GraMi: Generalized Frequent Pattern Mining in a Single Large Graph

Mohammed El Saeedy and Panos Kalnis

Mathematical & Computer Sciences and Engineering
King Abdullah Univertsity of Science and Technology (KAUST)
{mohammed.elsaeedy, panos.kalnis}@kaust.edu.sa

ABSTRACT

Mining frequent subgraphs is an important operation on graphs. Most existing work assumes a database of many small graphs, but modern applications, such as social networks, citation graphs or protein-protein interaction in bioinformatics, are modeled as a single large graph. Interesting interactions in such applications may be transitive (e.g., friend of a friend). Existing methods, however, search for frequent isomorphic (i.e., exact match) subgraphs and cannot discover many useful patterns.

In this paper we propose GRAMI, a framework that generalizes frequent subgraph mining in a large single graph. GRAMI discovers frequent patterns. A pattern is a graph where edges are generalized to distance-constrained paths. Depending on the definition of the distance function, many instantiations of the framework are possible. Both directed and undirected graphs, as well as multiple labels per vertex, are supported. We developed an efficient implementation of the framework that models the frequency resolution phase as a constraint satisfaction problem, in order to avoid the costly enumeration of all instances of each pattern in the graph. We also implemented CGRAMI, a version that supports structural and semantic constraints; and AGRAMI, an approximate version that supports very large graphs. Our experiments on real data demonstrate that our framework is up to 3 orders of magnitude faster and discovers more interesting patterns than existing approaches.

1. INTRODUCTION

Graphs model complex relationships among objects in a variety of applications such as chemi- and bio-informatics, computer vision, social networks, video indexing, text retrieval and web analysis. Mining frequent subgraphs is a central and well studied problem in graphs. Frequent subgraphs play a critical role in many data mining tasks that include: classification algorithms [5], modeling of user profiles [7], graph clustering [10], database design [6] and index selection [28].

The majority of the existing literature [11, 13, 15, 20, 26] focuses on the transactional setting. There is a database of many, typically small graphs, where each graph represents a transaction.

Technical Report, November 2011
Division of Mathematical and Computer Sciences and Engineering
King Abdullah University of Science and Technology (KAUST).

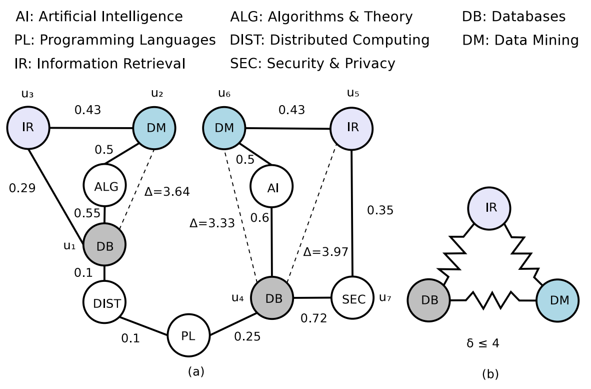


Figure 1: (a) Co-authorship graph; nodes are authors; labels represent the author's field of work; solid edges represent co-authorship; *weights* show the collaboration strength. Dotted lines are calculated transitive relationships; their weights are *distances* (e.g., $\Delta(u_5, u_4) = 0.35^{-1} \cdot 0.72^{-1} = 3.97$). For $\tau = 2$ and $\delta = 4$, GRAMI finds the frequent pattern in (b).

A subgraph is frequent if it exists in at least τ transactions, where τ is a user-defined threshold. In contrast, we focus on a *single large graph*. Such a setting is required in many modern applications, including social networks and protein-protein interaction (*PPI*) networks. The single graph setting is a generalization of the transactional case, since a set of small graphs can be considered as connected components within a single large graph. Detecting frequent subgraphs inside a single graph is more complicated because multiple instances of identical subgraphs may overlap. Moreover, computationally the single graph case is more demanding, because the complexity is exponential to the size of the graph.

Figure 1.a shows an example of a collaboration network, where nodes correspond to authors and solid edges represent co-authorship. Each edge is assigned a weight $0 < w \leq 1$ that represents the strength of collaboration (evaluated by the number of co-authored papers). Existing frequent subgraph mining techniques such as SIGRAM [17], use the apriori principle to search for subgraph instances that appear at least τ times and are *isomorphic* (i.e., match exactly). Assume $\tau = 2$. Nodes DB, IR and DM are frequent subgraphs (consisting of a single node), because their frequency $\sigma_{\mathtt{DB}} = \sigma_{\mathtt{IR}} = \sigma_{\mathtt{DM}} \geq \tau$. Similarly, subgraph IR $\leftrightarrow$ DM is frequent because two instances appear in the graph (assuming that the graph is unweighted because SIGRAM works on labeled graphs only). On the other hand, IR $\leftrightarrow$ DB is *not* frequent according to SIGRAM, because its frequency $\sigma_{\mathtt{IR}\leftrightarrow\mathtt{DB}} = 1 < \tau$.

Mining instances of subgraphs that match exactly is useful when the topological and label information is accurate and complete (e.g., chemical compounds). However this is not the case for other kinds of graphs, such as social or other networks, that may contain incomplete information and transitive relationships. In such cases *indirect* relationships may reveal useful information. Continuing the example of Figure 1.a, there exists only one direct connection between `IR` and `DB` (i.e., $u_3 \leftrightarrow u_1$) but there is also an indirect connection via `SEC` (i.e., $u_5 \leftrightarrow u_7 \leftrightarrow u_4$). This transitive connection increases the importance of the relationship between `IR` and `DB`. Let us define a monotonically increasing distance function Δ as the product of the inverse of the weights on that path: $\Delta(u_5, u_4) = \Delta(u_5, u_7) \cdot \Delta(u_7, u_4) = 0.35^{-1} \cdot 0.72^{-1} = 3.97$. Moreover, assume a user-defined distance threshold $\delta = 4$. Since $\Delta(u_5, u_4) \leq \delta$, path $u_5 \leftrightsquigarrow u_4$ is considered isomorphic to $u_3 \leftrightarrow u_1$; therefore, pattern `IR` $\leftrightsquigarrow$ `DB` is frequent. Observe that this definition of frequent patterns is a generalization of the existing approaches. We can obtain the same results as SIGRAM by redefining Δ to be the number of hops in the path and $\delta = 1$.

In this paper we propose GRAMI, a general framework for frequent subgraph mining in large graphs. GRAMI works with directed and undirected graphs. It supports both single and multiple labels per node, and edges that may or may not have weights. The input for GRAMI is a graph G, a frequency threshold τ and a path distance threshold δ. GRAMI also requires a monotonically increasing distance function Δ. Different instantiations of the framework are possible based on Δ. Intuitively, the framework replaces the notion of an edge between two nodes in the subgraph, with a path, allowing the discovery of interesting frequent interactions among nodes, even if those interactions are not direct. Continuing our example, GRAMI calculates the transitive interactions that appear as dotted lines in Figure 1.a; therefore, it can discover the frequent pattern shown in Figure 1.b.

Note that there exist approximate methods, such as gApprox [2], that decide on subgraph isomorphism based on a similarity metric. In some cases, gApprox may discover a subset of the solutions of GRAMI. Nevertheless, GRAMI is *not* an approximate approach. Instead, GRAMI returns the complete set of frequent patterns based on the generalized definition of subgraph isomorphism.

A naïve implementation of the framework is the following: (*i*) Find all node labels with frequency $\sigma \geq \tau$ and store all instances (also called *embeddings*) of the corresponding nodes. (*ii*) Extend stored embeddings to construct larger frequent patterns that satisfy the distance threshold δ; store the new embeddings. (*iii*) Repeat *ii* until no more frequent patterns can be found. Variations of this algorithm are used by existing approaches such as SIGRAM. However, the number of embeddings generated in step *ii* increases exponentially with the size of the pattern, rendering the problem intractable in practice. GRAMI follows a different approach: We model the count operation as a constraint satisfaction problem (*CSP*). At each iteration, we solve the CSP until we find enough embeddings to prove the pattern frequent; additional embeddings are ignored. We store the frequent patterns only (i.e., we do not store their embeddings), and repeat the process by extending the patterns, until no more frequent patterns can be found.

By avoiding the enumeration of the embeddings, GRAMI scales much better than the existing approaches. Nevertheless, solving the CSP can still take exponential time in the worst case. In order to support large graphs in real-life applications, we developed two variations: (*i*) CGRAMI allows the user to define a set of *constraints*, both structural (e.g., the pattern is allowed to have up to α edges) and semantic (e.g., a particular label cannot occur more than α times in the pattern). The constraints are used to prune the search space. (*ii*) AGRAMI is an *approximate* version, which approximates the frequency of patterns. It may miss some patterns (i.e., false negatives), but the returned patterns are *not* approximate (i.e., no false positives).

In summary, our contributions are:

- We propose GRAMI, a framework to mine frequent patterns in a large single graph, by generalizing the notion of subgraph isomorphism to arbitrary distance-constrained patterns.
- We present an efficient implementation of GRAMI that avoids the costly enumeration of pattern embeddings.
- We develop CGRAMI, a version that supports structural and semantic constraints; and AGRAMI, an approximate version with no false positives.
- We demonstrate experimentally that GRAMI is up to 3 orders of magnitude faster and can discover more interesting patterns than existing methods in large real-life graphs.

The rest of the paper is organized as follows. Section 2 formalizes the problem, Section 3 surveys the related work and Section 4 discusses the naïve approach. GRAMI's smart enumeration technique is presented in Section 5, followed by a set of optimizations in Section 6. CGRAMI and AGRAMI are discussed in Sections 7 and 8, respectively. Section 9 presents the experimental evaluation, whereas Section 10 concludes the paper.

2. PRELIMINARIES

This section introduces the graph model, the frequency metric, and formalizes the problem definition.

