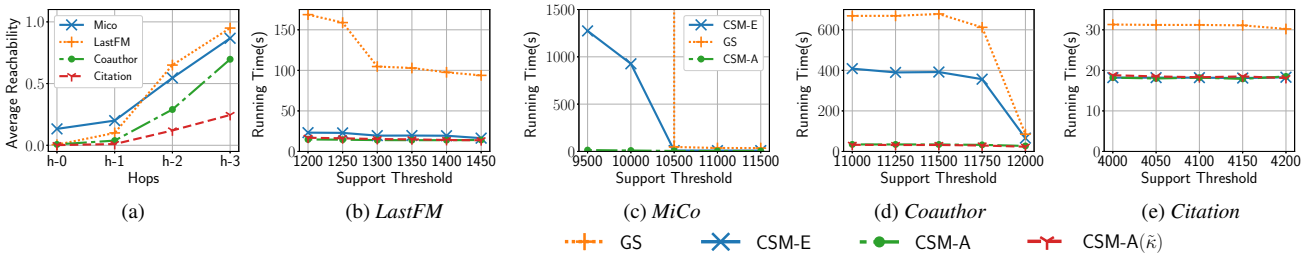
***Memetracker* [7]**. In this dataset, each node corresponds to a web document and two documents are connected if one document contains a hyperlink to the other. The node labels correspond to the *memes* contained in the document. A meme corresponds to quotes or phrases that appear frequently over a time period.

***MiCo* [8]**. *MiCo* models *Microsoft* coauthorship information. Each node represents an author and the label is the author's field of interest. Edges represent collaboration between authors and the edge label is the number of coauthored papers.

***Yeast* [9]**. This dataset represents the Protein-Protein Interaction (PPI) network in Yeast. The node labels denote their gene ontology tags [11], which capture their biological functions and properties.

5.1.2 Parameters

Unless otherwise stated, the default value of k is 20. The default distance threshold is set to $h = 1$. Here, we point out that $h \geq 3$ is not semantically meaningful since at large h, most nodes are reachable from each other, and hence, almost all pairs of frequent patterns report high correlation values. We provide empirical evidence in Figure 7a: while varying h, the average proportion of frequent pattern instances that are reachable from a randomly chosen frequent pattern instance is measured. Notice that at $h = 3$,



(a) (b) *LastFM* (c) *MiCo* (d) *Coauthor* (e) *Citation*

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

barring *Citation*, datasets have a *reachability* of at least 70%. Even in *Citation* (a sparse dataset), the reachability is almost 25% and increases exponentially with h.

5.1.3 Baselines

(1) GRAMI+VF3 (GVF3): This baseline approach has been discussed in detail in § 1.1. As reported, CSM-A is up to 5 orders of magnitude faster than GVF3 on *Citation*.

(2) GROWSTORE (GS): GS [40] employs the same algorithm as CSM-E but for one key difference: instead of employing $\mathcal{R}$eplica to summarise all instances, GS explicitly stores *every* instance in memory. Comparison against GS allows us to quantify the benefits of using the $\mathcal{R}$eplica data structure.

5.2 Impact of Optimisation Techniques

Our approach leverages three different optimisation techniques: $\mathcal{R}$eplica for data compression, CSM-A, which is a near-optimal approximation strategy to remove redundant enumerations and best-first search for efficient searching. In this section, we systematically study the impact of each of these optimisation strategies.

Impact of $\mathcal{R}$eplica and CSM-A on Running Time. First, we measure the impact of $\mathcal{R}$eplica and CSM-A. To that end, we compare the performance of CSM-E with CSM-A and GS. Due to large memory consumption in GS, in this experiment, we restrict to mining correlations only among subgraphs of size up to 5 nodes.

