GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph

Mohammed Elseidy
Ecole Polytechnique Fédérale de Lausanne
mohammed.elseidy@epfl.ch

Ehab Abdelhamid
King Abdullah University of Science and Technology
ehab.abdelhamid@kaust.edu.sa

Spiros Skiadopoulos [*]
University of Peloponnese
spiros@uop.gr

Panos Kalnis
King Abdullah University of Science and Technology
panos.kalnis@kaust.edu.sa

ABSTRACT

Mining frequent subgraphs is an important operation on graphs; it is defined as finding all subgraphs that appear frequently in a database according to a given frequency threshold. Most existing work assumes a database of many small graphs, but modern applications, such as social networks, citation graphs, or protein-protein interactions in bioinformatics, are modeled as a single large graph. In this paper we present GRAMI, a novel framework for frequent subgraph mining in a single large graph. GRAMI undertakes a novel approach that only finds the *minimal* set of instances to satisfy the frequency threshold and avoids the costly enumeration of *all* instances required by previous approaches. We accompany our approach with a heuristic and optimizations that significantly improve performance. Additionally, we present an extension of GRAMI that mines frequent patterns. Compared to subgraphs, patterns offer a more powerful version of matching that captures transitive interactions between graph nodes (like friend of a friend) which are very common in modern applications. Finally, we present CGRAMI, a version supporting structural and semantic constraints, and AGRAMI, an approximate version producing results with no false positives. Our experiments on real data demonstrate that our framework is up to 2 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 chemical, bioinformatics, computer vision, social networks, text retrieval and web analysis. Mining frequent subgraphs is a central and well studied problem in graphs, and plays a critical role in many data mining tasks that include graph classification [9], modeling of user profiles [11], graph clustering [15], database design [10] and index selection [31]. The goal of frequent subgraph mining is to find subgraphs whose appearances exceed a user defined threshold. This is useful in several real life applications. Consider for example protein-protein interaction (PPI) networks [5]. These networks are graphs where nodes represent proteins (and are labeled with their functionality) and edges represent interactions between these proteins. Such graphs are constantly updated to include new proteins and their interactions. A critical task for biologists is to predict the functionality (and add the corresponding label) of a new protein without experimental testing. The above task may be accurately preformed by mining frequent subgraphs with similar interactions to the new protein [5].

Consider the collaboration graph G of Fig. 1 and a user interested to mine important collaborations among authors. Typically, in such graphs, frequent subgraphs are most likely to show collaborations among authors having the same field of work (i.e., collaborations among DB researchers). In order to reveal more interesting subgraphs, the user would progressively reduce the frequency threshold until subgraphs showing interdisciplinary collaborations are discovered (i.e., among AI, DB and IR researchers). Lowering the frequency threshold increases the number of qualified intermediate results and intensifies the already expensive computations of the mining process. For example, a state-of-the-art method for frequent subgraph mining crashes after a day consuming 192GB for an input graph of 100K nodes and 1M edges. Therefore, the development of efficient frequent subgraph mining algorithms that support large graphs and low frequency thresholds is very crucial.

Existing literature considers two settings: transactional and single graph. The *transactional* case assumes a database of many, relatively small graphs, where each graph represents a transaction [18, 29]. A subgraph is frequent if it exists in at least τ transactions, where τ is a user-defined threshold. In this paper, the focus is on the *single-graph* setting that considers one large graph [17, 19, 20]. For this setting, a subgraph is frequent if it has at least τ appearances in the graph. Such a context is required in many modern applications, including social and PPI networks. The *single-graph* setting is a generalization of the transactional one, since a set of small graphs can be considered as connected components within a single large graph. Detecting frequent subgraphs in a single graph is more complicated because multiple instances of identical subgraphs may overlap. Moreover, it is more computationally demanding because complexity is exponential in the graph size.

The most straightforward method to evaluate frequency of a subgraph S in a graph G is to look for *isomorphisms* of S in G [12, 16, 19, 20]. Isomorphisms are exact matches of S in G that pair nodes, edges and labels. For example, in the collaboration graph G of Fig. 1, subgraph S_1 has three isomorphisms.

A typical method to mine frequent subgraphs in a single graph, is a *grow-and-store* method that proceeds with the following steps:

1. Find all nodes that appear at least τ times and store all of their appearances.
2. Extend the stored appearances to construct larger potential frequent subgraphs, evaluate their frequency, and store all the appearances of the new frequent subgraphs.

[*]Supported by EU/Greece Research Funding Program: Thales

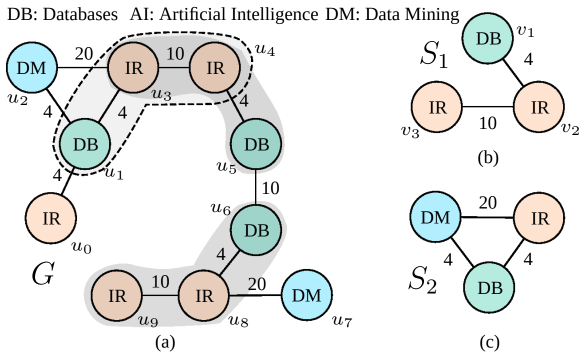


Figure 1: (a) A collaboration graph G; nodes correspond to authors (labeled with their field of work) and edges represent co-authorship (labeled with number of co-authored papers). (b) and (c) Subgraphs S_1 and S_2.

3. Repeat Step 2 until no more frequent subgraphs can be found.

Existing approaches such as SIGRAM [20] use variations of this grow-and-store method. These approaches take advantage of the stored appearances to evaluate the frequency of a subgraph. The main bottleneck of such algorithms is the creation and storage of all appearances of each subgraph. The number of such appearances depends on the size and the properties of the graph and the subgraph; it can be prohibitively large to compute and store, rendering grow-and-store solutions infeasible in practice.

In this work, we propose GRAMI (GRA*ph* MI*ning*); a novel framework that addresses the frequent subgraph mining problem. GRAMI undertakes a novel approach differentiating it from grow-and-store methods. First, it stores only the templates of frequent subgraphs, but not their appearances on the graph. This eliminates the limitations of the grow-and-store methods and allows GRAMI to mine large graphs and support low frequency thresholds. Also, it employs a novel method to evaluate the frequency of a subgraph. More specifically, GRAMI models the frequency evaluation as a constraint satisfaction problem (*CSP*). At each iteration, GRAMI solves the CSP until it finds the *minimal* set of appearances that are enough to evaluate subgraph frequency, and it ignores all remaining appearances. The process is repeated by extending the subgraphs until no more frequent subgraphs can be found.

Solving the CSP can still take exponential time in the worst case. In order to support large graphs in real-life applications, GRAMI employs a heuristic search and a series of optimizations that significantly improve performance. More specifically, GRAMI introduces novel optimizations that *(a)* prune large portions of the search space, *(b)* prioritize fast and postpone slow searches and *(c)* take advantage of special graph types and structures. By avoiding the exhaustive enumeration of appearances and using the proposed optimizations, GRAMI supports larger graphs and smaller frequency thresholds than existing approaches. For example, to compute the frequent patterns of the 100K nodes/1M edges graph that the state-of-the-art grow-and-store method crashed after a day, GRAMI needs only 16 minutes.

Additionally, we propose three extensions to the original GRAMI framework. The first one considers graphs such as social or research networks, that may contain incomplete information and transitive relationships. In such cases *indirect* relationships (like a friend of a friend) reveal neighborhood connectivity and proximity information. To explore these relationships, *patterns* were introduced [4, 17, 34]. Patterns establish a more powerful definition of matching, than subgraphs, that captures indirect connections by replacing edges with paths. To mine frequent patterns, we have appropriately extended GRAMI. For instance in Fig. 1, GRAMI may also consider $u_5 \overset{\dots}{—} u_8 \overset{10}{—} u_9$ to be a match of S_1 since u_5 (labeled DB) is indirectly connected to u_8 (labeled IR). The second extension, CGRAMI, allows the user to define a set of *constraints*, both structural (e.g., the subgraph is allowed to have up to α edges) and semantic (e.g., a particular label cannot occur more than α times in the subgraph). The constraints are used to prune undesirable matches and limit the search space. The final extension, AGRAMI, is an *approximate* version, which approximates subgraph frequencies. The approximation method may miss some frequent subgraphs (i.e., has false negatives), but the returned results are *not* approximate (i.e., does not have false positives).

Noteworthily, GRAMI and its extensions support directed and undirected graphs and may be applied to both single and multiple labels (or weights) per node and edge.

In summary, our main contributions are:

- We propose GRAMI, a novel framework to mine frequent subgraphs in a large single graph. GRAMI is based on a novel idea that refrains from computing and storing large intermediate results (appearances of subgraphs). A key part of the underlying idea is to evaluate the frequency of subgraphs using CSP.
- We offer a heuristic search with novel optimizations that significantly improve GRAMI's performance by pruning the search space, postponing searches, and exploring special graph types.
- We develop a variation of GRAMI that is able to mine frequent patterns, a more powerful version of matching that is required in several modern applications.
- We present CGRAMI, a version that supports structural and semantic constraints, and AGRAMI, an approximate version which produces results with no false positives.
- We experimentally evaluate the performance of GRAMI and demonstrate that it is up to 2 orders of magnitude faster than existing methods in large real-life graphs.

The rest of the paper is organized as follows. Section 2 formalizes the problem. Section 3 presents GRAMI and its optimizations. Section 4 discusses the extensions of GRAMI. Section 5 presents the experimental evaluation. Section 6 surveys related work, and Section 7 concludes.

2. PRELIMINARIES

A *graph* $G = (V, E, L)$ consists of a set of nodes V, a set of edges E and a labeling function L that assigns labels to nodes and edges. A graph $S = (V_S, E_S, L_S)$ is a *subgraph* of a graph $G = (V, E, L)$ iff $V_S \subseteq V$, $E_S \subseteq E$ and $L_S(v) = L(v)$ for all $v \in V_S \cup E_S$. Fig. 1a illustrates an example of a collaboration graph. Node labels represent author's field of work (e.g., Databases) and edge labels represent the number of co-authored papers. To simplify presentation, all examples illustrate undirected graphs with a single label for each node. However, the proposed methods also support directed graphs and multiple labels per node/edge.