Definition 2.1 (Graph *G*) *$G = (V_G, E_G, \lambda_G, \omega_G)$ is a graph composed of a set of vertices V_G, a set of edges $E_G \subseteq V_G \times V_G$, a labeling function $\lambda_G : V_G \rightarrow L$ that assigns labels to the vertices, and a weight function $\omega_G : E_G \rightarrow \mathbb{R}^+$ that maps each edge $(u, v) \in E_G$ to a real number $w > 0$.*

To simplify the presentation, all examples assume undirected graphs and a single label for each vertex. However, our implementation also supports directed graphs and multiple labels per vertex; the extension is straightforward.

Definition 2.2 (Distance Δ) *Given a graph G, $\Delta_G : V_G \times V_G \rightarrow \mathbb{R}^+$ is a monotonically increasing function that assigns a distance between a pair of nodes $u, v \in V_G$. If v is reachable from u through multiple paths, $\Delta_G(u, v)$ is the minimum of the distances that correspond to these paths.*

Typically, Δ is a function of the edge weights, but this is not a requirement of GRAMI. For example, $\Delta(u, v)$ can be defined as the shortest path (in terms of hops) between u and v.

Definition 2.3 (Pattern *P*) *A pattern $P = (V_P, E_P, \lambda_P)$ is a graph without the weight function.*

Definition 2.4 (Embedding ψ) *Given graph G, pattern P, distance function Δ_G and a user-defined distance threshold δ, an embedding is a function $\psi : V_P \rightarrow V_G$ that maps the vertices in P to those in G, such that:* (i) *$\forall v \in V_P \Rightarrow \lambda_P(v) = \lambda_G(\psi(v))$, and* (ii) *$\forall (u, v) \in E_P \Rightarrow \Delta_G(\psi(u), \psi(v)) \leq \delta$.*

Condition *i* states that the labels of the corresponding vertices in P and G must match, whereas condition *ii* ensures there exists a path from $\psi(u)$ to $\psi(v)$ in G having distance at most δ. If condition *ii* is replaced with $\forall (u, v) \in E_P \Rightarrow (\psi(u), \psi(v)) \in E_G$, the

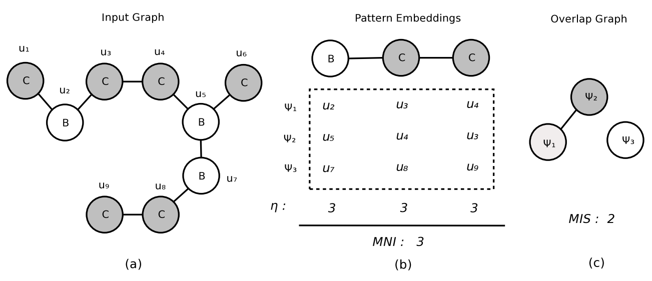


Figure 2: Frequency metrics. (a) Input graph G. (b) Embeddings for MNI metric. Last row shows COUNT(DISTINCT nodeID) of each column; support is the minimum of the values in the last row. (c) Overlap graph of MIS; support is the size of the maximal set of non-connected nodes (i.e., $\{\psi_1, \psi_3\}$ or $\{\psi_2, \psi_3\}$) in the overlap graph.

above definition is equivalent to the *subgraph isomorphism* problem, which is known to be NP-complete [8]. GRAMI generalizes subgraph isomorphism by substituting the edge constraint with a distance-constrained path.

Note that GRAMI also supports unbounded distance, by setting $\delta = \infty$. This generalizes $\Delta(u, v)$ to a reachability test between u and v; it is useful for directed graphs that are already constrained in terms of reachability.

Definition 2.5 (Frequent pattern) *Let σ_P be a value that represents the frequency of pattern P in graph G; σ_P is called the* support *of P. P is said to be frequent if its support $\sigma_P \geq \tau$, where τ is a user-defined threshold.*

The above definition does not explain how to calculate the support σ_P. In the transactional case (i.e., a set of graphs) the calculation is simple: the support of a pattern P is the number of distinct graphs that contain embeddings of P. In contrast, in a single graph the definition of an appropriate frequency metric is more difficult because of overlaps. Figure 2.a shows an example, where node labels B and C appear 3 and 6 times, respectively. However the combination B $\leftrightarrow$ C appears 5 times, violating the antimonotonic property (i.e., the apriori principle does not hold).

Assume Δ is the number of hops and $\delta = 1$. Figure 2.b shows the three embeddings $\psi_{1\ldots3}$ of pattern $P \equiv$ B $\leftrightarrow$ C $\leftrightarrow$ C. Columns $i = 1 \ldots 3$ correspond to vertices $v_i \in P$, whereas cell values correspond to node IDs in G. For instance, ψ_2 maps P to nodes u_5, u_4, u_3. Let $|\eta_i|$ be the number of distinct nodes (called *node images*) in column i. For example, the number of node images for the rightmost C is $|\eta_3| = 3$. Bringmann and Nijssen [1] define support as the minimum of all $|\eta_i|$; in our example, the support of P is 3. Formally:

Definition 2.6 (Minimum imaged-based support σ) *Let G be a graph, P a pattern, and $\psi_{1..m}$ all possible embeddings of P in G. Let $\eta_i = \{\forall \psi_j : \psi_j(v_i)\}$ be the set of (unique) nodes in G where $v_i \in P$ is mapped to. The minimum imaged-based support (*MNI*) is: $\sigma = min_{v_i \in V_P} |\eta_i|$.*

MNI is antimonotonic [1], therefore the apriori principle holds. Observe that MNI allows some form of overlapping among embeddings. In our example u_3 and u_4 appear both in ψ_1 and ψ_2. Kuramochi and Karypis [17] used the maximum independent set (*MIS*) metric that does not allow overlappings. Figure 2.c shows the *overlap graph* for our example, where a node corresponds to an embedding ψ_j and there is an edge if two embeddings share an edge in G. MIS defines support as the size of the maximal set of nodes in the overlap graph that are not connected (i.e., 2 in the example). Unfortunately, calculating MIS is NP-hard [8]. Moreover, MNI is a tight upper bound for MIS [1], it is much less expensive to compute and is often more intuitive than MIS. Therefore, we use MNI in the rest of this paper.

Problem formulation *Given graph G, distance function Δ_G, distance threshold δ, minimum support threshold τ and assuming the MNI frequency metric, find all frequent patterns P_i.*

3. RELATED WORK

Most of the work on frequent subgraph mining has focused on the transactional case that assumes a dataset of many, usually small, graphs. AGM [13] and FSG [15] are early works that use an apriori level-wise approach, meaning that they construct new candidate patterns by joining smaller frequent ones. The drawback for this approach is that they have to face two major challenges namely, the complicated and costly join operation and pruning false positives (false candidates). Later work, such as gSpan [26], FFSM [11] and Gaston [20], propose variations of pattern growth approaches. They avoid the previous drawbacks by proposing a candidate generation theme that relies on an extension mechanism, where patterns are grown directly from a single graph instead of joining two previous subgraphs. For large scale datasets, Wang et al. [24] propose a disk-based approach. Other methods focus on a subset of all frequent patterns. For example, SPIN [12] and MARGIN [22] return maximal patterns only, whereas CloseGraph [27] generates closed frequent patterns (i.e., patterns that have strictly smaller support than any of their sub-patterns). Yan et al. [25], on the other hand, discover significant patterns that are not necessarily frequent. A similar approach is also taken by Graphsig [21], which introduces a discriminative subgraph mining technique.

Closer to our work is TSMiner [14], which discovers frequent patterns in a database of graphs by employing the concept of topological minor to define subgraph equivalence. A topological minor is an abstraction achieved by replacing independent paths with single edges. This approach is not expected to scale in the single graph setting, because it relies on storing all intermediate embeddings that increase exponentially. Compared to TSMiner, GRAMI is more general because (*i*) it is not confined to independent paths only, and (*ii*) it supports a general distance function Δ, instead of only the number of hops.

In the single graph setting there exists less work. This setting generalizes the transactional case, if each small graph in the database is considered as a connected component of a single large graph [17]. Recall from Section 2 that the major difference in the single graph case is the definition of an appropriate antimonotonic support metric, which complicates the adoption of methods from the transactional setting. Kuramochi and Karypis [17] used the MIS metric and proposed SIGRAM, an algorithm that finds frequent connected subgraphs in a single, labeled, sparse and undirected graph. SIGRAM needs to enumerate all embeddings and the computation of MIS is NP-hard, therefore the method is prohibitively expensive in practice. The same authors developed GREW [16], a heuristic approach that prunes large parts of the search space, but discovers only a subset of the answers. In contrast, GRAMI employs the MNI support metric [1] (Section 2), which can be computed much faster and is more intuitive than MIS. GRAMI does not need to enumerate all the embeddings, so it scales to much larger problems. Also, GRAMI is not restricted to sparse graphs, and it supports directed and multi-labeled graphs. More importantly, SIGRAM and GREW are restricted to exact subgraph match (i.e., isomorphism), whereas GRAMI generalizes the concept of frequent pattern mining by al-

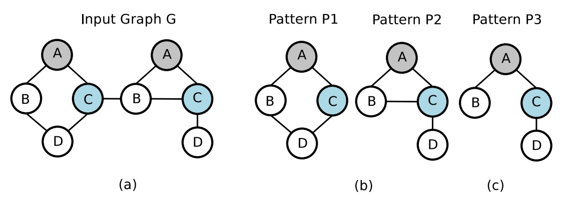


Figure 3: gApprox versus GRAMI; $\delta = 1, \tau = 2$ (a) Input graph G. (b) Induced subgraphs P_1 and P_2; none is frequent according to gApprox. (c) Non-induced subgraph P_3; it is discovered only by GRAMI.

lowing distance-constrained paths in the patterns.