Figures 7b-7e present the running times across various datasets. Several insights can be derived from these results. First, $\mathcal{R}$eplica imparts a computational speed-up of $1.5X$-$6X$ as is evident from the differences in running times between CSM-E and GS. Second, the impact of removing redundant enumerations, i.e., CSM-A, varies across datasets with the difference being more prominent at lower support threshold Σ. This is expected since as Σ is lowered, the search space increases exponentially and the impact of avoiding repetitive computation (§ 4.2) magnifies. While in *MiCo* and *Coauthor*, the time gap between CSM-A and CSM-E is significant, in *LastFM* and *Citation*, we do not observe a significant speed-up. CSM-A removes enumeration of symmetric extensions. Thus, the speed-up achieved due to CSM-A is directly correlated to the prevalence of symmetric extensions in the dataset. Third, $\mathcal{R}$eplica has a higher impact on efficiency than removing redundant enumerations in CSM-A. Fourth, CSM-A is the fastest across all datasets due to utilising $\mathcal{R}$eplica and avoiding redundant enumerations. In Figures 7b-7e, for CSM-A, we include both (absolute) correlation κ and normalised correlation $\tilde{\kappa}$ as ranking metrics. The execution times remain almost identical for both metrics.

Impact of $\mathcal{R}$eplica on Memory Footprint. In Figure 8a, we compare the memory footprint of CSM-A with GS in *Citeseer*. Since CSM-E has similar memory requirements as CSM-A, we do not present CSM-E results in Figure 8a. So far, we have used the restriction of mining correlations only among subgraphs of size up to 5. Without this constraint, even in *Citeseer* which is a small dataset of 4500 edges, GS consumes more than 10GB at $\Sigma = 175$, which increases to exceed 256GB at $\Sigma = 150$. In contrast, the memory footprint of CSM-A is 100 times lower on average. Overall, the stark difference in memory consumption between CSM-A and GS highlights the benefits of using $\mathcal{R}$eplica as a compact data structure to store isomorphisms.

Search Strategy. We adopt best-first search (BEST) strategy to explore the search space. How is the performance impacted if we adopt BFS or DFS instead? Our next experiment answers this question. We explore the search space using each of these strategies and empirically track their pruning capacity by counting the number of patterns popped from the *Search Queue*.

Figures 8b-8c present 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 prioritisation scheme yields better results.

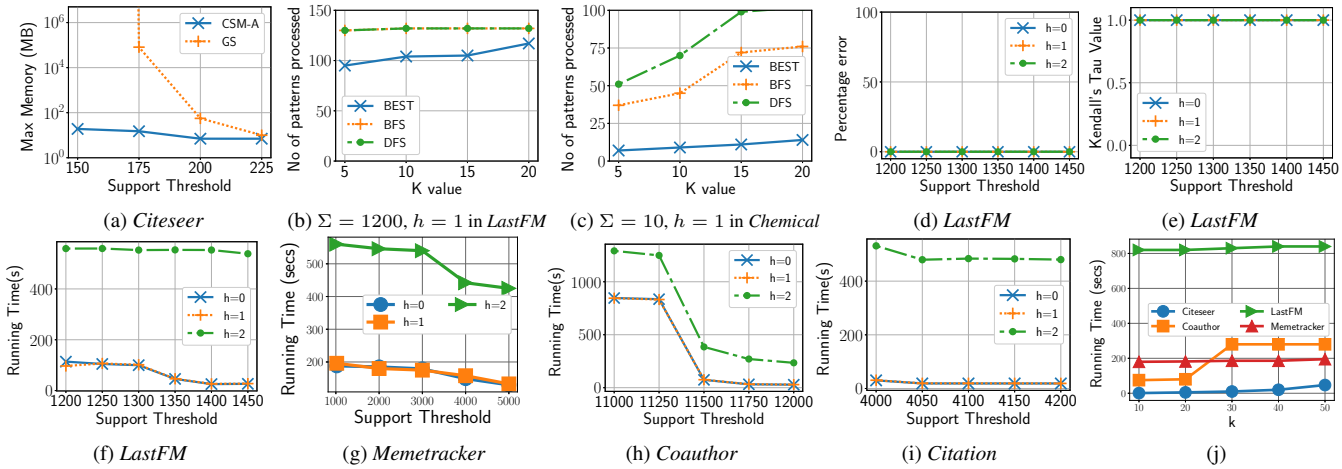
We observe that the impact of BEST is higher in *Chemical* than in *LastFM*. To better understand this result, we extract the correlation and support values of all patterns explored by BEST. Recall that BEST terminates when the maximum support of any unexplored subgraph is smaller than the k-th correlation count. Since BEST explores subgraphs in descending order of support, the maximum possible support of any unexplored subgraph is at most the minimum support of all explored subgraphs. In *Chemical*, if we sort the explored subgraph patterns based on support in descending order, the support drops steeply as we go down the sorted order. In *LastFM*, this decrease in support in the sorted order is more gradual. Consequently, the termination condition in *Chemical* is satisfied much earlier than in *LastFM*.