Definition 1 *Let $S = (V_S, E_S, L_S)$ be a subgraph of a graph $G = (V, E, L)$. A* subgraph isomorphism *of S to G is an injective function $f : V_S \to V$ satisfying* (a) *$L_S(v) = L(f(v))$ for all nodes $v \in V_S$, and* (b) *$(f(u), f(v)) \in E$ and $L_S(u, v) = L(f(u), f(v))$ for all edges $(u, v) \in E_S$.*

Intuitively, a subgraph isomorphism is a mapping from V_S to V such that each edge in E is mapped to a single edge in E_S and vice versa. This mapping preserves the labels on the nodes and edges. For example in Fig. 1, subgraph S_1 ($v_1 \overset{4}{—} v_2 \overset{10}{—} v_3$) has three isomorphisms with respect to graph G, namely $u_1 \overset{4}{—} u_3 \overset{10}{—} u_4$, $u_5 \overset{4}{—} u_4 \overset{10}{—} u_3$ and $u_6 \overset{4}{—} u_8 \overset{10}{—} u_9$.

The most intuitive way to measure the support of a subgraph in a graph is to count its isomorphisms. Unfortunately, such a metric is not *anti-monotone* since there are cases where a subgraph appears less times than its extension. For instance, in Fig. 1a the single node subgraph DB appears 3 times while its extension DB $\xrightarrow{4}$ IR appears 4 times. Having an anti-monotone support metric is of crucial importance since it allows the development of methods that effectively prune the search space; without an anti-monotone metric exhaustive search is unavoidable [12, 20]. The literature defines several anti-monotone support metrics such as *minimum image based* (*MNI*) [2], *harmful overlap* (*HO*) [12], and *maximum independent sets* (*MIS*) [20]. These metrics differ in the degree of overlap they allow between subgraph isomorphisms, and the complexity of their computation. In this paper, we adopt the *MNI* [2] metric mainly because it: *(a)* is the only metric that can be efficiently computed; the computation of *MIS* and *HO* are NP-complete [12, 20] and *(b)* provides a superset of the results of the alternative metrics; if we are interested in the *MIS* or *HO* metric we may pay their expensive computational cost and exclude the unqualified subgraphs [12]. Formally, the *MNI* metric is defined as follows [2].

Definition 2 *Let $f_1, \ldots, f_m$ be the set of isomorphisms of a subgraph $S(V_S, E_S, L_S)$ in a graph G. Also let $F(v) = \{f_1(v), \ldots, f_m(v)\}$ be the set that contains the (distinct) nodes in G whose functions $f_1, \ldots, f_m$ map a node $v \in V_S$. The* minimum image based support *(MNI) of S in G, denoted by $s_G(S)$, is defined as $s_G(S) = \min\{\, t \mid t = |F(v)|$ for all $v \in V_S\}$.*

For instance, for the subgraph S_1 of Fig. 1b and the graph G of Fig. 1a, we have $F(v_1) = \{u_1, u_5, u_6\}$, $F(v_2) = \{u_3, u_4, u_8\}$ and $F(v_3) = \{u_3, u_4, u_9\}$, thus $s_G(S_1) = 3$. To compare, the respective *MIS* metric is 2 since isomorphisms $u_1 \xrightarrow{4} u_3 \xrightarrow{10} u_4$ and $u_5 \xrightarrow{4} u_4 \xrightarrow{10} u_3$ overlap and the *MIS* metric regards them as one.

The frequent subgraph mining problem is defined as:

Problem 1 *Given a graph G and a minimum support threshold τ, the* frequent subgraph isomorphism mining problem *is defined as finding all subgraphs S in G such that $s_G(S) \geq \tau$.*

Problem 1 does not consider finding the actual number of appearances (i.e., frequency) provided that it is greater than τ. This is very useful in several applications [6, 20], but there are others that demand the exact number of appearances (like graph indexing [31]). Also note, that Problem 1 is computationally expensive since it relies on the NP-hard subgraph isomorphism problem [13].

Definition 1 enforces matching on both node and edge labels. For instance in Fig. 1, subgraph S_2 has only one isomorphism (formed by nodes u_1, u_2 and u_3). Recent research argues that this matching is rather restrictive, and relaxes it by allowing indirect relationships and differences between the edges of the graph and the subgraph [4, 17, 34]. Such frameworks may also consider subgraph $u_6 \xrightarrow{4} u_8 \xrightarrow{20} u_7$ to be a match of S_2 since DM and DB are indirectly connected. We refer to this match as a pattern. For mining frequent patterns, we adopt the pattern matching definition as outlined in [34]. Specifically, we employ a distance metric to measure the distance between two nodes. To this end, we may use any *metric* function, i.e., a function that satisfies the triangle inequality [34]. Typically, the distance function is computed based on the edge labels (or weights) but it may also be defined on other graph properties (e.g., the number of hops between two nodes).

For graph G of Fig. 1, we may use a distance function $\Delta_h(u, v)$ defined as the number of hops in the shortest path that connects u and v. For instance, $\Delta_h(u_0, u_3) = 2$. Alternatively, we may use $\Delta_p(u, v)$ defined as the minimum sum of the inverse of edge weights among the paths that connect u and v. For an example, $\Delta_p(u_6, u_7) = 1/4 + 1/20 = 0.3$. Intuitively, a shorter distance denotes a stronger collaboration. Fig. 2 illustrates the values of Δ_p for the graph G of Fig. 1. Solid lines correspond to the original edges of the graph, while dotted lines illustrate some additional transitions (for figure clarity, we do not show all transitions).

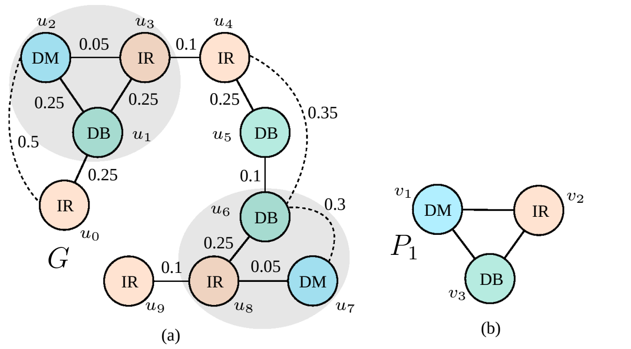


Figure 2: (a) The distance Δ_p for the graph G of Fig. 1. (b) A pattern P_1.

Definition 3 *A graph $P = (V_P, E_P, L_P)$ is a* pattern *of a graph $G(V, E, L)$ iff $V_P \subseteq V$, $L_P(v) = L(v)$ for all $v \in V_P$ and $L_P(e) = \emptyset$ for all $e \in E_P$.*

In other words, a pattern is analogous to a subgraph but without considering edge labels. For instance, a pattern P_1 of the graph G is presented in Fig. 2b.

Definition 4 *Let $P = (V_P, E_P, L_P)$ be a pattern of a graph $G = (V, E, L)$, Δ be a distance metric function, and δ be a user-defined distance threshold. A* pattern embedding *of P to G is an injective function $\phi : V_P \rightarrow V$ satisfying* (a) *$L_P(v) = L(\phi(v))$ for all nodes $v \in V_P$ and* (b) *$\Delta(\phi(u), \phi(v)) \leq \delta$ for all edges $(u, v) \in E_P$.*

The minimum image based support for a pattern, denoted by $\sigma_G(P)$, can be computed as in Definition 2 by replacing the isomorphisms $f_1, \ldots, f_m$ with the pattern embeddings $\phi_1, \ldots, \phi_\mu$. For example consider Fig. 2; setting a threshold $\delta = 0.3$, we have $\sigma_G(P_1) = 2$. The corresponding embeddings are illustrated by the gray areas. Note that there are other possible matches to P_1 but only the indicated two satisfy the constraint $\Delta(\phi(u), \phi(v)) \leq \delta$.

Problem 2 *Given a graph G, a distance function Δ, a distance threshold δ, and a minimum support threshold τ, the* frequent pattern embedding mining problem *is defined as finding all patterns P of G such that $\sigma_G(P) \geq \tau$.*

3. THE GRAMI APPROACH

GRAMI proposed a novel technique that addresses the frequent subgraph mining problem without exhaustively enumerating all isomorphisms in the graph. To this end, GRAMI models the underlying problem as a *constraint satisfaction problem* (Section 3.1). Following, Section 3.2 applies the model to solve the frequent subgraph problem. Section 3.3 proposes several optimizations to enhance performance. The frequent pattern mining problem together with other interesting extensions are discussed in Section 4.

3.1 The CSP Model

A *constraint satisfaction problem (CSP)* is represented as a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where *(a)* $\mathcal{X}$ is an ordered set of variables, *(b)* $\mathcal{D}$ is a set of domains corresponding to variables $\mathcal{X}$, and *(c)* $\mathcal{C}$ is a set of constraints between the variables in $\mathcal{X}$. A *solution* for the CSP is an assignment to the variables in $\mathcal{X}$, such that all constraints in $\mathcal{C}$ are satisfied. The subgraph isomorphism problem (Definition 1) can be mapped to a CSP as follows.

Definition 5 *Let $S(V_S, E_S, L_S)$ be a subgraph of a graph $G(V, E, L)$. The* subgraph S to graph G CSP, *is a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where:*

1. *$\mathcal{X}$ contains a variable x_v for every node $v \in V_S$.*
2. *$\mathcal{D}$ is the set of domains for each variable $x_v \in \mathcal{X}$. Each domain is a subset of V.*
3. *Set $\mathcal{C}$ contains the following constraints:*
 a) $x_v \neq x_{v'}$, for all distinct variables $x_v, x_{v'} \in \mathcal{X}$.
 b) $L(x_v) = L_S(v)$, for every variable $x_v \in \mathcal{X}$.
 c) $L(x_v, x_{v'}) = L_S(v, v')$, for all $x_v, x_{v'} \in \mathcal{X}$ such that $(v, v') \in E_S$.

To simplify notation, whenever it is clear from the context, we use v to refer to a node of the subgraph and to the corresponding variable x_v of the CSP as we do in the following example.