Although GRAMI is an exact method, there is some related work in approximate graph mining. gApprox [2] employs an approximate version of the MIS metric. It mainly relies on enumerating all the embeddings, but allows approximate match both for the node labels and the structure of the pattern. In some cases, gApprox can produce a subset of GRAMI's answers. However, gApprox discovers only induced patterns. Figure 3 shows an example, assuming $\delta = 1$, $\tau = 2$ and Δ defined as number of hops. P_1 and P_2 are induced patterns, but both of them appear only once in G, so they are not frequent. On the other hand, pattern P_3 appears twice. GRAMI marks P_3 frequent, but gApprox cannot discover P_3 because it is not induced in G. SEuS [9] is another approximate method that constructs a compact summary of G, which facilitates the pruning process of infrequent candidates. Its authors explicitly note that SEuS is only useful when G contains few and very frequent subgraphs, but it is incapable of extracting subgraphs with low frequency. For the case of very large graphs that do not fit in memory, Zhou and Holder [31] propose a random sampling approach. Finally, SUBDUE [4] is a technique that aims to mine patterns with *minimum description length* such that they can be used to compress the original graph. It also proposes an approximate branch-and-bound approach to identify these structures. Two subgraphs are considered equivalent if their graph edit distance is within a threshold.

There is also work on pattern match queries over graphs. [3] searches for a query pattern Q in a directed graph G. If two vertices u, v are reachable in Q then their corresponding mappings $\psi(u), \psi(v)$ in G must also be reachable. [30] extends the idea to undirected graphs. In this case, there must exist a distance-constrained path between $\psi(u)$ and $\psi(v)$. In both approaches the query pattern Q is given, whereas in our problem GRAMI needs to discover the frequent patterns.

4. NAÏVE APPROACH

Frequent pattern mining in graphs has two phases: (*i*) Candidate generation search tree. The majority of existing work focuses on this phase. Section 4.1 explains how we employ these methods in GRAMI. (*ii*) Computation of support σ. In Section 4.2 we discuss a naïve algorithm similar to the one used by existing work. We will see later how our smart enumeration approach improves this phase.

4.1 Candidate Generation

Most state-of-the-art frequent subgraph miners follow the *growth* approach: First they identify all frequent nodes (i.e., support $\sigma \geq \tau$) in the graph. In each subsequent step they extend the current subgraphs by adding an edge or a node. The support of the new subgraphs is evaluated and those that do not satisfy the support threshold τ are eliminated because, according to the antimonotone property, their supergraphs are infrequent, as well.

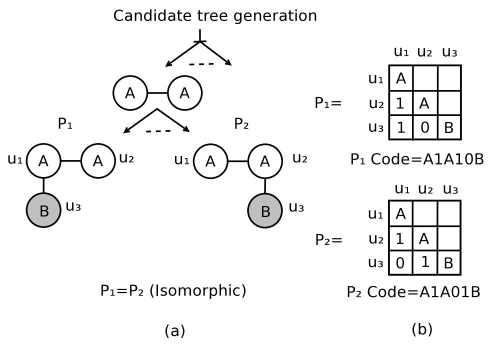


Figure 4: Redundant candidates and canonical codes. (a) P_1 and P_2 are isomorphic patterns, although they are generated in different ways: P_1 by connecting u_3 to u_1, and P_2 by connecting u_3 to u_2. (b) Matrix representation of P_1 and P_2 ('1' denotes edge existence). The matrix canonical codes for P_1 and P_2 are $\texttt{A1A\underline{1}0B}$ and $\texttt{A1A\underline{0}1B}$. The code of P_2 is minimal in the lexicographic order, so P_1 is redundant and is pruned.

Intuitively, the above process corresponds to a candidate generation search tree. Observe that an isomorphic subgraph can be redundantly generated from different parents in the tree, by adding nodes and edges in different orders. This results in a redundant subtree that is already present somewhere else in the search space. Figure 4(a) gives an example of this problem. P_1 and P_2 are isomorphic, but P_1 is generated by extending node u_1, whereas P_2 is generated by extending u_2. Ideally, P_1 should be pruned without needing to refer to P_2. To achieve this, various forms of canonical representations that identify uniquely a graph, have been proposed. Figure 4(b) shows an example, where P_1 and P_2 are depicted as adjacency matrices. The code of each subgraph is a string constructed by concatenating the rows of its matrix. The canonical code is defined as follows [13]: For a subgraph P_i, construct all isomorphic subgraphs and generate their codes. Order the codes lexicographically; the minimum one is the canonical code. Following this definition, the canonical code for graph $\texttt{A} - \texttt{A} - \texttt{B}$ is $\texttt{A1A01B}$. Therefore, P_2 is retained, whereas P_1 is pruned.

Although the canonical representation prunes the redundant subtrees, it is better to avoid constructing redundant patterns in the first place. Our implementation adopts the approach of gSpan [26], which recursively generates the complete set of candidate subgraphs by restricting the growth of a subgraph to the nodes in the right-most path of its depth-first search tree. Furthermore a node can not be extended with a backward edge (to a previously existing node) except for the right-most node. This approach reduces greatly the possibility of generating redundant candidates. However, some redundant candidates may be generated, therefore gSpan employs a form of canonical representation called DFScode. Each vertex is given a unique identifier based on the depth-first traversal of the subgraph. Let u and v be vertex identifiers, $\lambda(u), \lambda(v)$ be their labels and $\lambda_{u,v}$ be the edge label connecting them. Edge (u, v) is represented by a 5-tuple: $(u, v, \lambda(u), \lambda_{u,v}, \lambda(v))$. gSpan defines an order on these tuples to construct the canonical form. This approach is applicable to directed and undirected graphs. Note that, although the search process in based on gSpan, GRAMI supports generalized patterns. In GRAMI an edge in the candidate pattern corresponds to a distance-constrained path in the input graph. Therefore, the search space is much larger than that of gSpan.

4.2 Support Calculation - Blind Enumeration

For each candidate pattern P generated in the previous step, we

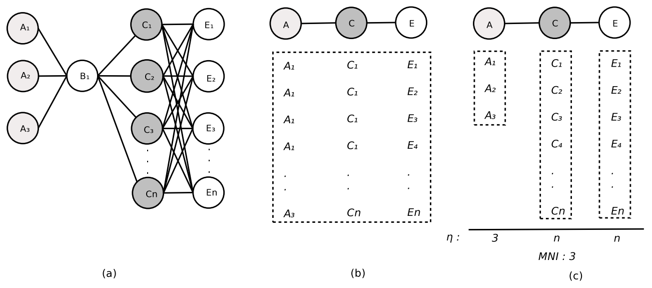


Figure 5: Exponential growth of embeddings in Blind enumeration; Δ is number of hops; $\delta = 2$. (a) Input graph G. (b) Embeddings generated by Blind for pattern $P = \texttt{A} - \texttt{C} - \texttt{E}$. (c) Node images of P. According to MNI, $\sigma_P = 3$.

must calculate its support σ_P; if $\sigma_P < \tau$, then P is pruned. An intuitive algorithm, followed by existing frequent subgraph miners, is to enumerate all embeddings of P in the input graph. We call this algorithm *Blind enumeration*.

Consider the example of Figure 5, assuming that Δ is defined as the number of hops and the distance threshold is $\delta = 2$. For every i, let the labels of all nodes A_i be $\lambda(A_i) = \texttt{A}$; similarly $\lambda(C_i) = \texttt{C}$ and $\lambda(E_i) = \texttt{E}$. Let P be the candidate pattern $\texttt{A} - \texttt{C} - \texttt{E}$. For the $\texttt{A} - \texttt{C}$ path, all A_i and C_j satisfy $\Delta(A_i, C_j) \leq \delta$; the same is true for the $\texttt{C} - \texttt{E}$ path. Therefore, the number of possible embeddings is $|\psi(\texttt{A})| \times |\psi(\texttt{C})| \times |\psi(\texttt{E})| = 3n^2$, where $|\psi(\texttt{A})| = 3$ and $n = |\psi(\texttt{C})| = |\psi(\texttt{E})|$. Figure 5(b) shows the embeddings generated by Blind enumeration. Observe that the algorithm cannot stop after finding τ embeddings, because there may exist overlaps that are not allowed by the MNI support metric (Section 2). For example, the first two embeddings are $\psi_1 = A_1 - C_1 - E_1$ and $\psi_2 = A_1 - C_1 - E_2$. Only one of these embeddings is counted in the support, because there exists overlap for vertices A_1 and C_1.

In general, if P contains k vertices and there are $|\eta|$ vertex images, the worst case complexity of Blind is $\mathcal{O}(|\eta|^k)$. However, based on the definition of the MNI metric, the support of P can be at most $|\eta|$. This is shown in Figure 5(c), where the support in our example is $\sigma_P = |\psi(\texttt{A})| = 3$. In the next section we propose an algorithm that calculates support much more efficiently than Blind.

5. SMART EMBEDDING ENUMERATION

GRAMI models the support calculation process as a constraint satisfaction problem (*CSP*). It avoids enumerating all possible embeddings by solving the CSP only until a pattern is proven frequent or infrequent. Our smart enumeration method allows GRAMI to be 1-3 orders of magnitude faster than existing approaches. The rest of this section describes the basic algorithm, whereas Section 6 discusses further optimizations.