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 *F-score*, *Kendall's Tau* [2], and *Percentage error in correlation count*. 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 CSM-E for this same pair. From Lemma 1, 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). Table 3 shows that CSM-A

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	14.0	16.3	346000
Coauthor (DBLP)	27.8	64.3	503015
Citation (DBLP)	18.2	18.2	1311474
Citeseer	0.1	0.1	10



(a) *Citeseer* (b) $\Sigma = 1200$, $h = 1$ in *LastFM* (c) $\Sigma = 10$, $h = 1$ in *Chemical* (d) *LastFM* (e) *LastFM*

(f) *LastFM* (g) *Memetracker* (h) *Coauthor* (i) *Citation* (j)

Figure 8: **(a) Growth of memory consumption against support threshold. (b-c) Impact of different search strategies. (d-e) Approximation quality of** CSM-A **against support threshold. (f-i) Growth rate of running time against support threshold for** CSM-A**. (j) Impact of k on running time at support thresholds: *LastFM*: 1200, *Coauthor*: 11000, *Citeseer*: 150, *Memetracker*: 2000.**

produces near-optimal results. To further analyse how the quality changes with support threshold, in Figures 8d and 8e, 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.*

5.4 Impact of Parameters

Running time. Figures 8f-8i present the results from the four largest datasets listed in Table 1. As expected, the running time decreases with increasing support. It is worth noting that even at $h = 2$, CSM-A terminates within 20 minutes on million-scale datasets. As we increase h, time and memory required to compute proximate-nodes index ($CorV$) increases and as a result, the time taken for correlation computation also increases.

Figure 8j analyses 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.

It is interesting to observe that despite *Memetracker* network being the largest among the benchmarking datasets, its running times are lower than those for *Coauthor* and *LastFM*. Upon analysing the datasets, we observe three properties in *Memetracker* that contribute towards faster execution. First, the diversity of labels in *Memetracker* is higher, which leads to less instances of a frequent pattern. Second, homophily is not as abundant, which leads to smaller sizes of frequent patterns. Third, the graph density of *Memetracker* is less than that of *Coauthor* and *LastFM*, which leads to less chances of two patterns being present within h hops.

Incremental top-k. Figure 9a presents the results where we start at $k = 1$ and increase it *incrementally* by 5 units. In the y-axis, Figure 9a plots the time incurred to add each increment of 5 more patterns. As visible, the additional time incurred to increase k by 5 is always less than a minute and typically around 25 seconds even in *LastFM*, which has running times of ≈ 800 seconds (Figure 8j). This behaviour reduces reliability on setting the optimal value of k since it can be easily updated with low computational overhead. Note that we do not expect a monotonic increase in running times with incremental increase in k. Although every increment adds exactly 5 more patterns, the time to process this increment depends on the number of candidate pairs explored by best-first search. The number of explored candidates is not a monotonic function of k.