Example 1 *Consider Fig. 1. The subgraph S_1 to graph G CSP is defined as:*

$$\left(\begin{array}{c} (v_1, v_2, v_3), \{\{u_0, \ldots, u_9\}, \ \ldots, \ \{u_0, \ldots, u_9\}\}, \\ \{v_1 \neq v_2 \neq v_3, \ L(v_1) = \text{DB}, \ L(v_2) = L(v_3) = \text{IR}, \\ L(v_1, v_2) = 4, \ L(v_2, v_3) = 10\} \end{array} \right)$$

The following proposition relates the subgraph to a graph CSP with the subgraph isomorphism f (Definition 1).

Proposition 1 *A solution of the subgraph S to graph G CSP corresponds to a subgraph isomorphism of S to G.*

Intuitively, a solution assigns a different node of G to each node of S, such that the labels of the corresponding nodes and edges match. For instance, a solution to the CSP of Example 1 is the assignment $(v_1, v_2, v_3) = (u_1, u_3, u_4)$.

Definition 6 *An assignment of a node u to a variable v is* valid *if and only if there exists a solution that assigns u to v. Note that each valid assignment corresponds to an isomorphism.*

In Example 1, $v_2 = u_3$ is a valid assignment; $v_2 = u_0$ is invalid.

Proposition 2 *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be the subgraph S to graph G CSP. The MNI support of S in G satisfies τ, i.e., $s_G(S) \geq \tau$, iff every variable in $\mathcal{X}$ has at least τ distinct valid assignments (i.e., isomorphisms of S in G).*

Proposition 2 is a key part of this work since it provides a method to determine if a subgraph S is frequent in G. To this end, we may consider the S to G CSP and check the number of valid assignments of every variable. If for every variable there exists τ or more valid assignments, then $s_G(S) \geq \tau$ and S is considered frequent. Continuing Example 1, we have $s_G(S_1) \geq 3$ since all domains contain at least 3 valid assignments (more specifically, the domains of variables v_1, v_2 and v_3 are $\{u_1, u_5, u_6\}$, $\{u_3, u_4, u_8\}$ and $\{u_4, u_3, u_9\}$ respectively).

3.2 Frequent Subgraph Mining

We now apply the CSP model presented in Section 3.1 to solve the frequent subgraph mining problem (Problem 1). We start by presenting Algorithms FREQUENTSUBGRAPHMINING and SUBGRAPHEXTENSION that are used in many related methods to generate candidate subgraphs [29, 20] and are illustrated for completeness. Then, we consider methods to measure the number of appearances (frequency) of these subgraphs. Algorithm ISFREQUENTCSP shows how we may address frequency evaluation without computing and storing all intermediate results. Algorithm ISFREQUENTHEURISTIC offers a heuristic approach and Algorithm ISFREQUENT supplements it with optimizations that highly improve performance. The frequent pattern embedding mining problem (Problem 2) is discussed in Section 4.

Algorithm: FREQUENTSUBGRAPHMINING

```
Input: A graph $G$ and the frequency threshold $\tau$
Output: All subgraphs $S$ of $G$ such that $s_G(S) \geq \tau$
1 $result \leftarrow \emptyset$
2 Let $fEdges$ be the set of all frequent edges of $G$
3 foreach $e \in fEdges$ do
4     $result \leftarrow result \cup$ SUBGRAPHEXTENSION$(e, G, \tau, fEdges)$
5     Remove $e$ from $G$ and $fEdges$
6 return $result$
```

Algorithm: SUBGRAPHEXTENSION

```
Input: A subgraph $S$ of a graph data $G$, the frequency threshold $\tau$ and the set of
       frequent edges $fEdges$ of $G$
Output: All frequent subgraphs of $G$ that extend $S$
1 $result \leftarrow S$, $candidateSet \leftarrow \emptyset$
2 foreach edge $e$ in $fEdges$ and node $u$ of $S$ do
3     if $e$ can be used to extend $u$ then
4         Let $ext$ be the extension of $S$ with $e$
5         if $ext$ is not already generated then
              $candidateSet \leftarrow candidateSet \cup ext$
6 foreach $c \in candidateSet$ do
7     if $s_G(c) \geq \tau$ then
8         $result \leftarrow result \cup$ SUBGRAPHEXTENSION$(c, G, \tau, fEdges)$
9 return $result$
```

FREQUENTSUBGRAPHMINING starts by identifying set *fEdges* that contains all frequent edges (i.e., with support greater or equal to τ) in the graph. Based on the anti-monotone property, only these edges may participate in frequent subgraphs. For each frequent edge, SUBGRAPHEXTENSION is executed. This algorithm takes as input a subgraph S and tries to extend it with the frequent edges of *fEdges* (Lines 2-5). All applicable extensions that have not been previously considered are stored in $candidateSet$. To exclude already generated extensions (Line 5) we adopt the *DFScode* canonical form as in GSPAN [29]. Then, SUBGRAPHEXTENSION (Lines 6-8) eliminates the members of $candidateSet$ that do not satisfy the support threshold τ since according to the anti-monotone property, their extensions are also infrequent. Finally, SUBGRAPHEXTENSION is recursively executed (Line 8) to further extend the frequent subgraphs.

According to Proposition 2, a subgraph S is frequent in G (i.e., $s_G(S) \geq \tau$) if there exist at least τ nodes in each domain $D_1, \ldots, D_n$ that are valid variable assignments (i.e., are part of a solution) for the corresponding variables $v_1, \ldots, v_n$. To evaluate frequency, we may use ISFREQUENTCSP that returns $true$ iff S is a frequent subgraph of G. Initially, ISFREQUENTCSP enforces *node and arc consistency* [22]. Node consistency excludes unqualified nodes from the domains (like nodes with different labels or with lower degree) and arc consistency ensures the consistency between the assignments of two variables. Specifically, for every constraint $C(v, v')$, arc consistency ensures that for every node in the domain of v there exists a node in the domain of v' satisfying $C(v, v')$. If, after node and arc consistency enforcement, the size of a domain is smaller than τ the algorithm returns $false$ (Line 3). Following, ISFREQUENTCSP considers every solution $\mathcal{S}ol$ and marks the nodes assigned to variables to the corresponding domains (Line 5). If all domains have at least τ marked nodes then (according to

Algorithm: ISFREQUENTCSP

```
Input: Graphs $S$ and $G$ and the frequency threshold $\tau$
Output: $true$ if $S$ is a frequent subgraph of $G$, $false$ otherwise
1 Consider the subgraph $S$ to graph $G$ CSP
2 Apply node and arc consistency
3 if the size of any domain is less than $\tau$ then return $false$
4 foreach solution $\mathcal{S}ol$ of the $S$ to graph $G$ CSP do
5     Mark all nodes of $\mathcal{S}ol$ in the corresponding domains
6     if all domains have at least $\tau$ marked nodes then return $true$
7 return $false$ // Domain is exhausted
```

Algorithm: IsFrequentHeuristic

```
Input: Graphs $S$ and $G$ and the frequency threshold $\tau$
Output: $true$ if $S$ is a frequent subgraph of $G$, $false$ otherwise
1  Consider the subgraph $S$ to graph $G$ CSP
2  Apply node and arc consistency
3  foreach variable $v$ with domain $D$ do
4      $count \leftarrow 0$
5      Apply arc consistency
6      if the size of any domain is less than $\tau$ then return $false$
7      foreach element $u$ of $D$ do
8          if $u$ is already marked then $count$++
9          else if a solution $\mathcal{S}ol$ that assigns $u$ to $v$ exists then
10             Mark all values of $\mathcal{S}ol$ in the corresponding domains
11             $count$++
12         else Remove $u$ from the domain $D$
13         if $count = \tau$ then Move to the next $v$ variable (Line 3)
14     return $false$ // Domain is exhausted and $count < \tau$
15 return $true$
```

Proposition 2) S is frequent in G. Otherwise, IsFrequentCsp continues with the following solution.

Complexity. Let N and n be the number of nodes of graph G and subgraph S respectively. The complexity of FrequentSubgraphMining is determined by the complexity of SubgraphExtension and IsFrequentCsp. The former computes all subgraphs of G, which takes $\mathcal{O}(2^{N^2})$ time. The latter evaluates frequency which is reduced to the computation of subgraph isomorphisms (a well-known *NP*-hard problem) and takes $\mathcal{O}(N^n)$ time. Overall, the complexity of the mining process is $\mathcal{O}(2^{N^2} \cdot N^n)$ time which is exponential in the problem size. Thus, it is of crucial importance to devise appropriate heuristics and optimizations that improve execution performance. Several works study the subgraph generation process and propose techniques that significantly improve performance [29, 20]. These techniques are implemented in Algorithm SubgraphExtension. In the following section, we consider the optimization of Algorithm IsFrequentCsp that computes subgraph isomorphisms.

3.3 Optimizing Frequency Evaluation

Algorithm IsFrequentCsp naively iterates over the solutions of the subgraph S to graph G CSP trying to find τ valid assignments for every variable. To guide this search process, we propose the heuristic illustrated in Algorithm IsFrequentHeuristic. Intuitively, the algorithm considers each variable at a time and searches for τ valid assignments. If these are found, it moves to the next variable and repeats the process. In more details, IsFrequentHeuristic starts by enforcing node and arc consistency. Then, the algorithm considers every variable and counts the valid assignments in its domain (stored in variable *count*). If, during the process, any variable domain remains with less than τ candidates, then the subgraph cannot be frequent, so the algorithm returns *false* (Line 6 and 14). To count the valid assignments, IsFrequentHeuristic iterates over all nodes u in the domain D of a variable x and searches for a solution that assigns u to x. If the search is successful then *count* is incremented by 1, and the process continues to the next node in D until the number of valid assignments (*count*) becomes τ, in which case the algorithm proceeds to the next domain (Line 13). On the other hand, if search is unsuccessful then u is removed from D and the algorithm continues with the next node in D. Updating D may trigger new inconsistencies in other domains, thus, arc consistency (Line 5) is checked again. IsFrequentHeuristic also implements the following optimization. Assume that for a domain D a solution was found for some node $u \in D$. Then, *count* is incremented by 1 and all nodes (including u) that belong to this solution are *marked* in the respective