5.1 CSP Model

Definition 5.1 (Constraint Satisfaction Problem) *A* CSP *is represented as a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where: $\mathcal{X}$ is an ordered set of variables $(x_1, \ldots, x_n)$; $\mathcal{D}$ is a set of domains $(D_1, \ldots, D_n)$ where $D_\imath$ represents a finite set of candidate values for variable $x_\imath$; $\mathcal{C}$ is a set of constraints among the variables in $\mathcal{X}$. A solution for the CSP is an assignment $(a_1, \ldots, a_n) \in D_1 \times \cdots \times D_n$ to $\mathcal{X}$, such that all constraints in $\mathcal{C}$ are satisfied.*

An embedding ψ can be mapped to a CSP as follows:

Definition 5.2 (Embedding ψ as CSP) *Given graph G, pattern P, distance function Δ_G and a user-defined distance threshold δ, let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP, where:* (i) *$\mathcal{X} = |V_P| = n$ and there is an one-to-one mapping for each $x_\imath \in \mathcal{X}$ to a vertex $v_\imath \in V_P$.* (ii) *For every $D_\imath \in \mathcal{D}$, $D_\imath = V_G$.* (iii) *$\mathcal{C}$ contains the* ALLDIFF *constraint: $x_\imath \neq x_\jmath$ for all $x_\imath, x_\jmath \in \mathcal{X}, \imath \neq \jmath$.* (iv) *For each $x_\imath \in \mathcal{X}$, there is a constraint $C_\imath \in \mathcal{C}$ stating that $\lambda_P(v_\imath) = \lambda_G(x_\imath)$.* (v) *For each $x_\imath, x_\jmath \in \mathcal{X}$ such that $(v_\imath, v_\jmath) \in E_P$, there is a constraint $C_{\imath,\jmath} \in \mathcal{C}$ stating that $\Delta_G(x_\imath, x_\jmath) \leq \delta$. A solution to $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ corresponds to an embedding ψ.*

In other words, the solution assigns a different vertex of G to each vertex of P, such that the labels of the corresponding vertices match and the distance constraints are satisfied. This problem is a generalization of the subgraph isomorphism test, which is NP-complete [8].

To avoid enumerating all embeddings we use a property [1] of the MNI metric: to calculate the support of a pattern, it is sufficient to resolve an occurrence for every pair of $v \in V_P$ and $u \in V_G$ such that $\psi(v) = u$. In our case we can constrain the problem further, since we do not need the exact value of the support. Instead, we need to prove whether a pattern P is frequent (i.e., appears at least τ times in G). To do so, the corresponding CSP is solved until every variable in $\mathcal{X}$ has at least τ distinct assignments (also called *images*) in valid embeddings of P in G. During this process, some members in domains in $\mathcal{D}$ are pruned because they do not satisfy the CSP constraints. If any domain is left with less than τ members, P is proven infrequent and search is terminated.

5.2 Preparation

GRAMI performs the following steps before solving the CSP:

Frequent node labels. The input graph G is scanned and vertices are grouped by label. For those labels that appear at least τ times, the corresponding vertices are marked frequent. Based on the anti-monotonic property, only these vertices can be part of frequent patterns. Note that infrequent vertices are *not* eliminated from G, since they may be intermediates in the generalized path of some frequent pattern.

Distance precomputation. For each frequent vertex, GRAMI precomputes the set of vertices that are reachable within distance δ. We run a distance-bound Dijkstra algorithm from each frequent vertex to find the shortest path to the reachable vertices, where the path distance is defined by Δ_G; the algorithm terminates when the distance of the shortest path exceeds δ.

Node consistency. This filtration step asserts the unary conditions on the domain $D_\imath$ of each variable $x_\imath \in \mathcal{X}$. Let $v_\imath \in P$ be the pattern node that corresponds to $x_\imath$. Then, for $1 \leq \imath \leq |\mathcal{X}|$: (*i*) Enforce *node label* consistency by restricting $D_\imath$ to contain vertices in G that match the label of $v_\imath$: $\forall u \in D_\imath \Rightarrow \lambda_G(u) = \lambda_P(v_\imath)$. Note that if two vertices in P have the same label then the corresponding domains will include the same vertices of G. Similarly, if a vertex in G has multiple labels, it may appear in multiple domains. To avoid duplicate assignments, the ALLDIFF constraint (Definition 5.2) is applied during the search. (*ii*) Enforce *node degree* consistency: $u \in D_\imath \Rightarrow deg(u) \geq deg(v_\imath)$. Intuitively, a candidate node u must have enough neighbors to match those in the pattern node $v_\imath$. In case of directed graphs degree can be further classified to *in* and *out*. (*iii*) Enforce *neighbor count* consistency: if $v_\imath$ has k neighbors with label ℓ, then $u \in D_\imath$ must have at least k neighbors with label ℓ. Intuitively this is a generalized form of node degree consistency.

Arc consistency. This is a binary check that ensures consistency between the assignments of two variables $x_\imath, x_\jmath \in \mathcal{X}$, with domains $D_\imath$ and $D_\jmath$, respectively. $(x_\imath, x_\jmath)$ is arc consistent if $\forall v_\imath \in$

$D_\imath \exists v_\jmath \in D_\jmath \mid \Delta_G(v_\imath, v_\jmath) \leq \delta$. In other words, every vertex in the domain of $x_\imath$ must be able to reach at least one vertex in the domain of $x_\jmath$ within distance δ. Note that arc consistency is directional, that is, if $(x_\imath, x_\jmath)$ is arc consistent, $(x_\jmath, x_\imath)$ may not be. To ensure arc consistency, all inconsistent vertices are removed from their corresponding domains. A simple algorithm would iterate over all variable pairs $(x_\imath, x_\jmath)$, remove the vertices that do not satisfy arc consistency and repeat the entire process until no change happens to any of the domains. GRAMI uses a more efficient version of the algorithm, called *AC-3* [19]. Instead of checking every domain in every iteration, AC-3 rechecks only the domains that were modified in the previous step.

5.3 Smart Enumeration Algorithm

The smart enumeration algorithm consists of two functions: (*i*) SEARCH searches for a single embedding of a pattern P in G. This function extends previous work [18, 23] on subgraph isomorphism to generalized, more interesting patterns. (*ii*) ISFREQUENT proves whether a pattern is frequent or not, without enumerating all embeddings. To the best of our knowledge, we are the first to map this problem to a CSP and improve significantly the efficiency.

5.3.1 Search for a single embedding

SEARCH (see Algorithm 1) is a recursive function that returns $True$ if there exists at least one embedding of pattern P, in the input graph G. Assume the search has progressed a few steps. Let $PV \subseteq \mathcal{X}$ be the set of variables that have already been instantiated and $FV = \mathcal{X} - PV$ be the set of variables that have not been instantiated yet. The variables in PV constitute a consistent partial solution. Therefore, if FV is empty, a complete solution (i.e., embedding) is found. The function returns $True$ and exits the recursion immediately (line 14).

If there are still uninstantiated variables, the algorithm calls function LOOKAHEAD to perform consistency check. This is necessary because the previous recursion step updated some $D_\imath \in \mathcal{D}$; the update may have generated inconsistent combinations of values in $\mathcal{D}$. LOOKAHEAD prunes the domains of the uninstantiated variables by propagating the constraints, to ensure that future assignments to variables in FV will be consistent. There have been proposed several implementations of LOOKAHEAD in the literature depending on the level of constraint propagation. GRAMI implements two approaches:

Forward checking (FC). This is GRAMI's default look-ahead function. The algorithm checks (*i*) the ALLDIFF constraint (see Definition 5.2); and (*ii*) arc consistency between variables in PV and only those variables in FV that are directly connected to variables in PV. FC checks one step ahead only, to prune non-valid candidate images from the domains of variables in FV.

Really full lookahead (RFLA). Observe that the one-step pruning performed by FC may result in inconsistent states two or more steps ahead. RFLA solves this problem by further propagating the consistencies recursively between the FV and themselves until all domains are pruned from non-valid candidates. By doing that, it prunes early much larger search space compared to FC; the trade-off is higher processing cost. RFLA is used by Ullmann's subgraph isomorphism algorithm [23].

LOOKAHEAD may prune an entire domain in $\mathcal{D}$. In this case no solution is possible based on the current instantiations. Therefore, SEARCH backtracks to the previous variable instantiation. Else (line 7) a variable $x_\imath$ is selected to be instantiated. GRAMI selects the variable that participates in the largest number of constraints. This approach ensures that the unsuccessful branches of the search

Algorithm 1: SEARCH

```
   Input: $PV$ the set of already instantiated variables, $FV$ the set of
          uninstantiated variables, $\mathcal{D}$ the set of current domains
   Output: $True$ if an embedding exists, $False$ otherwise
1  if $FV = \varnothing$ then
2      return $True$;            // All variables are instantiated
3  else
4      $\mathcal{D} \leftarrow$ LOOKAHEAD($\mathcal{D}$);          // Forward check or RFLA
5      if there is no empty domain in $\mathcal{D}$;  // Solution still possible
6      then
7          $\imath \leftarrow$ POPNEXTVARIABLE($FV$); // $x_\imath$ to be instantiated
8          Let $D_\imath \in \mathcal{D}$ be the domain of variable $x_\imath$
9          While $D_\imath \neq \varnothing$ do
10             $v \leftarrow$ POPNODE($D_\imath$); // possible assignment to $x_\imath$
11             $x_\imath \leftarrow v$
12             $\bar{\mathcal{D}} \leftarrow \mathcal{D}$; $\bar{\mathcal{D}}.D_\imath \leftarrow \{v\}$;          // restrict domain
13             $found \leftarrow$SEARCH($PV \cup \{x_\imath\}, FV - \{x_\imath\}, \bar{\mathcal{D}}$)
14             if $found$ then return $True$
15 return $False$;    // at least one domain in $\mathcal{D}$ is exhausted
```

tree are pruned early. A node v from the corresponding domain $D_\imath$ is assigned to $x_\imath$. To check whether this assignment is valid, the algorithm assumes that $D_\imath$ contains only v (line 12) and SEARCH is called recursively (intuitively, all variables in PV are assigned the single value that currently exists in their domains). If an embedding is found then the process is terminated, else it is repeated until all values in $D_\imath$ have been checked, in which case SEARCH backtracks to the previous variable.

5.3.2 Prove a pattern frequent or infrequent

Function ISFREQUENT (see Algorithm 2) returns $True$ if pattern P is frequent. Based on the definition of the MNI metric, to prove P frequent, it suffices to show that there exist at least τ vertices in each domain $D_\imath \in \mathcal{D}$ that are valid assignments (i.e., images) for the corresponding variables in $\mathcal{X}$.

The algorithm starts by enforcing node and arc consistency. If, after the consistency check, any of the domains in $\mathcal{D}$ is left with less than τ candidates the pattern cannot be frequent, so the algorithm returns $False$. For each domain $D_\imath$, the algorithm assumes that the set PV of already instantiated variables contains only the corresponding variable $x_\imath$. Then the algorithm starts counting the number of images for $x_\imath$.