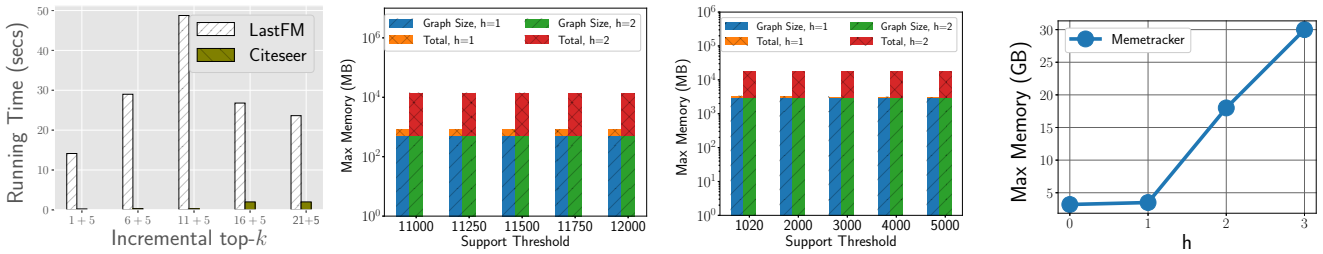
Memory footprint. Figures 9b-9c analyse the memory footprint against support threshold for various values of h. In Figure 9d, we plot the growth in memory against h.

• *Memory consumption of components.* We individually evaluate the memory consumption of the four primary components: the data graph, the proximate-nodes index $CorV$ storing which frequent nodes are reachable from each other within h hops, the proximate-patterns index $CorP$, and the $\mathcal{R}$eplica. We find that the combined memory footprint of the graph and $CorV$ is about 95% of the total consumption. This happens as we do not store the $\mathcal{R}$eplicas of all identified frequent patterns, i.e., we delete the $\mathcal{R}$eplicas of old (parent) patterns once all their frequent child patterns (generated via rightmost extension) have been included in the pattern search tree T. Thus, at any point in time, only the $\mathcal{R}$eplicas of the leaves of the best-first search tree are stored. $CorP$, on the other hand, stores only the patterns (but not their instances) and therefore has a minimal impact on the overall memory consumption.

• *Impact of support threshold* Σ. As support increases, the decrease in memory consumption of CSM-A is minimal. This observation can be explained from the behaviour of its four components. The size of the data graph, which is one of the major contributors to memory consumption, is independent of Σ. The size of $CorV$ does depend on Σ, however, the impact of Σ on its memory consumption is minimal. $CorV$ stores reachability information only among those nodes whose label frequency is higher than Σ. Since the distribution of node label frequencies follows power-law, therefore, even if Σ were high enough to prune out most of the node labels, the top-k most frequent labels span a large portion of the entire nodes set. Consequently, as Σ is lowered, despite some new

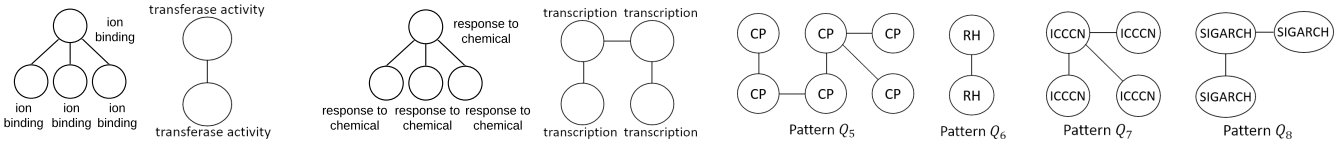
Table 3: **Quality evaluation of CSM-A**

Datasets	F-score	Kendall's Tau	% Error	Σ
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) (b) *Coauthor* (c) *Memetracker* (d) *Memetracker*

Figure 9: **(a) Running times in incremental increase of k. Memory usage of CSM-A against (b-c) support threshold and (d) h.**



(a) Pattern Q_1 (b) Pattern Q_2 (c) Pattern Q_3 (d) Pattern Q_4 (e) Pair in *LastFM* (f) Pair in *Coauthor*

Figure 10: **Case study. (a-d) Correlated subgraph pairs $\langle Q_1, Q_2 \rangle$ and $\langle Q_3, Q_4 \rangle$ in *Yeast*. (e) Correlated pair in *LastFM*.** CP **denotes *Coldplay* and** RH **denotes *Radiohead*. (f) Correlated pair in *Coauthor*.**

labels becoming frequent, the number of new nodes that they bring in is a small percentage of the entire nodes set. Hence, size of the $CorV$ index remains minimally impacted. Finally, although the size of the $\mathcal{R}$eplica and $CorP$ index increases with decrease in Σ, in the context of the total memory consumption, it does not have any significant impact.