Algorithm: IsFrequent

```
Input: Graphs $S$ and $G$ and the frequency threshold $\tau$
Output: $true$ if $S$ is a frequent subgraph of $G$, $false$ otherwise
1  Consider the subgraph $S$ to graph $G$ CSP and apply node and arc consistency
   // Push-down pruning
2  foreach edge $e$ of $S$ do
3      Let $S^{/e}$ be the graph after removing $e$ from $S$
4      Remove the values of the domains in $S$ that correspond to invalid
       assignments of $S^{/e}$
   // Unique labels
5  if $S$ and $G$ satisfy the unique labels optim. conditions then
6      if the size of any domain is less than $\tau$ then return $false$
7      else return $true$
   // Automorphisms
8  Compute the automorphisms of $S$
9  foreach variable $x$ and its domain $D$ do
10     $count \leftarrow 0$, $timedoutSearch \leftarrow \emptyset$
11     if there is an automorphism with a computed domain $D'$ then
12         $D \leftarrow D'$ and move to the next $x$ variable (Line 9)
13     Apply arc consistency
14     if the size of a domain is less than $\tau$ then return $false$
       // Lazy search
15     foreach element $u$ of $D$ do
16         if $u$ is already marked then $count$++
17         else
18             Search for a solution that assigns $u$ to $x$ for a given time
               threshold
19             if search timeouts then Save the search state in a structure
               $timedoutSearch$
20             if a solution $\mathcal{S}ol$ is found then
21                 Mark all values of $\mathcal{S}ol$ to the corresponding domains
22                 $count$++
23             else Remove $u$ from the domain $D$ and add $u$ to the invalid
               assignments of $D$ in $S$
24             if $count = \tau$ then Move to the next variable (Line 9)
       // Resume timed-out search if needed
25     if $|timedoutSearch| + count \geq \tau$ then
           // Decompose
26         Decompose graph $S$ into a set of graphs $\mathbf{Set}$ that contain the newly
           added edge
27         foreach $s \in \mathbf{Set}$ do Remove invalid assignments of $s$ from the
           respective domains of $S$
28         foreach $t \in timedoutSearch$ do
29             Resume search from the saved state $t$
30             if a solution $\mathcal{S}ol$ is found then
31                 Mark all values of $\mathcal{S}ol$ to the corresponding domains
32                 $count$++
33             else Remove $u$ from the domain $D$ and add $u$ to the invalid
               assignments of $D$ in $S$
34             if $count = \tau$ then Move to the next variable (Line 9)
35     return $false$ // Domain is exhausted and $count < \tau$
36 return $true$
```

domains (Line 10). Hence, if these nodes are considered in a later iteration of the algorithm, they are recognized as already belonging to a solution (Line 8). This precludes any further search.

In the following, we introduce Algorithm IsFrequent that enhances IsFrequentHeuristic through several optimizations that significantly improve execution performance. IsFrequent uses three novel optimizations, namely, *Push-down pruning*, *Lazy search* and *Unique labels*. Finally, IsFrequent specializes, for frequent mining, *Decomposition pruning* and *Automorphisms*, that are known to speed-up search [8] and frequent subgraph mining [1] respectively. In the sequel, we present the optimization techniques according to their execution order in the IsFrequent algorithm.

Push-down pruning. The subgraph generation tree is constructed by extending a parent subgraph with one edge at a time. Since the parent is a substructure of its children, those assignments that were pruned from the domains of the parent, cannot be valid assignments for any of its children. For example, Fig. 3a illustrates a part of a subgraph generation tree consisting of subgraph S_1 which is extended to S_2, S_3 and then to S_4 (via S_2). Assume that when considering subgraph S_1, ISFREQUENT excludes elements a_3, b_1, and a_3 from the domain of variables v_1, v_2, and v_3 respectively (depicted by light gray ovals in Fig. 3b). This information can be pushed down such that a_3, b_1, a_3 are also pruned from all descendants of S_1. This happens recursively; for instance, the assignments pruned because of S_2 are depicted by dark gray dotted ovals.

The same substructure may also appear in subgraphs that do not have an ancestor/descendant relationship. In the example of Fig. 3, S_4 is not a descendant of S_3; however, both contain substructure $A-B-A-C$. Since S_3 and S_4 are in different branches, pushing down the pruned assignments is not applicable. Instead, we use a hash table to store the pruned assignments of previously checked subgraphs. The hash key is the *DFScode* canonical representation of S_3 [29]. When S_4 is generated, the hash table is searched for matching substructures. If one is found, the corresponding invalid assignments are pruned from the domains of S_4. ISFREQUENT applies this optimization (Lines 2-4) using the invalid assignments populated while searching for valid nodes (Lines 23 and 33).

Saving the invalid assignments of subgraphs results in a significant performance gain for the following two reasons.

- Subgraphs (like S_4) take advantage of the respective pruning of smaller subgraphs (like S_1 and S_2) to prune invalid assignments. Thus, the domains of the subgraph variables are reduced avoiding the expensive search procedure (Lines 18 and 29). In many cases, a subgraph may be eliminated without search. For instance, in Fig. 3, assuming that $\tau = 3$, S_4 can be eliminated, because there are only two valid assignments of variable v_1 remaining in its domain.
- This domain reduction also speeds up the search process since it highly depends on the domain size. For instance, in Fig. 3, assuming that $\tau = 2$, when considering variable v_1, the search space has a size of $2{\cdot}2{\cdot}3{\cdot}4 = 48$ combinations (bottom of Fig. 3b), while without using this optimization the respective search space size is $5{\cdot}3{\cdot}5{\cdot}6 = 450$ combinations.

To perform push-down pruning, Line 3 constructs $\mathcal{O}(n^2)$ subgraphs $S^{/e}$ by removing an edge from S, (n is the number of nodes in S) and uses a hash lookup to remove the invalid assignment (Line 4). Thus, the overall complexity is $\mathcal{O}(n^2)$ time.

Unique labels. In the case of data graphs with a single label per node and subgraphs having a tree-like structure and unique node labels, the following optimization can be applied:

Proposition 3 *Let G be a graph with a single label per node, $S(V_S, E_S, L_S)$ be a subgraph of G, S's underlying undirected graph is a tree, and all of its node labels are unique, i.e., $L_S(v) \neq L_S(v')$ for all v and v' in V_S such that $v \neq v'$. To calculate $s_G(S)$ directly, it suffices to consider the S to G CSP and refine the domains of variables by enforcing node and arc consistency.*

PROOF: Since each graph node has a single label and the query has unique labels, no node can appear in more than one domain. For any S, we will use induction to prove that each value N in each domain of S (after applying the node and arc consistency constraints) is part of a valid solution. Let Q be a copy of S where all of S's directed edges are replaced with undirected ones. Q is connected, undirected, and acyclic, therefore it is a tree. Let Q be rooted at the node corresponding to N's domain.

- For Q with $height = 1$, N is guaranteed to be part of a valid solution (by definition of the node and arc consistency constraints and by considering the fact that the same node cannot appear in other domains).

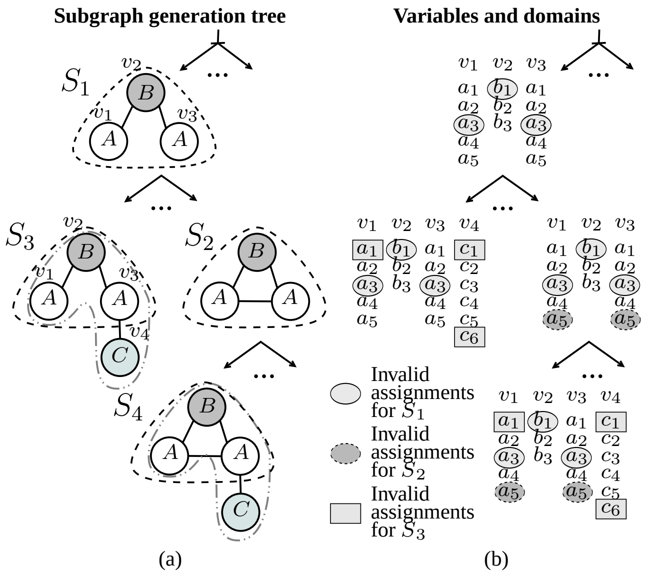

Subgraph generation tree
Variables and domains
S_1
S_2
S_3
S_4
Invalid assignments for S_1
Invalid assignments for S_2
Invalid assignments for S_3
(a)
(b)

Figure 3: (a) Construction of the subgraph tree. (b) Variables and domains of the corresponding subtrees. Marked nodes represent the pruned assignments which are pushed down the tree.

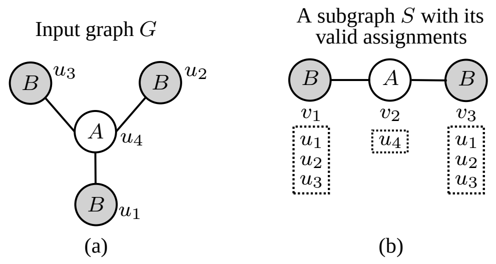


Figure 4: Automorphisms. (a) Input graph G. (b) Subgraph S and its valid assignments.

- For Q with $height = R$, let T be a subgraph of S and its underlying undirected graph is a subtree of Q sharing the same root but with $height = R - 1$. Let L be the set of T's leaf nodes and assume that T has a solution. Q is composed of T and the set of trees Z with height 1 (or 0) each rooted at a distinct node from L. Since each element in Z has a solution in G, and each solution joins with T's solution only by its corresponding root in Z, hence, a valid solution for S exists.

Note that the final step cannot be applied when the underlying undirected graph Q contains a cycle. For example if S is an undirected triangle of 3 nodes labeled (A, B, C) and the data graph G is undirected and contains 6 nodes forming a cycle: (A, B, C, A, B, C). When considering the S to G CSP after enforcing node and arc consistency the count $s_G(S)$ is 2, but, the correct result is 0. □

Example 2 *Consider the subgraph* DB$-$IR *and the graph G of Fig. 1. Let v_1 (resp. v_2) be the variable that corresponds to nodes labeled with* DB *(resp.* IR*). The initial domains are $D_{v_1} = D_{v_2} = \{u_0, \ldots, u_9\}$. After applying node and arc consistency we have $D_{v_1} = \{u_1, u_5, u_6\}$ and $D_{v_2} = \{u_0, u_3, u_4, u_8\}$ which encodes the actual isomorphisms of the subgraph to graph G.*

If the conditions hold (Line 5), GRAMI uses the current domain sizes to directly decide whether S is frequent or not (Lines 6-7). The overall process can be performed in $\mathcal{O}(n)$ time.