ISFREQUENT has two phases. In *phase one*, it iterates over all vertices $v \in D_\imath$. The algorithm assumes that $D_\imath$ is restricted to v only and calls SEARCH to determine whether there exists at least one embedding with $x_\imath = v$. If SEARCH returns $True$ then the number of images is increased by 1, and the process continues to the next value of $D_\imath$ until the number of images becomes at least τ, in which case the algorithm proceeds to the next domain $D_{\imath+1}$. On the other hand, if SEARCH returns $False$ then there is no embedding that includes v, so v is removed from $D_\imath$ and the algorithm continues with the next vertex in $D_\imath$. Updating $D_\imath$ may trigger new inconsistencies in other domains. For this reason, arc consistency (line 4) is checked again for each domain.

The algorithm implements the following optimization in line 12: Assume that in a previous step $\jmath < \imath$ the SEARCH function was called for some vertex in $D_\jmath$ and the result was a valid embedding that happened to assigned $v \in D_\imath$ to $x_\imath$. The image count was increased for $D_\jmath$ only, but in all domains the vertices that belonged to the assignment (including v) were flagged. In the current step $\imath$, vertex v is recognized as flagged. Therefore the image count is increased for $D_\imath$, without searching again for the embedding.

Observe (line 17) that SEARCH may also time out, if it is not able to find a solution or search the entire space within a specific user-

Algorithm 2: ISFREQUENT

```
    Input: $P$ a pattern,$G$ the input graph ,$\tau$ the frequency threshold
    Output: True if the pattern is frequent, False otherwise
1   $\mathcal{D} \leftarrow$ list of domains $(D_1, D_2, \ldots, D_n)$ for all variables $x_{1\ldots n}$ in the CSP
2   Apply node consistency in $\mathcal{D}$
3   foreach $D_i \in \mathcal{D}$ do
4       Apply arc consistency in $\mathcal{D}$
5       if exists $D_j \in \mathcal{D}$ such that $|D_j| < \tau$ then return $False$
6       $PV \leftarrow \{x_i\}$ ; // variable corresponding to domain $D_i$
7       $FV \leftarrow \{x_1, \ldots, x_n\} - \{x_i\}$ ; // all variables except $x_i$
8       $imageCnt \leftarrow 0$ ;      // number of candidate solutions
9       $Tmp \leftarrow \varnothing$
10      for each $v \in D_i$;                                  // Phase One
11      do
12          if $v$ was part of a solution for some $D_1, \ldots, D_{i-1}$ then
13              $imageCnt++$;         // consider as solution
14          else
15              $\bar{\mathcal{D}} \leftarrow \mathcal{D}; \bar{\mathcal{D}}.D_i \leftarrow \{v\}$; // restrict search domain
16              $found \leftarrow$ SEARCH($PV$, $FV$, $\bar{\mathcal{D}}$)
17              if search timed out then
18                  save $v$ together with its search state to $Tmp$
19                  continue to next node in the domain (go to line 10)
20              else if $found$ then $imageCnt++$; else
21                  $D_i \leftarrow D_i - \{v\}$;     // $v$ is a non-candidate
22          if $imageCnt \geq \tau$ then continue to $D_{i+1}$ (go to line 3)
23      if $|Tmp| + imageCnt \geq \tau$ // Phase Two ($imageCnt < \tau$)
24      then
25          foreach $v \in Tmp$ do
26              $found \leftarrow$ continue SEARCH of $v$ from saved state
27              if $found$ then $imageCnt++$; else
28                  $D_i \leftarrow D_i - \{v\}$;     // $v$ is a non-candidate
29              if $imageCnt \geq \tau$ then continue to $D_{i+1}$ (go to line 3)
30      return $False$;     // Domain exausted but $imageCnt < \tau$
31  return $True$
```

defined time period. In this case, vertex v together with the state of SEARCH are saved temporarily in structure Tmp. In the worst case the complexity of SEARCH is exponential to the size of the pattern. The intuition of the optimization is that the remaining vertices in D_i may produce much faster results that indicate whether there exist τ images of x_i. In such a case, the result of v is irrelevant, so there is no reason to waste more time searching.

After iterating over all vertices in D_i, if image count is still less than τ and Tmp is not empty (i.e., there are candidate nodes that have not been searched fully), the algorithm proceeds to *phase two*. For each $v \in Tmp$, SEARCH is resumed from the saved state. This phase is similar to phase one, but there is no time-out option. Note that, if necessary, ISFREQUENT eventually searches the entire space for each variable; therefore the solution is exact (i.e., no approximation).

6. OPTIMIZATIONS

This section presents four optimizations that are applied to the basic ISFREQUENT algorithm.

6.1 Automorphisms

Automorphism is an embedding of a graph to itself. Automorphisms appear because of symmetries in the graph. Symmetries can be used to prune equivalent branches in the search space. An example is shown in Figure 6, where pattern $P = B - A - B$ and x_1, x_2 are the variables corresponding to vertices v_1, v_2. Assume that while iterating over the domain of x_1, ISFREQUENT finds an embedding $B_1A_1B_3$, meaning that B_1 is an image for v_1 and B_3 is an image for v_2. Because of the symmetry, B_3 is also considered an image of v_1; similarly B_1 is considered an image of v_2. When

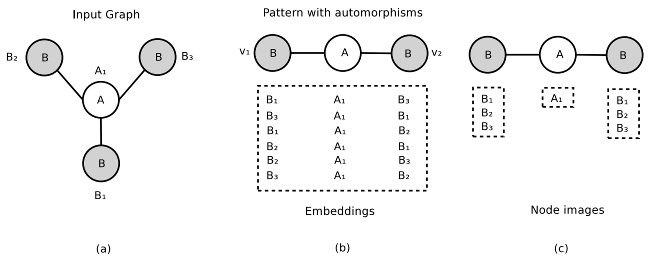


Figure 6: Automorphisms. (a) Input graph G. (b) Embeddings of pattern $B - A - B$. (c) Node images for the pattern.

the iteration for x_1 completes, all images for v_2 are already found, so it is not necessary to iterate over the domain of x_2.

An automorphism is a special case of subgraph isomorphism; therefore it can be detected using function SEARCH. The cost in practice is low, because the domains are small.

6.2 Unique labels

Lemma 6.1 (Unique labels) *Let G be a graph with a single label per vertex. Let P be a pattern, where all of its node labels are distinct: $\lambda_P(v_1) \neq \lambda_P(v_2)$, $\forall (v_1, v_2) \in V_P, v_1 \neq v_2$. To calculate the support of P it is sufficient to refine the domain lists by enforcing node and arc consistencies, without proceeding with a search.*

PROOF: According to Definition 5.2 there are three types of constraints in the CSP: the node and arc constraints and the ALLDIFF constraint. A vertex of G can appear in multiple domains in $\mathcal{D}$ only if it has multiple labels, or if two domains D_i, D_j correspond to vertices of P with the same label. Since none of these conditions is true, any mapping that satisfies node and arc consistency is by default injective, i.e., ALLDIFF is satisfied. Therefore, it is sufficient to check only node and arc consistency. □

6.3 Caching substructures

The pattern search tree is constructed by extending a parent pattern one edge at a time. Since the parent is a substructure of its children, those candidate images that where pruned from the domains of the parent, cannot be valid candidates for any of its children. An example is shown in Figure 7 where pattern P_1 is extended to P_2, P_3 and recursively to P_4. The pruned vertices a_3, b_1, a_3 of P_1 are depicted inside circles in Figure 7(b). This information is pushed down, so a_3, b_1, a_3 are also pruned from all descendants of P_1. This happens recursively; for instance, the vertices pruned because of P_2 are depicted inside dotted circles. Observe that this optimization is applied among different patterns; therefore it is not equivalent to FC and RFLA (Section 5.3.1), which are applied on the variables of a single pattern.

There is also the case of the same substructure appearing in patterns that do not have ancestor - descendant relationship. In the example of Figure 7, P_4 is not a descendant of P_3; however, both contain substructure S_3 (notice the difference in edge $A{-}A$). Since P_3 and P_4 are in different branches, pushing down the pruned vertices is not applicable. Instead, we use a hash table to store the pruned vertices of S_3 (depicted inside squares in the figure). The hash key is the canonical representation (Section 4.1) of S_3. When P_4 is generated, the hash table is searched for matching substructures. If found, the corresponding non-valid candidates are pruned from P_4.

Caching the non-valid candidates, results in a significant performance gain, because the initial pruning was done in smaller patterns. If the same pruning had to be done in the larger patterns the

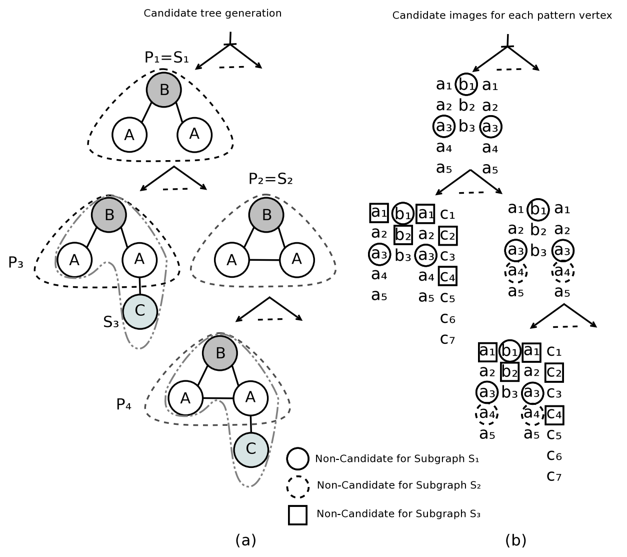

Candidate tree generation
Candidate images for each pattern vertex
P₁=S₁
P₂=S₂
P₃
S₃
P₄
Non-Candidate for Subgraph S₁
Non-Candidate for Subgraph S₂
Non-Candidate for Subgraph S₃
(a)
(b)

Figure 7: (a) Construction of the pattern search tree. (b) Domains of corresponding patterns. Marked vertices are pruned by the Caching Substructures optimization.

cost would increase exponentially. Additionally, a pattern may be eliminated without calling the SEARCH function. In the example, assuming $\tau = 2$, P_4 can be eliminated, because there is only one remaining vertex in the domain of B.