• *Impact of h.* The primary contributor to memory consumption at $h > 1$ is the $CorV$ index. As h increases, more frequent nodes become reachable from each other and thus CSM-A requires more memory. Note that the memory consumption values at $h = 0$ and $h = 1$ are almost identical since in these cases the adjacency list of the data graph suffices to serve as the $CorV$ index.

5.5 Case Study: Qualitative Analysis

To demonstrate the utility of CSM, we present insights derived from the correlated pairs mined in the following datasets.

***Yeast* [9].** The node labels (Gene Ontology tags) in this Protein Interaction (PPI) network (§ 5.1.1) indicate their biological function. In Figure 10, we present two of the top-10 correlated pairs mined at $\Sigma = 300$. Figures 10a (Q_1) and 10b (Q_2) show the first pair. Q_1 is constituted entirely of units specialising in *ion binding*, and Q_2 is composed of those specialising in *transferase activity*. Keike et al. [37, 38] have shown that transferases can gain enzymatic activity following the binding of ionic ligands, undergoing conformational changes to insulate the ligands from surrounding water molecules.

We highlight another pair $\langle Q_3, Q_4 \rangle$ in Figures 10c and 10d. Q_3 is made up of genes that regulate responses to *chemicals* and Q_4 is associated with *transcription*. This co-occurrence is not a coincidence. Specifically, the Gene Ontology [1] database describes in detail the involvement of positive transcription regulation in cellular response to chemical stimulus. Finally, we verified each of the top-10 correlated pairs through domain experts to determine what percentage of pairs suggested biological correlation. The experts could not reject any pair as biologically non-linked. In their opinion, all 10 correlated pairs are either definitely linked, or suggest interesting correlations requiring further investigation.

***LastFM* [6].** Figure 10e depicts a correlated pair from the top-k set. This pair showcases that communities of users who like the music band *Coldplay*, are often in close proximity to communities of users who like *Radiohead*. These two bands are contemporaries, originate from UK and they are often compared on social media.

***Coauthor* [5].** Figure 10f presents a pair from the top-10 list in the *Coauthor (DBLP)* network. Nodes correspond to authors and node labels denote the most frequent publication venue of the authors. In this regard, the presented pair is interesting since it reveals proximity between the networking community (ICCCN) and the architecture community (SIGARCH). While most of the other pairs are between conferences centred on similar areas, this pair shows that there is high collaboration between networking and architecture communities. More importantly, this pair reveals that correlated subgraphs may unearth non-trivial connections between communities.

6. 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 from frequent subgraph patterns themselves?* In this paper, we explored this question by mining correlated pairs of frequent subgraphs. Unlike frequent subgraphs mining, we not only need to identify if a subgraph pattern is frequent, but also enumerate, maintain, and compute distances between all instances of all frequent subgraphs. Managing instances imposes a severe scalability challenge both computationally and memory-wise. We tackled this challenge by designing a data structure called $\mathcal{R}$eplica, which stores all instances in a compact manner. $\mathcal{R}$eplica also allowed us to design a near-optimal approximation scheme to identify and enumerate instances efficiently. Through extensive evaluation across a series of real datasets, we demonstrated that the proposed mining algorithms scale to million-sized networks, exhibit up to 5 orders of magnitude speed-up over baseline techniques and discover correlations 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, we propose to extend the analysis to include mining arbitrary-sized groups of correlated subgraphs.

Acknowledgement. The second author, Arijit Khan, is supported by MOE Tier1 and Tier2 grants RG117/19, MOE2019-T2-2-042.

7. REFERENCES

[1] GO for utility process. http://www.candidagenome.org/cgi-bin/GO/go.pl?goid=1901522.

[2] Kendall's Tau. https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient.

[3] Source for Chemical dataset. http://pubchem.ncbi.nlm.nih.gov.

[4] Source for Citeseer dataset. http://networkrepository.com/citeseer.php.

[5] Source for Coauthor and Citation (DBLP) datasets. https://www.aminer.org/citation.

[6] Source for LastFM dataset. https://www.last.fm/.