Automorphisms. Automorphism is an isomorphism of a graph to itself. Automorphisms appear because of symmetries. Following

[1], such symmetries in the subgraph can be used to prune equivalent branches and reduce the search space. For example, consider subgraph S of graph G presented in Fig. 4; S has automorphisms. To determine if S is frequent in G, while iterating over the domain of v_1, IsFrequent finds the assignment $(v_1, v_2, v_3) = (u_1, u_4, u_2)$ to be a solution (i.e., an isomorphism of S to G). Due to the symmetry of the subgraph S, assignment $(v_1, v_2, v_3) = (u_2, u_4, u_1)$ is also a solution. The benefits of this observation are twofold. First, we may identify the valid assignments of a variable more efficiently. More importantly, when we compute all valid assignments of a variable (like v_1) we also compute the valid assignments for its symmetric counterpart (i.e., v_3).

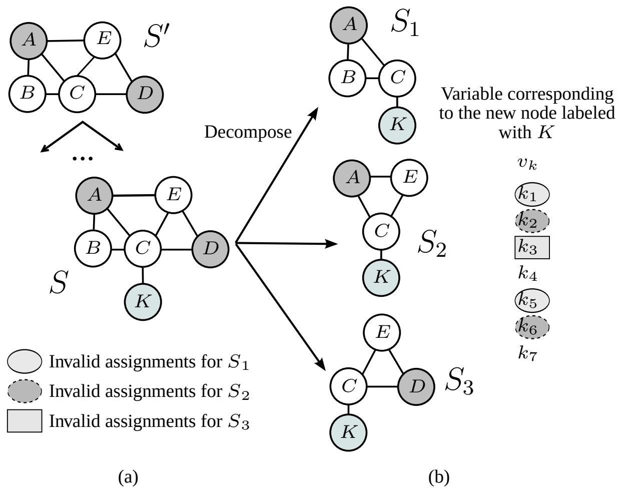
IsFrequent detects automorphisms in Line 8. This requires $\mathcal{O}(n^n)$ time where n is the number of nodes in subgraph S. In practice, despite the exponential worst-case bound, the cost of automorphisms is very low since the size of subgraph S is negligible compared to the size of the graph G.

Lazy search. Intuitively, to prove that a partial assignment does not contribute to any valid solution, the search algorithm has to exhaust all available options; a rather time consuming process. Thus, if a search for a solution that pertains to a specific partial assignment takes a long time, then this is probably because the partial assignment cannot contribute to a complete valid assignment. To address such cases, initially IsFrequent searches for a solution only for a limited time threshold (Line 18). The intuition of the optimization is that other assignments may produce much faster results that will help indicate if the subgraph is frequent ($s_G(S) \geq \tau$). In such a case, the result of the timed out search would be irrelevant, hence, there is no reason to waste time in further search. Nevertheless, this cannot guarantee that a timed out partial assignment will not eventually be essential for proving the frequency of the subgraph. Thus, if search is timed out, the algorithm stores the search state in the $timedoutSearch$ set of nodes with incomplete check. These searches will only be resumed when the non-timed out cases are not sufficient to show that a subgraph is frequent. More specifically, timed-out searches are considered if after the time limited search, $count < \tau$ and $count$ plus the size of $timedoutSearch$ (i.e., the number of timed out searches) surpasses the threshold τ (Line 25). Only then, the algorithm resumes each timed out search $t \in timedoutSearch$ from its saved state but without a time-out option until enough assignments are found to prove frequency (Line 34). Note that, if necessary, IsFrequent eventually searches the entire search space for each variable to provide the exact solution.

The complexity of Lazy search (Lines 15-24) can be done in $\mathcal{O}(N)$ time (note that the search of Line 18 takes constant time since it is performed for a specific time frame).

Decomposition pruning. The final optimization is performed in Lines 26 and 27. At this point, the algorithm is about to resume the timed out searches. To reduce the problem size, the algorithm decomposes the input subgraph S into a set of distinct subgraphs $\mathbf{Set}$. Recall that algorithm SubgraphExtension extends subgraphs by adding an edge e from the set of frequent edges *fEdges*. Set $\mathbf{Set}$ is constructed by removing one edge at a time from S and adding to $\mathbf{Set}$ the connected component that includes edge e. Any other decomposition has already been considered by the *Push-down pruning* optimization. Finding and removing invalid assignments from the domains of the elements of $\mathbf{Set}$ is a much easier task because they are smaller than the original subgraph S.

For example, consider Fig. 5. Subgraph S extends S' with edge $C{-}K$ and, thus, it is decomposed into $\mathbf{Set}$ that contains subgraphs S_1 to S_3. Let us assume that the variable corresponding to the new node labeled with K is v_k and the initial domain of v_k contains



S'
S_1
S_2
S_3
S
Decompose
Variable corresponding to the new node labeled with K
v_k
k_1
k_2
k_3
k_4
k_5
k_6
k_7
Invalid assignments for S_1
Invalid assignments for S_2
Invalid assignments for S_3
(a)
(b)

Figure 5: (a) Subgraph S is generated by extending S' with edge $C-K$. (b) S is decomposed into overlapping subgraphs S_1 to S_3 containing the newly extended edge $C-K$.

values k_1 to k_7. Further, assume that using subgraphs S_1, S_2 and S_3 we can exclude values $\{k_1, k_5\}$, $\{k_2, k_6\}$ and $\{k_3\}$ respectively. The decomposition optimization removes all these values from the domain of v_k, therefore, it only contains the values k_4 and k_7.

Decomposition pruning can be done in $\mathcal{O}(n^2)$. Resuming timed-out searches (Lines 28-34) requires solving a CSP on $n-1$ variables with domain of size N and can be done in $\mathcal{O}(N^{n-1})$ time.

Complexity analysis of IsFrequent**.** Let N and n be the number of nodes in G and S respectively. Push-down pruning, unique labels and automorphisms can be done in $\mathcal{O}(n^2)$, $\mathcal{O}(n)$ and $\mathcal{O}(n^n)$ respectively. Subgraph size is negligible in comparison to the data graph size, and thus these procedures are not expensive. IsFrequent applies arc consistency, lazy search and resumes timed-out search that can be done in $\mathcal{O}(Nn)$, $\mathcal{O}(N)$ and $\mathcal{O}(N^{n-1})$ respectively. Thus, the complexity of IsFrequent is determined by the resumed timed-out searches. More specifically, if p is the possibility expressing that a node in a domain of a variable is valid, then to find the required τ valid assignments we need to consider τ/p nodes and solve τ/p CSPs of size $n-1$ for each one of the n variables. In total, the complexity bound is $\mathcal{O}(n \cdot \tau/p \cdot N^{n-1})$.

4. GraMi EXTENSIONS

Generalization to pattern mining. Section 3 models the subgraph isomorphism problem (Definition 1) as a subgraph to graph CSP (Definition 5). Similarly, a pattern embedding ϕ (Definition 4) can be mapped to a CSP by replacing Condition 3c of Definition 5 as follows.

3c) $\Delta(x_v, x_{v'}) \leq \delta$, *for every* $x_v, x_{v'} \in \mathcal{X}$ *such that* $(v, v') \in E_P$ *(where* Δ *is the distance metric and* δ *is the distance threshold).*

Whenever it is clear from the context, we use v to refer to a node of the pattern and x_v to refer to the corresponding variable of the CSP as we do in the following example.

Example 3 *Consider Fig. 2. For* $\delta = 0.3$*, the pattern* P_1 *of graph G CSP is defined as:*

$$\left(\begin{array}{c} (v_1, v_2, v_3),\ \{\{u_0, \ldots, u_9\},\ \ldots,\ \{u_0, \ldots, u_9\}\}, \\ \{\ v_1 \neq v_2 \neq v_3,\ L(v_1){=}\text{DM},\ L(v_2){=}\text{IR},\ L(v_3){=}\text{DB}, \\ \Delta(v_1, v_2) \leq 0.3,\ \Delta(v_2, v_3) \leq 0.3,\ \Delta(v_1, v_3) \leq 0.3\ \} \end{array} \right)$$

The notations for a solution (Proposition 1) and valid (or invalid) assignments (Definition 6) are easily extended to support pattern to

Table 1: Definitions of the anti-monotonic structural constraints for pattern P, implemented in CGRAMI

$\|V_P\| \leq \alpha$	Number of nodes should not exceed α
$\|E_P\| \leq \alpha$	Number of edges should not exceed α
$\max(degree(V_P)) \leq \alpha$	The maximum node degree is α

Table 2: Definitions of the anti-monotonic semantic constraints for pattern P, implemented in CGRAMI

$(\forall v \in V_P)(L(v) \in \mathcal{L})$	P contains only labels from $\mathcal{L}$
$(\forall v \in V_P)(L(v) \notin \mathcal{L})$	P does not contain any label from $\mathcal{L}$
$(\forall v, v' \in E_P)(L(v, v') \in \mathcal{E})$	P contains only edges from $\mathcal{E}$
$(\forall v, v' \in E_P)(L(v, v') \notin \mathcal{E})$	P does not contain any edges from $\mathcal{E}$
$(\neg subgraph(P', P))$	Pattern P must not contain a specific subgraph P'
$(\forall v \in V_P)(\text{count}(L(v)) \leq \alpha)$	A node label cannot appear more than α times in P

graph CSPs. For instance, assignment $(v_1, v_2, v_3) = (u_7, u_8, u_6)$ is a solution of the CSP of Example 3 and a pattern embedding of P_1 to G. Moreover, $v_2 = u_3$ is a valid assignment while $v_2 = u_0$ is invalid (and thus, cannot be extended to a solution).

Proposition 4 *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be the pattern P to graph G CSP. The MNI support of P in G satisfies τ, i.e., $\sigma_G(S) \geq \tau$, iff every variable in $\mathcal{X}$ has at least τ distinct valid assignments (i.e., embeddings of P in G).*

Continuing Example 3, we have $\sigma_G(P_1) \geq 2$ since all domains contain at least 2 valid assignments (the domains of variables v_1, v_2 and v_3 are $\{u_2, u_7\}$, $\{u_3, u_8\}$ and $\{u_1, u_6\}$ respectively).