6.4 Partial consistency

The complexity of SEARCH is exponential to the number of variables (i.e., vertices in pattern P) and the sizes of the domains in $\mathcal{D}$. In the worst case, when a solution is not found, the search will examine all possible combinations of domain values. To minimize the problem, we employ a divide-and-conquer strategy by decomposing P into smaller substructures. Those candidate images pruned from the domains of the substructures are also pruned from $\mathcal{D}$. Since the remaining candidate images are fewer, it is easier to solve the CSP for P. Recall that P is generated by extending its parent by one edge. Only the substructures that contain the newly extended edge can prune the search space. Any other substructure appears also in the parent, so pruning has already been done by the Caching Substructures optimization (Section 6.3). Figure 8 shows an example where, after pruning, the domain of K is reduced from 7 candidates to only k_4.

7. USER-DEFINED CONSTRAINTS

Depending on the application, many of the generated patterns may not be interesting (e.g., interactions between the same class labels). The problem was mentioned in [29], which proposes structural constraints for the mining of isomorphic subgraphs. In the case of generalized distance-constraint patterns, the problem becomes more pronounced because the result set is much larger. To allow the user to focus on the interesting patterns, be developed CGRAMI, a version of our framework that supports two types of user-defined constraints: (*i*) *Structural*, such as "the number of vertices in pattern P should be at most α"; and (*ii*) *Semantic*, such as "P must not contain some specific labels". Although not a requirement, it is desirable the user-defined constraints to be anti-monotonic. In such a case, the constraints can be pushed down in the search tree and prune early large parts of the search space, thus accelerating the process. Table 1 presents a set of useful structural and semantic antimonotonic constraints that are implemented in CGRAMI.

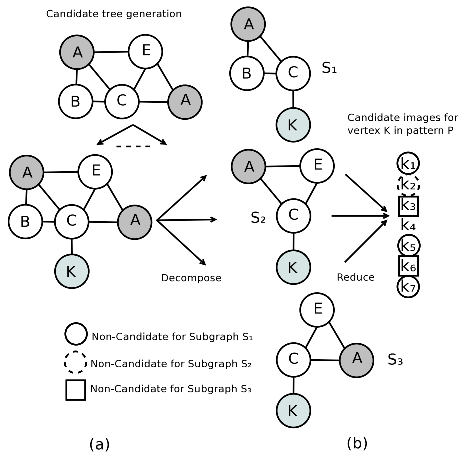

Candidate tree generation
S₁
P
S₂
S₃
Candidate images for vertex K in pattern P
Decompose
Reduce
Non-Candidate for Subgraph S₁
Non-Candidate for Subgraph S₂
Non-Candidate for Subgraph S₃
(a)
(b)

Figure 8: (a) Pattern P is generated by extending its parent with edge $C - K$. (b) P is decomposed in overlapping subgraphs S_1, S_2, S_3 containing the newly extended edge $C - K$. All candidate images for K, except k_4, are pruned.

8. APPROXIMATE MINING

Mining the complete set of frequent patterns is very expensive in large graphs because it is dominated by the NP-complete subgraph isomorphism problem. Motivated by this, we developed AGRAMI, an approximate version of our framework, which can support large graphs with reasonable efficiency. To maintain the quality of results, AGRAMI will not return any infrequent pattern (i.e., no false positives), although it may miss some frequent ones (i.e., false negatives). To achieve this, we modified the way ISFREQUENT handles time-outs (line 17 in Algorithm 2) as follows: we set the time-out to occur after $f(\varepsilon)$ iterations in SEARCH. If an embedding is found before the time-out, the image count is updated as normal. On the other hand, if a time-out occurs it is assumed that SEARCH returned $False$. If enough time-outs occur for the vertices of a specific domain, then the image counter will be less than τ, so the pattern is considered infrequent. $f(\varepsilon)$ is defined as:

$$f(\varepsilon) = \varepsilon^n \prod_1^n |D_\imath| + \beta \qquad (1)$$

where β is a constant, $D_\imath$ are the domains of the variables, n is the number of variables and $0 < \varepsilon \le 1$ is a user-defined approximation parameter. $\prod_1^n |D_\imath|$ grows exponentially; thus it has to be bounded by an exponential weight ε^n. Increasing ε decreases the approximation error at the expense of longer execution time. When $\varepsilon = 1$, $f(1)$ becomes an upper bound of the worst case complexity of SEARCH, therefore AGRAMI becomes equivalent to the exact solution.

9. EXPERIMENTAL EVALUATION

In this section we evaluate our framework in terms of efficiency and quality of results. For simple subgraph isomorphism (i.e., patterns that match exactly), we show that GRAMI is up to 3 orders of magnitude faster than the existing approaches, due to our CSP-based count operator. We also show that by using distance-constrained paths to generalize the notion of subgraph isomorphism, GRAMI discovers many complex and interesting interactions that are missed by existing methods. All experiments were run using

Table 1: Definitions of the antimonotonic structural and semantic constraints for pattern P, implemented in CGRAMI

Structural constraints	
$C_{vertexSize}(P) \equiv (\vert V_P\vert \leq \alpha)$	The number of vertices should not exceed α.
$C_{edgeSize}(P) \equiv (\vert E_P\vert \leq \alpha)$	The number of edges should not exceed α.
$C_{maxDegree}(P) \equiv (\max(deg(V_P)) \leq \alpha)$	The maximum vertex degree is α.
Semantic constraints	
$C_{vertex}(P) \equiv (\forall v \in V_P \Rightarrow \lambda(v) \in L)$	P contains only labels from L.
$\bar{C}_{vertex}(P) \equiv (\forall v \in V_P \Rightarrow \lambda(v) \notin L)$	P does not contains any label from L.
$C_{edge}(P) \equiv (\forall (v_\imath, v_\jmath) \in E_P \Rightarrow (\lambda(v_\imath), \lambda(v_\jmath)) \in E)$	P contains only edges from E.
$\bar{C}_{edge}(P) \equiv (\forall (v_\imath, v_\jmath) \in E_P \Rightarrow (\lambda(v_\imath), \lambda(v_\jmath)) \notin E)$	P does not contain any edges from E.
$C_{subgraph}(P, P') \equiv (\neg subgraph(P', P))$	Pattern P must not contain a specific subgraph P'.
$C_{count}(P) \equiv (\forall v \in V_P \Rightarrow \max(count(\lambda(v))) \leq \alpha)$	A label cannot occur more than α times in P.

Java JRE v1.6.0. on a Linux (Fedora 11) server with two quad-core 2.6GHz Xeon CPUs, 24GB RAM and 1TB disk; our code runs on only one core. We used the following real graph datasets:

Aviation[1] network: It is extracted from the aviation safety database and was used in [4, 17]. It consists of 101,185 nodes and 98,576 edges. There are 6,173 distinct node labels and 51 edge labels. Since we focus on weighted graphs, we replace edge labels with weight $w = 1$. Following [17] we consider the edges undirected.

Cora[2] citation graph: It is a directed graph consisting of 2,708 publications (nodes) in the field of Machine Learning. There are 5,429 citation links (edges); all edge weights are set to 1. Each node has a single label representing an area of Machine Learning; there are 7 distinct labels.

CiteSeer[2] citation graph. It is a directed graph consisting of 3,312 publications (nodes) and 4,732 citation links (edges). Each node has a single label representing an area of Computer Science; there are 6 distinct labels. Each edge has a normalized weight (0 to 100) that measures the dissimilarity between the corresponding pair of publications.

Microsoft co-authorship (MiCo[3]) graph: We crawled the Computer Science collaboration graph from Microsoft Academic and generated an undirected graph with 100,000 nodes and 1,080,298 edges. Each node represents an author and can have multiple labels representing the author's field of interest. There are 29 distinct labels and a total of 183,578 labels. An edge represents collaboration between two authors. We followed [30] to assign to each edge (u, v) a weight $w_{u,v} = \frac{\max_{\forall ui,vj}(C(u_i,v_j))}{C(u,v)}$, where $C(u, v)$ is the number of co-authored papers between authors u and v.

MiCo-S: We selected randomly 10,000 nodes from MiCo. The resulting dataset is multilabeled (22,393 total labels); it contains 54,581 edges and 29 distinct labels.

9.1 Frequent Isomorphic Subgraphs

First we focus on traditional frequent subgraph mining using subgraph isomorphism (i.e., exact match). Δ is defined as the shortest path (in terms of number of hops) between two nodes, and $\delta = 1$. We evaluate the performance of plain GRAMI (Section 5) against GRAMI with optimizations (Section 6): Automorphisms, Partial Consistency, Caching Substructures and all optimizations enabled. We use the Cora, CiteSeer and MiCo-S[4] datasets, and measure the total execution time versus the support threshold τ. Obviously, the number of discovered patterns depends on τ and the dataset. Therefore, for each dataset we selected a different range for τ to demonstrate clearly the effect of the optimizations.

The results are shown in Figure 9. As expected, when τ decreases, the number of frequent patterns, as well as the running time, increase. Few of the patterns extracted from Cora and CiteSeer have automorphisms, therefore this optimization does not offer any benefit. Most of the patterns mined from MiCo-S, on the other hand, have automorphisms, so the performance gain is significant (i.e., up to 4 times faster; note the logarithmic scale). In contrast, Partial Consistency is not useful in MiCo-S because most patterns contain relatively few nodes, but the optimization is effective only for larger patterns. This is evident in CiteSeer, especially for low values of τ that tend to generate large patterns; in this case, Partial Consistency is 28% faster than plain GRAMI. The optimization that achieved substantial gains in all datasets is Caching Substructures. Recall that it identifies and prunes early redundant parts of the embedding search tree. In Cora, for instance, it is up to 6 times faster than plain GRAMI. Finally, by combining all optimizations, additional gain can be achieved in most cases. The most pronounced effect is in MiCo-S, where GRAMI-All is almost an order of magnitude faster than the plain version.