[7] Source for Memetracker dataset. https://snap.stanford.edu/data/memetracker9.html.

[8] Source for MiCo dataset. http://academic.research.microsoft.com.

[9] Source for Yeast dataset. http://string-db.org/cgi/download.pl.

[10] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *VLDB*, 1994.

[11] M. Ashburner. Gene Ontology: Tool for the Unification of Biology. *Nature Genetics*, 25:25–29, 2000.

[12] C. Borgelt and M. R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *ICDM*, 2002.

[13] B. Bringmann and S. Nijssen. What is Frequent in a Single Graph? In *PAKDD*, 2008.

[14] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(4):804–818, 2018.

[15] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining Graph Patterns Efficiently via Randomized Summaries. *PVLDB*, 2(1):742–753, 2009.

[16] D. J. Cook and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. *J. Artif. Intell. Res.*, 1:231–255, 1994.

[17] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *PVLDB*, 7(7):517–528, 2014.

[18] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph Pattern Matching: From Intractable to Polynomial Time. *PVLDB*, 3(1):264–275, 2010.

[19] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph Homomorphism Revisited for Graph Matching. *PVLDB*, 3(1):1161–1172, 2010.

[20] M. Fiedler and C. Borgelt. Support Computation for Mining Frequent Subgraphs in a Single Graph. In *MLG*, 2007.

[21] B. Gallagher. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. *AAAI FS.*, 2006.

[22] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan. Assessing and Ranking Structural Correlations in Graphs. In *SIGMOD*, 2011.

[23] S. Gurukar, S. Ranu, and B. Ravindran. Commit: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *SIGMOD*, 2015.

[24] M. A. Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: Mining Representative Orthogonal Graph Patterns. In *ICDM*, 2007.

[25] M. A. Hasan and M. J. Zaki. Output Space Sampling for Graph Patterns. *PVLDB*, 2(1):730–741, 2009.

[26] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *FOCS*, 1995.

[27] J. Huan, W. Wang, and J. F. Prins. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In *ICDM*, 2003.

[28] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. In *PKDD*, 2000.

[29] N. Jayaram, S. Goyal, and C. Li. VIIQ: Auto-Suggestion Enabled Visual Interface for Interactive Graph Query Formulation. *PVLDB*, 8(12):1940–1943, 2015.

[30] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. PRAGUE: Towards Blending Practical Visual Subgraph Query Formulation and Query Processing. In *ICDE*, 2012.

[31] N. Jin, C. Young, and W. Wang. GAIA: Graph Classification using Evolutionary Computation. In *SIGMOD*, 2010.

[32] Y. Ke, J. Cheng, and J. X. Yu. Efficient Discovery of Frequent Correlated Subgraph Pairs. In *ICDM*, 2009.

[33] B. P. Kelley, B. Yuan, F. Lewitter, R. Sharan, B. R. Stockwell, and T. Ideker. PathBLAST: A Tool for Alignment of Protein Interaction Networks. *Nucleic Acids Res*, 32(Web-Server-Issue):83–88, 2004.

[34] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood Based Fast Graph Search in Large Networks. *SIGMOD*, 2011.

[35] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. NeMa: Fast Graph Search with Label Similarity. *PVLDB*, 6(3):181–192, 2013.

[36] A. Khan, X. Yan, and K.-L. Wu. Towards Proximity Pattern Mining in Large Graphs. In *SIGMOD*, 2010.

[37] R. Koike, T. Amemiya, M. Ota, and A. Kidera. Protein Structural Change upon Ligand Binding Correlates with Enzymatic Reaction Mechanism. *Journal of molecular biology*, 379:397–401, 07 2008.

[38] R. Koike, A. Kidera, and M. Ota. Alteration of State and Domain Architecture is Essential for Functional Transformation between Transferase and Hydrolase with the Same Scaffold. *Protein Science : a Publication of the Protein Society*, 18:2060–6, 10 2009.

[39] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. In *ICDM*, 2001.

[40] M. Kuramochi and G. Karypis. Finding Frequent Patterns in a Large Sparse Graph. In *SDM*, 2004.