To address the frequent pattern mining problem (Problem 2), we can also employ Algorithms ISFREQUENTHEURISTIC and ISFREQUENT, with the following additional preprocessing step. For each frequent node, we precompute the set of nodes that are reachable within distance δ. We run a distance-bound Dijkstra algorithm from each frequent node to find the shortest path to the reachable nodes, where the path distance is defined by the distance function Δ; the algorithm terminates when the distance of the shortest path exceeds δ. All optimizations of Section 3.3 apply directly in this setting as well. To avoid confusion, we use GRAMI for the subgraph mining problem and GRAMI(δ) for the pattern mining problem.

User-defined constraints. Typically, frequent patterns show interactions between nodes bearing the same label. For instance, in citation graphs, most collaborations are among authors working in the same field. In many applications, interactions among nodes of different types (like interdisciplinary collaborations) are more interesting and important [33]. To allow the user to focus on the interesting patterns, we developed CGRAMI, a version of GRAMI that supports two types of user-defined constraints: *(a) Structural*, such as "the number of vertices in pattern P should be at most α' and *(b) Semantic*, such as "P must not contain specific labels".

Although not a requirement, it is desirable that the user-defined constraints are anti-monotonic. In such cases, the constraints can be pushed down in the subgraph extension search tree to early prune large parts of the search space, thus accelerating the process. Tables 1 and 2 present a set of useful structural and semantic anti-monotonic constraints that are supported by CGRAMI.

Approximate mining. Frequent subgraph mining is a computationally intensive task since it is dominated by the NP-hard subgraph isomorphism problem. Thus, its performance is prohibitively expensive when applied to large graphs. Motivated by this, we introduce AGRAMI, an approximate version of our framework, which is able to scale to larger graphs. To maintain the quality of results, AGRAMI does not return any infrequent pattern (i.e., does not have false positives), although it may miss some frequent ones (i.e., may have false negatives). To achieve this, we modified the way ISFREQUENT handles time-outs (Line 18) as follows: we set the time-out to occur after $f(\alpha)$ iterations of the search. If a solution is found before this time-out, the $count$ is updated as normal. On the other hand, if a time-out occurs it is assumed that the search was unsuccessful. If enough time-outs occur during the search of a specific domain such that its $count$ remains less than τ, the pattern is considered to be infrequent. Parameter $f(\alpha) = \alpha^n \prod_1^n |D_i| + \beta$, where β is a constant, D_i are the domains of the variables, n is the number of variables and $0 < \alpha \leq 1$ is a user-defined approximation parameter. $\prod_1^n |D_i|$ grows exponentially; thus it has to be bounded by an exponential weight α^n. Increasing α decreases the approximation error at the expense of longer execution time. When $\alpha = 1$, AGRAMI becomes equivalent to GRAMI.

Table 3: Datasets and their characteristics

Dataset	Nodes	Distinct node labels	Edges	Density
Twitter	11,316,811	100	85,331,846	Dense
Patents	3,942,797	453	16,522,438	Medium
Aviation	101,185	6,173	133,087	Sparse
MiCo	100,000	29	1,080,298	Dense
CiteSeer	3,312	6	4,732	Medium

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate GRAMI and its extensions. For comparison, we have implemented GROWSTORE that follows a pattern *grow-and-store* approach [20, 29]. GROWSTORE uses the original code of GSPAN [29] and takes advantage of all its optimizations. The only difference is that GROWSTORE, similarly to GRAMI, use the efficient *MNI* metric. Both GROWSTORE and GRAMI are completely memory based. All experiments are conducted using Java JRE v1.6.0 on a Linux (Ubuntu 12) machine with 8 cores running at 2.67GHz with 192GB RAM and 1TB disk. Our experimental machine used an exotic memory size to accommodate the memory requirements of GROWSTORE; GRAMI may also run on ordinary machines with 4GB RAM for all datasets but Twitter.

Datasets. We experiment on several different workload settings by employing the following real graph datasets; their main characteristics are summarized in Table 3.

Twitter (`socialcomputing.asu.edu/datasets/Twitter`). This graph models the social news of Twitter and consists of ∼11M nodes and ∼85M edges. Each node represents a Twitter user and each edge represents an interaction between two users. The original graph does not have labels, so we randomly added labels to the nodes. The number of distinct labels was set to 100 and the randomization follows a Gaussian distribution.

Patents. This dataset models U.S. patents' citations and consists of a directed graph with ∼4M nodes and ∼16M edges. Each node represents a patent and each edge represents a citation. The graph is maintained by the National Bureau of Economic Research [32]. As a preprocessing step, we remove all unlabeled nodes.

MiCo. This dataset models the Microsoft co-authorship information and consists of an undirected graph with 100K nodes and ∼1M edges. Nodes represent authors and are labeled with the author's field of interest. Edges represent collaboration between two authors and are labeled with the number of co-authored papers. To populate MiCo we crawled the computer science collaboration graph from `academic.research.microsoft.com`.

CiteSeer (`cs.umd.edu/projects/linqs/projects/lbc`). CiteSeer represents a directed graph consisting of ∼3K publications (nodes) and ∼4K citations between them (edges). Each node has a single label representing a Computer Science area. Each edge has a label (0 to 100) that measures the similarity between the corresponding

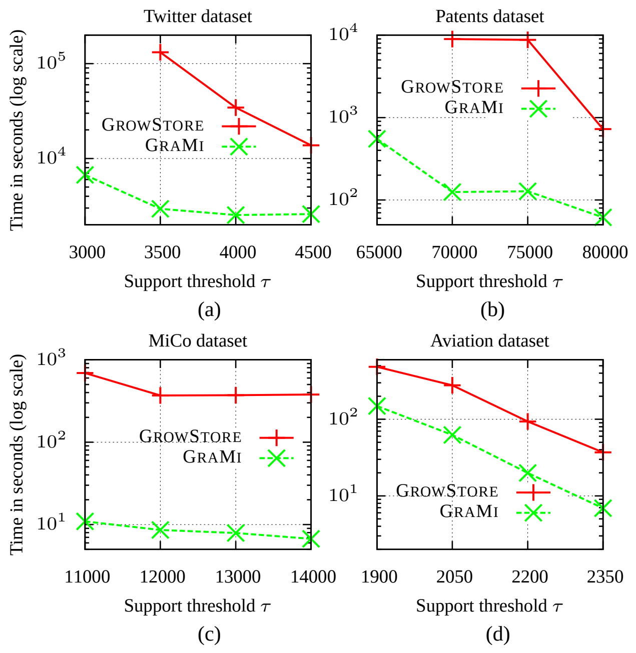


Figure 6: Performance of GraMi and GrowStore

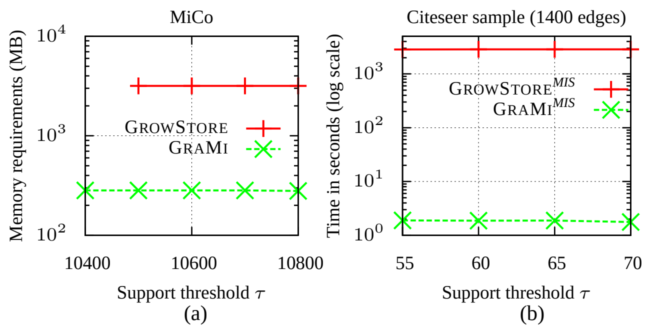


Figure 7: (a) Memory requirements for GraMi and GrowStore and (b) Using *MIS* metric

pair of publications, a smaller label denotes a stronger similarity.

Aviation (`ailab.wsu.edu/subdue`). This dataset contains a list of records extracted from the aviation safety database and was used in [7, 20] for evaluation. Each record corresponds to an event and has several attributes (like event type, location, flight condition). This information is represented by a graph having two types of nodes and edges. The first type of nodes represents the events (and are labeled with the ids of the event) while the second represents attribute values (and are labeled with the actual value). The first type of edges links events and is labeled with their relationship (e.g., near to) while the second type links events with attribute values and is labeled with the attribute name. Aviation consists of 100K nodes and 133K edges. Note that Aviation is a fundamentally different dataset when compared with the previous ones. The Aviation graph has on average one edge per node, thus, it is very sparse. Also it has a very large number of distinct node labels.

Metrics. The support threshold τ is the key evaluation metric as it determines when a subgraph or a pattern is frequent. Decreasing τ results in an exponential increase in the number of possible candidates and thus exponential decrease in the performance of the mining algorithms. For a given time budget, an efficient algorithm should be able to solve mining problems for low τ values. When τ is given, efficiency is determined by the execution time.

To evaluate a result set, we consider the number and the maximum size of subgraphs/patterns in the set. Obviously, these values should be as large as possible.

Computing frequent subgraphs. Initially, we consider Problem 1 that mines frequent subgraph isomorphisms. Fig. 6 shows the performance of GrowStore and GraMi on Twitter, Patents, MiCo and Aviation datasets. The number of results (intermediate and actual) grows exponentially when the support threshold τ decreases. Thus, the running time of all algorithms also grows exponentially. Unlike GrowStore, GraMi does not need to enumerate all intermediate results, thus, it is more efficient. Our results indicate that GraMi outperforms GrowStore by at least two orders of magnitude for Patents and MiCo datasets and by at least an order of magnitude for Twitter and Aviation datasets. For the larger datasets (Twitter and Patents) and for the lower τ (3K and 65K respectively), GrowStore was not able to produce results even when it was alloted 2 orders of magnitude more time than GraMi.

Memory requirements. Fig.7a illustrates the memory requirements for GrowStore and GraMi for the MiCo dataset. Since GrowStore needs to store all intermediate results, it consumes about an order of magnitude more memory. For τ=10, 400 the size of the intermediate results exceed the available memory (192GB), and hence GrowStore crashes. For this frequency, there is an increase in the number of the frequent subgraphs and thus an exponential increase in the number of intermediate candidates that need to be stored and checked for frequency. This trend also appears for the other datasets. GraMi on the other hand is not affected by the increase in the output size. Most of the memory GraMi uses, is required for the storage of the input graph G. The most costly data structure of IsFrequent is the hash table used by push-down pruning, but, still it does not exceed 2% for the overall required memory. Also the space needed to store timed-out searches (set $timedoutSearch$) was never above 1% of the total memory. For all our experiments, GraMi could be also executed in machines with the typical memory size of 4GB except for the Twitter dataset.