In the rest of the section we compare GRAMI-All against existing methods for frequent subgraph mining. Existing methods enumerate all embeddings in each iteration and either store those embeddings (to be used in the next iteration), or repeat the entire embedding construction process in each iteration. The trade-off is space versus speed. We implemented such a method (called *Blind* in the following) and opted for speed by storing the intermediate embeddings on disk, since they could not fit in memory. We also implemented a version of GRAMI with the Really Full Look Ahead (*RFLA*) approach (Section 5.3.1) that prunes aggressively large parts of the search tree.

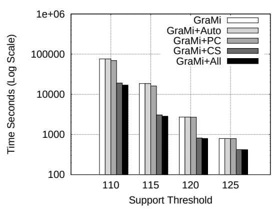
The results are shown in Figure 10. The number of intermediate embeddings grows exponentially when the support threshold τ decreases. Since Blind needs to enumerate all embeddings, its running time grows exponentially (note the logarithmic scale) and becomes prohibitively expensive below a certain threshold. Observe that the running time of GRAMI-All is affected exponentialy, too. This is inevitable, since the number of results and their sizes also increases exponentially with decreasing τ. However, our smart count operator does not need to enumerate all intermediate embeddings, so the rate of increase is much lower. Also, in contrast to

[1] http://ailab.wsu.edu/subdue

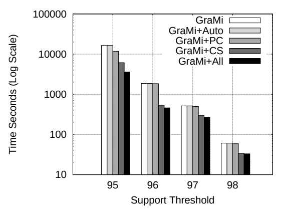
[2] http://www.cs.umd.edu/projects/linqs/projects/lbc

[3] Crawled from http://academic.research.microsoft.com

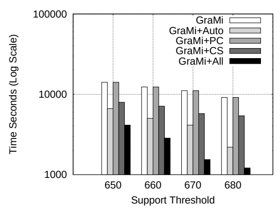
[4] The competitor (Blind) could not handle the entire MiCo-S. In this section only, we simplified MiCo-S by selecting randomly only one label per node and limiting patterns to 10 nodes maximum.



(a) Cora dataset

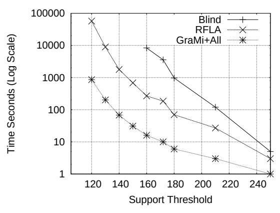


(b) CiteSeer dataset

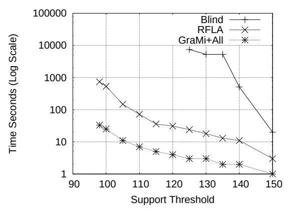


(c) MiCo-S (single labeled)

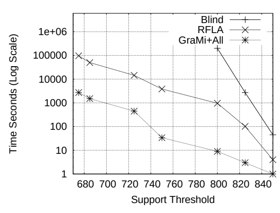
Figure 9: Traditional subgraph isomorphism. Comparison between plain GRAMI (execution time vs support threshold τ) and the effect of optimizations. Auto: Automorphisms; PC: Partial Consistency; CS: Caching Substructures; All: all optimizations enabled. GRAMI-All is 2 to 8 times faster than GRAMI.



(a) Cora dataset



(b) CiteSeer dataset



(c) MiCo-S (single labeled)

Figure 10: Traditional subgraph isomorphism. Execution time vs support threshold τ. GRAMI-All is up to three orders of magnitude faster than Blind and more than one order of magnitude faster than RFLA.

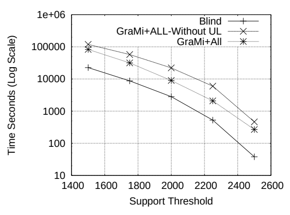


Figure 11: Traditional subgraph isomorphism; Aviation dataset; execution time vs support threshold τ. Blind is the fastest because the dataset contains many small non-connected components. UL is the unique labels optimization.

Blind, the memory requirements of our approach are very low. The results show that GRAMI-All is up to 3 orders of magnitude faster than Blind. This allows our approach to scale to lower τ values and discover larger and more interesting frequent subgraphs. For example, in the Cora dataset Blind cannot extract frequent subgraphs larger than 7 nodes, whereas GRAMI-All discovers much larger subgraphs, up to 19 nodes. Figure 10 also shows that RFLA, although faster than Blind, is at least an order of magnitude slower than GRAMI-All. The reason is that, although RFLA prunes more parts of the search space, the process of identifying branches to be pruned adds significant overhead to each iteration, which makes it not cost-effective.

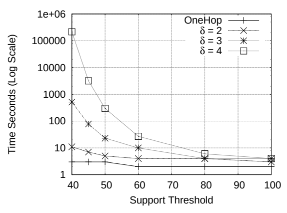
We also run an experiment using the Aviation dataset. Aviation has at least an order of magnitude more nodes than Cora, CiteSeer and MiCo-S, so intuitively is should be prohibitively slow with Blind. Interestingly, Kuramochi and Karypis [17] show that extracting frequent labeled subgraphs was achievable in reasonable time by generating all the embeddings. Figure 11 shows the results for Blind versus GRAMI-All; Blind is 3 to 7 times faster. To explain this, we analyzed the dataset and found that it is a very sparse graph, with 0.97 edges per node on average. It consists of 2,608 non-connected very small star-shaped components, with 39 nodes per component on average. This means that the search space is very restricted, so it is feasible for Blind to enumerate and store the few resulting embeddings. GRAMI-All, in contrast, does not store intermediate embeddings, so it has to redo part of the work in each iteration. Note that it is questionable whether datasets like Aviation need a mining method that targets single large graphs, since the same frequent subgraphs could be mined by a much simpler transactional miner [1]. Another observation in Figure 11 is that the Unique Labels optimization (Section 6) offers significant gain. This is due to the fact that most subgraphs extracted from this dataset have unique labels; therefore it suffices to apply node and arc consistency only in order to decide whether they are frequent.

9.2 Generalized Distance-constraint Patterns

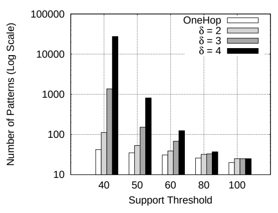
In the rest of the paper we focus on our generalized distance-constrained definition of subgraph isomorphism. Let u, v be nodes. Following [30], the distance $\Delta(u, v)$ is evaluated as the sum of the weights on the path from u to v; if v can be reached from u through multiple paths, the minimum of the corresponding distances is selected. All experiments use GRAMI-All (i.e., all optimizations are enabled). We compare the running time of the algorithm and the quality of results, for different values of the distance threshold δ. Since each dataset has different method of assigning weights to edges, the values of δ differ with the dataset. We also compare against traditional subgraph isomorphism by using GRAMI-All with the settings from Section 9.1 (i.e., Δ is the number of hops and $\delta = 1$); we call this case *OneHop*. The resulting frequent patterns from the generalized definition are a superset of those from OneHop. Many of the resulting patterns are not interesting, since they contain non-surprising interactions among nodes with the same label. To focus on the interesting patterns we imposed the following semantic constraint: a label can appear at most α times in a pattern, where α depends on the dataset; for Cora, $\alpha = 3$, whereas for MiCo-S[5], $\alpha = 1$.

Figure 12 shows the results for Cora. Execution time (Figure 12(a))

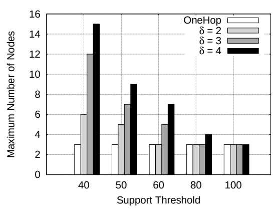
[5]We used the whole MiCo-S with multiple labels per node. Therefore, although $\alpha = 1$, the Unique Labels optimization cannot be used (Lemma 6.1).



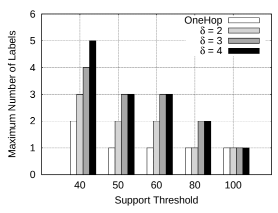
(a) Execution time



(b) Number of patterns

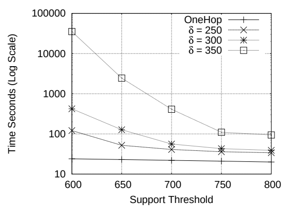


(c) Maximum size of patterns

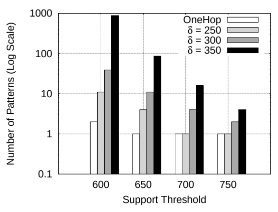


(d) Maximum number of labels

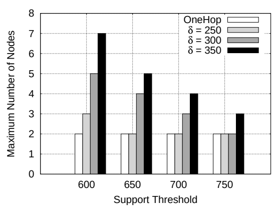
Figure 12: Generalized distance-constraint pattern mining. Cora dataset; execution time and quality of results (x-axis represents the support threshold τ). The GRAMI framework discovers higher quality results compared to existing methods (i.e., OneHop).