[41] E. A. Lee, S. Fung, H. Sze-To, and A. K. C. Wong. Discovering Co-occurring Patterns and their Biological Significance in Protein Families. *BMC Bioinformatics*, 15(S-12):S2, 2014.

[42] Z. Liang, M. Xu, M. Teng, and L. Niu. NetAlign: A Web-based Tool for Comparison of Protein Interaction Networks. *Bioinfo.*, 22(17):2175–2177, 2006.

[43] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing Topology in Graph Pattern Matching. *PVLDB*, 5(4):310–321, 2012.

[44] S. Maneth and F. Peternek. Compressing Graphs by Grammars. In *ICDE*, 2016.

[45] M. Mongiovì, R. D. Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. SIGMA: A Set-Cover-Based Inexact Graph Matching Algorithm. *J. Bioinfo. and Comp. Bio.*, 8(2):199–218, 2010.

[46] D. Natarajan and S. Ranu. A Scalable and Generic

Framework to Mine Top-k Representative Subgraph Patterns. In *ICDM*, 2016.

[47] D. Natarajan and S. Ranu. Resling: A Scalable and Generic Framework to Mine Top-k Representative Subgraph Patterns. *Knowledge and Information Systems*, 54(1):123–149, 2018.

[48] S. Nijssen and J. N. Kok. A Quickstart in Frequent Structure Mining Can Make a Difference. In *KDD*, 2004.

[49] S. Pande, S. Ranu, and A. Bhattacharya. SkyGraph: Retrieving Regions of Interest using Skyline Subgraph Queries. *PVLDB*, 10(11):1382–1393, 2017.

[50] S. Ranu, B. T. Calhoun, A. K. Singh, and S. J. Swamidass. Probabilistic Substructure Mining From Small-Molecule Screens. *Molecular Informatics*, 30(9):809–815, 2011.

[51] S. Ranu, M. Hoang, and A. Singh. Mining Discriminative Subgraphs from Global-state Networks. In *KDD*, 2013.

[52] S. Ranu, M. Hoang, and A. Singh. Answering Top-k Representative Queries on Graph Databases. In *SIGMOD*, 2014.

[53] S. Ranu and A. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*, 2009.

[54] S. Ranu and A. K. Singh. Mining Statistically Significant Molecular Substructures for Efficient Molecular Classification. *Journal of Chemical Information and Modeling*, 49:2537–2550, 2009.

[55] S. Ranu and A. K. Singh. Indexing and Mining Topological Patterns for Drug Discovery. In *EDBT*, 2012.

[56] A. Silva, J. W. Meira, and M. J. Zaki. Mining Attribute-structure Correlated Patterns in Large Attributed Graphs. *PVLDB*, 5(5):466–477, 2012.

[57] R. Singh, J. Xu, and B. Berger. Global Alignment of Multiple Protein Interaction Networks with Application to Functional Orthology Detection. *PNAS*, 105(35):12763–12768, 2008.

[58] Y. Tian, R. McEachin, C. Santos, D. States, and J. Patel. SAGA: A Subgraph Matching Tool for Biological Graphs. *Bioinfo.*, 23(2):232–239, 2006.

[59] Y. Tian and J. M. Patel. TALE: A Tool for Approximate Large Graph Matching. *ICDE*, 2008.

[60] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, 1976.

[61] J. Vachery, A. Arora, S. Ranu, and A. Bhattacharya. RAQ: Relationship-Aware Graph Querying in Large Networks. In *The WebConference*, 2019.

[62] N. Vanetik, E. Gudes, and S. E. Shimony. Computing Frequent Graph Patterns from Semistructured Data. In *ICDM*, 2002.

[63] M. Worlein, T. Meinl, I. Fischer, and M. Philippsen. A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. In *PKDD*, 2005.

[64] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *SIGMOD*, 2008.

[65] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*, 2002.

[66] X. Yan and J. Han. CloseGraph: Mining Closed Frequent Graph Patterns. In *KDD*, 2003.

[67] S. Zhang, J. Yang, and S. Li. RING: An Integrated Method for Frequent Representative Subgraph Mining. In *ICDM*, 2009.