Using *MIS* metric. In this experiment, we compare GrowStore^{MIS} the original version of GrowStore that uses the *MIS* metric with GraMi^{MIS}, the modified version of GraMi that also supports *MIS*. For the Aviation dataset, GraMi^{MIS} takes slightly more time than GraMi while GrowStore^{MIS} could not produce results even if it was alloted *three* orders of magnitude more time than GraMi^{MIS}. Interestingly, GrowStore^{MIS} cannot produce results in reasonable time even for the much smaller Citeseer dataset. To achieve a comparison, we have constructed a new dataset by randomly sampling 1400 edges from the Citeseer dataset. The results are illustrated in Fig. 7b. Clearly, GraMi^{MIS} outperforms GrowStore^{MIS} by up to 3 orders of magnitude.

Computing frequent patterns. We now consider Problem 2 that mines frequent pattern embeddings. We evaluate the performance of GrowStore and GraMi(δ) for several values of the distance threshold δ. We use the CiteSeer dataset and distance function $\Delta_h(u, v)$ defined as the number of hops in the shortest path that connects u and v. For GraMi(δ), we test on two different distance thresholds namely 1 and 4. Intuitively, for $\delta = 1$ (respectively $\delta = 4$) two pattern nodes that are connected with an edge may be matched with two graph nodes that are one hop (respectively four hops) away. GrowStore can only find matches that are only one hop away. Thus, only GrowStore and GraMi(1) are directly comparable since they both compute the same results. As shown in Fig. 8a, GraMi(1) is an order of magnitude faster than GrowStore (note the logarithmic scale). As expected GraMi(4) com-

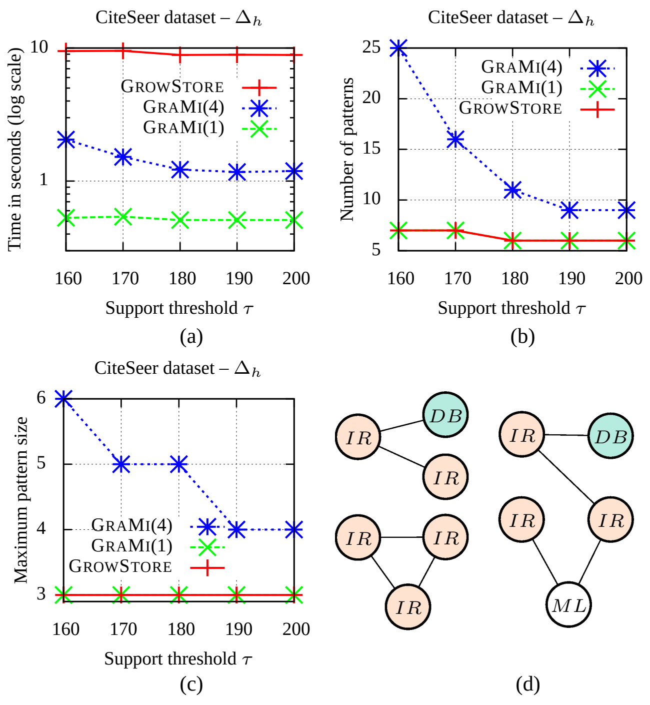


Figure 8: Performance evaluation for mining frequent patterns in CiteSeer dataset comparing between GrowStore **and** GraMi**(δ) where δ is the distance threshold**

putes more and larger patterns than GrowStore and GraMi(1) (Figs. 8b and 8c). An example of a frequent pattern discovered by GraMi is illustrated on the right of Fig. 8d and contains 5 nodes involving 3 different Computer Science areas. To compare, GrowStore computes the 3 nodes patterns at the left of Fig. 8d that involve 1 and 2 areas. To compute these results, GraMi(4) takes more time than GraMi(1) but is still faster than GrowStore.

To further illustrate the benefits of GraMi(δ) we have conducted another set of experiments (Fig. 9). The aim of the experiments is to illustrate the properties of the patterns that can be generated within a specific time budget. Figs. 9a,b, consider the Citeseer dataset with the distance function Δ_h and compare between GrowStore, GraMi(1) and GraMi(4). Specifically, Fig. 9a shows the minimum support threshold τ that can be achieved, when the above algorithms are allotted a time budget that ranges from 1 to 5 seconds (lower is better). For this budget range, Fig. 9b illustrates the number of result patterns (higher is better). In both cases, GraMi(1) and GraMi(4) accomplish lower thresholds and result in more patterns than GrowStore.

CGraMi: User-defined constraints. CGraMi supports the addition of constraints on the returned results (Section 4). Using these constraints, the focus can be on more interesting pattern types like the ones that show interactions between nodes of a different type. To evaluate CGraMi, we use the experimental setting of Fig. 9a,b. The only difference is that we now use CGraMi(δ) with a constraint that does not allow more than 4 nodes with the same label in a pattern. The corresponding results are illustrated in Fig. 9c,d and are directly comparable to Fig. 9a,b. In every case and within the same time budget allowed for both GraMi and CGraMi, CGraMi results in a significantly lower minimum support threshold τ and significantly larger frequent patterns set. For instance, for the Citeseer dataset with a time budget of 3 seconds, CGraMi(1) achieves a 3 times lower threshold and almost 3 times more patterns

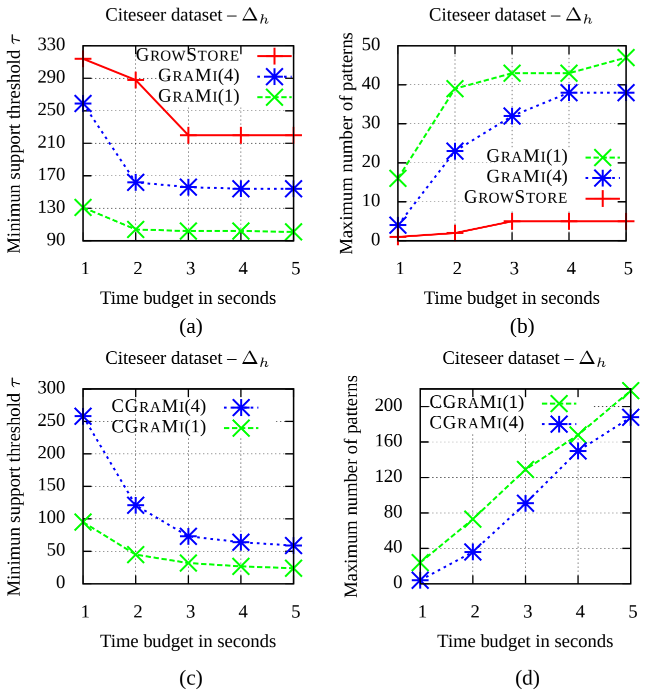


Figure 9: Comparing (a,c) the minimum support threshold and (b,d) the maximum number of frequent patterns that can be achieved within an allotted time budget. For (a,b) we used GraMi(δ) **and for (c,d) we use** CGraMi(δ) **constrained to reject patterns with more than 4 nodes with the same label**

than GraMi. Additionally, CGraMi generates patterns having about 3 times more label interactions than GraMi.

AGraMi: Approximate mining. AGraMi, which offers approximate subgraph and pattern mining (Section 4), can be tuned by the approximation parameter α, $0 < \alpha \leq 1$ (value 1 means no approximation). Fig. 10 illustrates the performance of GraMi and AGraMi for several values for the α parameter in the Patents and MiCo datasets. We evaluate two parameters, execution time and recall, i.e., the percentage of subgraphs returned by AGraMi with respect to the actual complete set of frequent subgraphs. For the Patents dataset, the performance gain is significant, nearly an order of magnitude for both $\alpha = 2{\cdot}10^{-5}$ and $\alpha = 3{\cdot}10^{-5}$. For $\alpha = 3{\cdot}10^{-5}$ the recall is always 100% (i.e., AGraMi provides all subgraphs) except for $\tau = 63.600$ that is 95%. For $\alpha = 2{\cdot}10^{-5}$ the recall is always over 90%. For the MiCo dataset, the performance gain is significant, nearly an order of magnitude when $\alpha = 4{\cdot}10^{-4}$ and

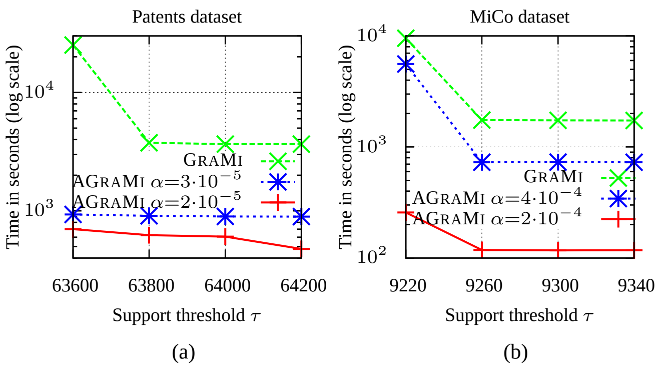


Figure 10: Performance evaluation of GraMi **and** AGraMi **with different values for the approximation parameter.**

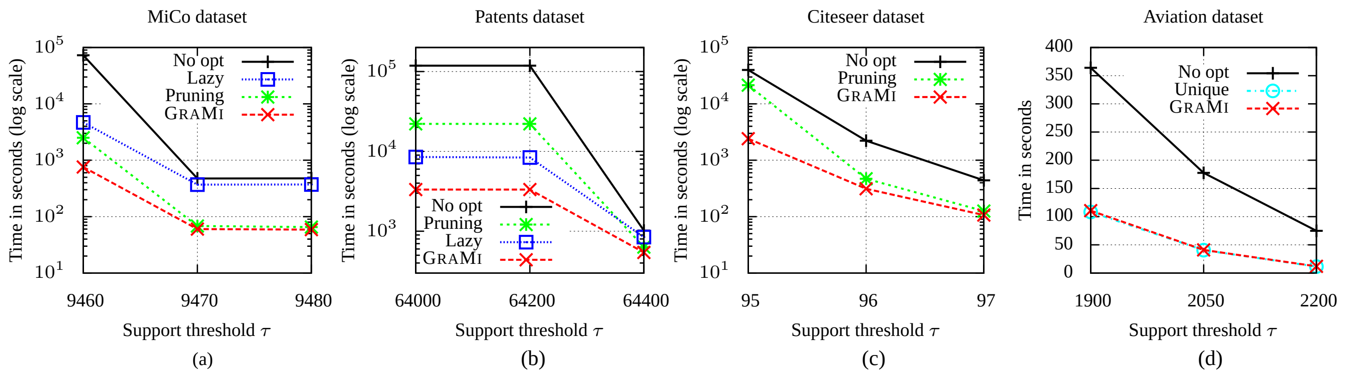


Figure 11: The effect of optimizations. No opt: **Algorithm** ISFREQUENTHEURISTIC **(Section 3.2).** Lazy: **Lazy search and decomposition optimizations enabled.** Pruning: **Only pruning push-down optimization enabled.** Unique: **Only unique labels optimization enabled.** GRAMI: **All optimization enabled (Algorithm** ISFREQUENT**).**