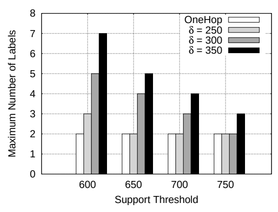
(a) Execution time



(b) Number of patterns



(c) Maximum size of patterns



(d) Maximum number of labels

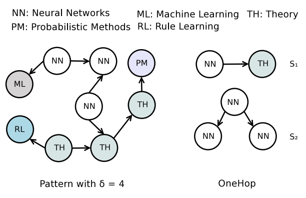
Figure 13: Generalized distance-constraint pattern mining. MiCo-S dataset; execution time and quality of results versus τ.

increases exponentially when the distance and threshold constraints are relaxed (i.e., δ increases and τ decreases), because in both cases the number of frequent patterns (Figure 12(b)) increases exponentially, too. The reason for relaxing the distance and threshold constraints is to discover more interesting frequent patterns. For example, let $\tau = 40$. Figure 12(c) shows that for $\delta = 4$ our framework discovers patterns with as many as 15 nodes. In comparison, OneHop can find patterns with up to only 3 nodes; therefore many interesting interactions are missed. Figure 12(d) also supports the claim that the quality of the results produced by our framework is higher than the existing approaches as it is reveals more interactions among more and different labels. For example, for $\tau = 40$ and $\delta = 4$ our framework identifies patterns with up to 5 distinct labels, whereas OneHop generates patterns with only two distinct labels. For $\tau > 40$ all patterns generated by OneHop are trivial (i.e., contain only one distinct label). Figure 13 shows the results for the MiCo-S dataset. Again, the patterns generated by our framework are of higher quality. Also, the number of patterns discovered by GRAMI is larger; OneHop can find only one pattern for $\tau \geq 650$. The results for CiteSeer were similar and are omitted.

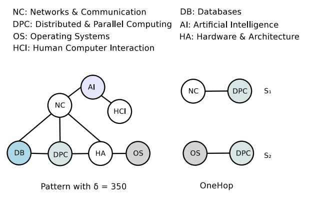
The examples in Figure 14 demonstrate the type of patterns discovered by GRAMI and OneHop. In the Cora dataset for instance, when $\delta = 4$, GRAMI identifies an interesting citation pattern that involves Neural Networks, Machine Learning, Theory, Probabilistic Methods and Rule Learning, and appears at least $\tau = 40$ times. OneHop, on the other hand, finds only one small pattern with two distinct labels (i.e., NN $\rightarrow$ TH), whereas all other patterns are trivial (i.e., contain only one distinct label). Similarly, for the MiCo-S dataset and $\delta = 350$ GRAMI finds a complex collaboration pattern with 7 nodes and 7 distinct labels that appears at least $\tau = 600$ times. OneHop can only discover simple, less interesting patterns with up to 2 nodes, like the ones shown in Figure 14(b).

9.3 AGRAMI and CGRAMI

The next experiment evaluates AGRAMI, the approximate version of our framework, against the exact algorithm on the Cora dataset. We define Δ to be the number of hops and $\delta = 1$. We set constant $\beta = 10^5$ in $f(\varepsilon)$ (see Equation 1) and vary the approximation parameter ε. Figure 15 shows the execution time and the recall, defined as the number of patterns found by AGRAMI over the number of patterns found by the exact one. The results demonstrate that AGRAMI can be more than an order of magnitude faster the exact solution, while achieving at least 93.5% recall. Note that precision is always 100%, because AGRAMI does not generate false positives.



(a) Cora; $\alpha = 3, \tau = 40$



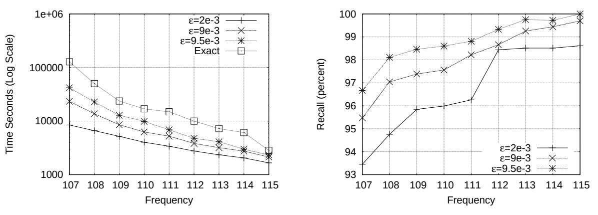
NC: Networks & Communication
DPC: Distributed & Parallel Computing
OS: Operating Systems
HCI: Human Computer Interaction
DB: Databases
AI: Artificial Intelligence
HA: Hardware & Architecture
Pattern with δ = 350
OneHop
S₁
S₂

(b) MiCo-S; $\alpha = 1, \tau = 600$

Figure 14: Example patterns discovered by GRAMI and OneHop. GRAMI identifies complex interactions. OneHop generates smaller, less interesting patterns.

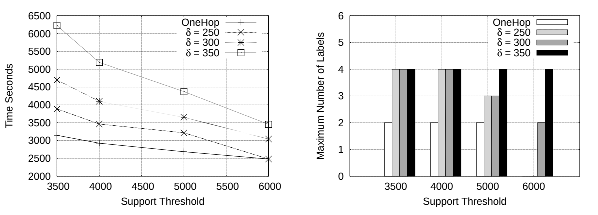
The final experiment combines AGRAMI with the constrained version CGRAMI. We enforced the following constraints: (*i*) Semantic: a label should not occur more than $\alpha = 1$ times in a pattern; and (*ii*) Structural: a pattern should not have more than 3 edges. We used the large MiCo dataset and compared OneHop against distance-constrained patterns with various δ thresholds. Figure 16(a) shows the execution time; as expected, time increases with δ. With the most demanding setting, our framework requires roughly 1h:45min, which is fast given the size of the graph. Figure 16(b) shows the maximum number of labels in the resulting patterns; distance-constrained patterns exhibit more diversity compared to OneHop. Moreover, Table 2 shows the total number of discovered patterns; OneHop generates very few, or none, results.

Finally, Figure 17 depicts examples of the mined patterns. With $\delta = 250$ our framework discovered a pattern that relates research in Networks with Distributed Computing, Scientific Computing and Theory; the pattern appears at least $\tau = 4000$ times. In contrast,



(a) Execution time (b) Recall (%)

Figure 15: AGRAMI on the Cora dataset



(a) Execution time (b) Maximum number of labels

Figure 16: AGRAMI+CGRAMI; $\varepsilon = 10^{-6}$; MiCo dataset.

OneHop managed to extract only two non-trivial patterns, which represent more-or-less expected relationships (i.e., Networks with Distributed Computing *or* Scientific Computing with Algorithms).

10. CONCLUSIONS

Many important applications, ranging from bioinformatics to social network study, and from personalized advertisement (e.g., recommendation systems) to security (e.g., identification of terrorist groups), depend on graph mining. This paper introduced GRAMI a versatile framework for discovering generalized patterns in a single large graph, a significantly more difficult problem compared to the usual case of mining a set of small graphs. The main contributions of this work are: (*i*) the generalization of the notion of frequent subgraphs to distance-constrained patterns, which allows the discovery of interesting patterns that are missed by existing approaches; and (*ii*) the modeling of the count operation as a constraint satisfaction problem, which enables the efficient implementation of the framework. We also implemented a version that supports structural and semantical constraints and an approximate version that scales to large graphs. Our experimental results with real datasets demonstrate that GRAMI is up to 3 orders of magnitude faster than existing approaches, while discovering larger and more interesting frequent patterns. We are currently working on a parallel version of GRAMI. We are also planning to formalize a language that will allow the user to express arbitrary constraints.

11. REFERENCES

[1] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *Proc. of PAKDD*, pages 858–863, 2008.

[2] C. Chen, X. Yan, F. Zhu, and J. Han. gapprox: Mining frequent approximate patterns from a massive network. In *Proc. of ICDM*, pages 445–450, 2007.

[3] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *Proc. of ICDE*, pages 913–922, 2008.

[4] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[5] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *Proc. of ICDM*, pages 35–, 2003.

[6] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proc. of ACM-SIGMOD*, pages 431–442, 1999.

Table 2: Number of patterns extracted from the MiCo dataset with approximations and constraints imposed.

	$\tau = 3500$	$\tau = 4000$	$\tau = 5000$	$\tau = 6000$
OneHop	6	2	1	0
$\delta = 250$	253	34	3	0
$\delta = 300$	1855	489	11	2
$\delta = 350$	5143	2314	129	10

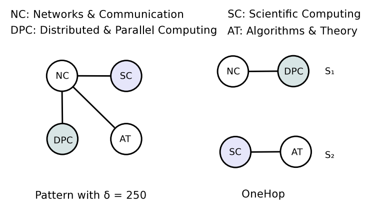


Figure 17: AGRAMI+CGRAMI; $\tau = 4000$; MiCo dataset.

[7] C. Domshlak, R. I. Brafman, and S. E. Shimony. Preference-based configuration of web page content. In *Proc. of Int. joint conf. on Artificia intelligence - Volume 2*, pages 1451–1456, 2001.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[9] S. Ghazizadeh and S. S. Chawathe. Seus: Structure extraction using summaries. In *Proc. of Int. Conf. on Discovery Science*, pages 71–85, 2002.

[10] V. Guralnik and G. Karypis. A scalable algorithm for clustering sequential data. In *Proc. of ICDM*, pages 179–186, 2001.

[11] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proc. of ICDM*, pages 549–, 2003.

[12] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *Proc. of ACM-SIGKDD*, pages 581–586, 2004.

[13] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of PKDD*, pages 13–23, 2000.

[14] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering frequent topological structures from graph datasets. In *Proc. of ACM-SIGKDD*, pages 606–611, 2005.

[15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of ICDM*, pages 313–320, 2001.

[16] M. Kuramochi and G. Karypis. Grew-a scalable frequent subgraph discovery algorithm. In *Proc. of ICDM*, pages 439–442, 2004.

[17] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11:243–271, 2005.

[18] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12:403–422, 2002.

[19] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[20] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proc. of ACM-SIGKDD*, pages 647–652, 2004.

[21] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *Proc. of ICDE*, pages 844–855, 2009.

[22] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. *TKDD*, 4:10:1–10:42, 2010.

[23] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of ACM*, 23:31–42, 1976.

[24] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *Proc. of ACM-SIGKDD*, pages 316–325, 2004.

[25] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *Proc. of ACM-SIGMOD*, pages 433–444, 2008.

[26] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. of ICDM*, pages 721–, 2002.

[27] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. of ACM-SIGKDD*, pages 286–295, 2003.

[28] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. of ACM-SIGMOD*, pages 335–346, 2004.

[29] F. Zhu, X. Yan, J. Han, and P. S. Yu. gprune: a constraint pushing framework for graph pattern mining. In *Proc. of PAKDD*, pages 388–400, 2007.

[30] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *PVLDB*, 2:886–897, 2009.

[31] R. Zou and L. B. Holder. Frequent subgraph mining on a single large graph using sampling techniques. In *Proc. of Workshop on Mining and Learning with Graphs*, pages 171–178, 2010.