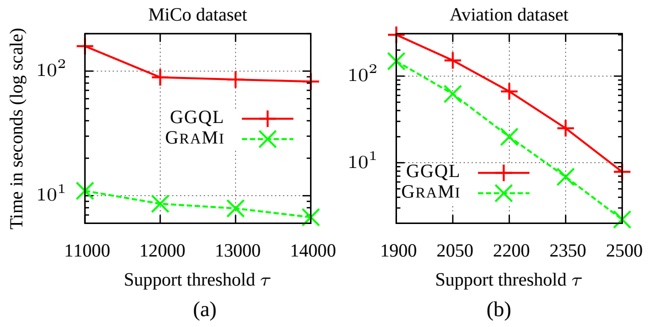


Figure 12: Performance comparison between GRAMI **and** GGQL**; a modified version of** GRAMI **that replaces ISFREQUENT with a counting function based on GraphQL**

nearly two orders of magnitude when $\alpha = 2{\cdot}10^{-4}$. Interestingly, the recall is always 100%.

Optimizations. This experiment demonstrates the effect of the optimizations discussed in Section 3.3 on mining the different datasets. A summary is illustrated in Fig. 11. For the MiCo dataset, the most effective optimization is *Push-down pruning* (denoted by Pruning in Fig. 11a) that achieves an improvement of up to 2 orders of magnitude. Following that, are the *Lazy search* and the *Decomposition pruning* optimizations, both are combined and denoted by Lazy in Fig. 11a. The two optimizations accomplish an improvement of up to an order of magnitude. Last comes the *Automorphism* and *Unique labels* optimizations that achieve only 4% improvement, since most of the frequent subgraphs in the MiCo dataset neither have automorphisms nor unique labels. For presentation clarity in Fig. 11a, we do not illustrate the results of the last two optimization methods. A similar trend also applies to Patents and Citeseer datasets (Figs. 11b and 11c).

For the Aviation dataset (Fig. 11d), a different optimization trend is noticed since this dataset is fundamentally different than MiCo Patents and Citeseer. In this case, the most effective optimization is Unique labels (denoted by Unique in Fig. 11d). As discussed earlier, the Aviation dataset is extremely sparse and has a very large number of distinct node labels, thus, the Unique label optimization is very effective. In contrast to the previous cases, all other optimizations do not offer any improvement and are not illustrated.

Comparison with subgraph isomorphism techniques. To address the frequent data mining problem, we may also employ subgraph isomorphism techniques [21]. For comparison, we have implemented GGQL; a modified version of GRAMI that replaces ISFREQUENT with a frequency evaluation function based on GRAPHQL [16]; one of the fastest state-of-the-art subgraph isomorphism techniques [21]. Clearly, as illustrated in Fig. 12, GRAMI outperforms GGQL by at least 3 times and up to more than an order of magnitude. This is easily justifiable since GRAMI uses several optimizations and visits only the necessary nodes in the input graph to solve the frequent subgraph mining problem.

6. RELATED WORK

This section discusses related work in many different directions.

Transactional mining. This setting is concerned with mining frequent subgraphs on a dataset of many, usually small, graphs. FSQ [18] construct new candidate patterns by joining smaller frequent ones. The drawback of this approach is the costly join operation and the pruning of false positives. GSPAN [29] proposes a variation of the pattern *growth* approach. It uses an extension mechanism, where subgraphs grow directly from a single subgraph instead of joining two previous subgraphs. Other methods focus on particular subsets of frequent subgraphs. MARGIN [26] returns maximal subgraphs only, whereas CLOSEGRAPH [30] generates subgraphs that have strictly smaller support than any of their parts. LEAP [28] and GRAPHSIG [24], on the other hand, discover important subgraphs that are not necessarily frequent.

Although GRAMI focuses on the single large graph setting, it may be easily specialized to also support graph transactions.

Single graph mining. On the equally important single graph setting there exists less work. The major difference is the definition of an appropriate anti-monotone support metric (Section 2). SIGRAM [20] uses the *MIS* metric and proposes an algorithm that finds frequent connected subgraphs in a single, labeled, sparse and undirected graph. SIGRAM follows a *grow-and-store* approach, i.e., it needs to store intermediate results in order to evaluate frequencies. Overall, SIGRAM needs to enumerate all isomorphisms and relies on the expensive computation of *MIS* (which is *NP*-complete), thus the method is very expensive in practice.

Since the number of intermediate embeddings increases exponentially with the graph size, such approaches do not scale for large graphs. In contrast, GRAMI does not need to construct all the isomorphisms, hence, it can scale to much larger graphs. More importantly, GRAMI supports frequent subgraph and pattern mining (Problems 1 and 2 respectively). Thus, it allows for exact isomorphism matching and the more general distance-constrained pattern matching. Additionally, GRAMI supports constraint-based mining and works on directed, undirected, single and multi-labeled graphs.

Approximate mining. There is work on approximate techniques for solving the frequent subgraph mining problem as well. In GREW [19], the authors propose a heuristic approach that prunes large parts of the search space, but discovers only a small subset of the answers. GAPPROX[3] employs an approximate version of the *MIS* metric. It mainly relies on enumerating all intermediate isomorphisms but allows approximate matches. SEUS [14] is another approximate method that constructs a compact summary of the input graph. This facilitates pruning many infrequent candidates, however, it is only useful when the input graph contains few and very frequent subgraphs. SUBDUE [7] is a branch-and-bound technique that mines subgraphs that can be used to compress the original graph. Finally, Khan *et al.* [17] propose proximity patterns, which relax the connectivity constraint of subgraphs and identify frequent patterns that cannot be found by other approaches.

In contrast to the existing work, AGRAMI, approximate version of GRAMI, may miss some frequent subgraphs, but the returned results do not have false positives.

Subgraph isomorphism. The frequent subgraph mining problem relies on the computation of subgraph isomorphisms. This problem is NP-complete and the first practical algorithm that addresses this problem follows a backtracking technique [27]. Since then, several performance enhancements were proposed, ranging from CSP based techniques [23], search order optimization [16], indexing [31] and parallelization [25].

Although the state-of-the-art subgraph isomorphism techniques lead to significant improvements, they are not as effective in the frequent subgraph mining problem for two reasons: First, subgraph isomorphism techniques are effective in finding all appearances of a subgraph, while for the frequent subgraph mining task, it is sufficient to find the minimum appearances that satisfy the support threshold; this difference affects the way graph nodes are traversed, minimizing the number of node visits during search. Additionally, modern techniques employ global pruning and indexing techniques. Forming such structures on large graphs results in a huge and often unacceptable overhead. GRAMI is based on a novel CSP method that overcomes the previous shortcomings and outperforms state-of-the-art subgraph isomorphism techniques by up to an order of magnitude. This is experimentally validated in Section 5.

Pattern matching. There is work on pattern matching over graphs as well. R-JOIN [4] supports reachability queries in a directed graph; If two nodes v and v' are reachable in the query then their corresponding mappings u and u' in the graph must also be reachable. DISTANCE-JOIN [34] extends the idea to undirected graphs and accommodates constraints on the distance in the path. GRAMI presents an extension to support frequent pattern mining, the extended version adopts the pattern definition from [34].

7. 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 introduces GRAMI; a versatile algorithm for discovering frequent patterns in a single large graph, a significantly more difficult problem compared to the usual case of mining a set of small graph transactions. The modeling of the frequency evaluation operation as a constraint satisfaction problem is the crux idea of GRAMI. We complement this idea with a set of optimizations that allows for the efficient performance of GRAMI. We also implement a version that supports structural and semantic constraints and an approximate version that scales to larger graphs. Our experimental results with real datasets demonstrate the effectiveness of GRAMI which is up to 2 orders of magnitude faster than existing approaches while discovering larger and more interesting frequent patterns.

8. REFERENCES

[1] B. Bringmann. *Mining Patterns in Structured Data*. PhD thesis, KU Leuven, 2009.

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

[3] 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.

[4] 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.

[5] Y.-R. Cho and A. Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *Trans. Info. Tech. Biomed.*, 14(1):30–36, Jan. 2010.

[6] W.-T. Chu and M.-H. Tsai. Visual pattern discovery for architecture image classification and product image search. In *Proc. of ICMR*, pages 27:1–27:8, 2012.

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

[8] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted CSP. In *Proc. of AAAI*, pages 22–27, 2006.

[9] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *Proc. of ICDM*, pages 35–42, 2003.

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

[11] C. Domshlak, R. I. Brafman, and S. E. Shimony. Preference-based configuration of web page content. In *Proc. of IJCAI*, pages 1451–1456, 2001.

[12] M. Fiedler and C. Borgelt. Subgraph support in a single large graph. In *Proc. of ICDMW*, pages 399–404, 2007.

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

[14] S. Ghazizadeh and S. S. Chawathe. Seus: Structure extraction using summaries. In *Proc. of DS*, pages 71–85, 2002.

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

[16] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. of SIGMOD*, pages 405–418, 2008.

[17] A. Khan, X. Yan, and K.-L. Wu. Towards proximity pattern mining in large graphs. In *Proc. of SIGMOD*, pages 867–878, 2010.

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

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

[20] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[21] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, Dec. 2012.

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

[23] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:228–250, 1979.

[24] 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.

[25] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, May 2012.

[26] L. T. Thomas, S. R. Valluri, and K. Karlapalem. MARGIN: Maximal frequent subgraph mining. *TKDD*, 4(3):10:1–10:42, 2010.

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

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

[29] X. Yan and J. Han. GSPAN: Graph-based substructure pattern mining. In *Proc. of ICDM*, pages 721–724, 2002.

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

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

[32] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.

[33] 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.

[34] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.