Expert Systems With Applications 141 (2020) 112897

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Extending association rules with graph patterns

Xin Wang[a,b,*], Yang Xu[a], Huayi Zhan[c]

[a] Southwest Jiaotong University, Chengdu, China
[b] National Engineering Laboratory of Integrated Transportation Big Data Application Technology, Chengdu, China
[c] Sichuan Changhong Electric Co. Ltd, Mianyang, China

ARTICLE INFO

Article history:
Received 23 August 2018
Revised 23 August 2019
Accepted 24 August 2019
Available online 5 September 2019

ABSTRACT

We propose a general class of graph-pattern association rules (GPARs) for social network analysis. Extending association rules for itemsets, GPARs can help us discover associations among entities in social networks and identify potential customers. Despite the benefits, GPARs bring us challenges: the problem of GPARs discovery is already intractable, not to mention mining over large social networks. Nonetheless, we show that it is still feasible to discover GPARs from large social networks. We first formalize the GPARs mining problem and decompose it into two subproblems: Frequent pattern mining and rule generation. To address two subproblems, we develop a parallel algorithm along with an optimization strategy to construct *DFS code graphs*, whose nodes correspond to frequent patterns. We also provide efficient algorithms to generate (resp. representative) GPARs by using (resp. maximal) frequent patterns. Using real-life and synthetic graphs, we experimentally verify that our algorithms not only scale well but can also identify interesting GPARs with high quality among social entities.

1. Introduction

Association rules have been studied for discovering regularities between items in relational data (Agrawal, Imielinski, & Swami, 1993). They have a traditional form $X \Rightarrow Y$, where X and Y are disjoint itemsets. For example, {diaper}⇒ {beer} is an association rule indicating that if a customer buys diapers, then he will also buy beer.

In recent years, there has been interest in studying how to identify association rules in graph data because such rules can capture associations among social entities and be used in social marketing. For example, Fan, Wang, Wu, and Xu (2015) extended association rules with special graph patterns and used the rules in social media marketing and social recommendation. While these rules are not capable of modelling more complicated associations among social entities, their consequents, as pattern graphs, take only a single edge. As a result, numerous meaningful rules cannot be captured. Nonetheless, graph-pattern association rules are more involved with generalized patterns as antecedents and consequents. This highlights the need for extending association rules with general graph patterns and discovering these rules on social graphs.

Example 1. A fraction of a social graph G is shown in Fig. 1 (a), where each node denotes a person with name as an identifier and job title (e.g., project manager (PM), database administrator (DBA), programmer (PRG), business analyst (BA) and software tester (ST)), and each edge indicates friendship, *e.g.*, (Bob, Mat) indicates that Bob and Mat are friends. The graph G is distributed to sites S_1, S_2 and S_3.

One can easily infer the following rule from graph G that among a group of people with titles PM, BA, DBA, PRG and ST, if PM and BA, PM and DBA, DBA and PRG, DBA and ST, PRG and ST are friends, then the chances are that PRG and BA, BA and DBA are likely to be friends. The rule R, referred to as the graph-pattern association rule (GPAR), is defined on graphs rather than itemsets. As shown in Fig. 1 (b), the antecedent and consequent of the rule are represented as graph patterns, *i.e.*, Q_l and Q_r. The graph patterns specify conditions on various entities in a social graph in terms of topological constraints. With the rule, one can infer social relationships and recommend friends to others who they will most likely be interested in, *e.g.*, recommend Mary to Tim, and Roy to Mary. Not limited to social recommendation, GPARs can also be used in *e.g.*, link prediction (Ebisu & Ichise, 2019; Lin, Song, Shen, & Wu, 2018), graph repairing (Cheng, Chen, Yuan, & Wang, 2018), and network evolution analysis (Chaturvedi, Tiwari, & Spyratos, 2019).

While useful, GPARs mining creates challenges. (1) Traditional techniques for transactional data cannot be applied. Superficially, a GPAR can also be defined following its traditional counterpart, *i.e.*, with antecedent and consequent being defined on edge sets. Then,

* Corresponding author.
E-mail addresses: xinwang@swjtu.cn (X. Wang), xuyang@my.swjtu.edu.cn (Y. Xu), huayi.zhan@changhong.com (H. Zhan).

https://doi.org/10.1016/j.eswa.2019.112897
0957-4174/

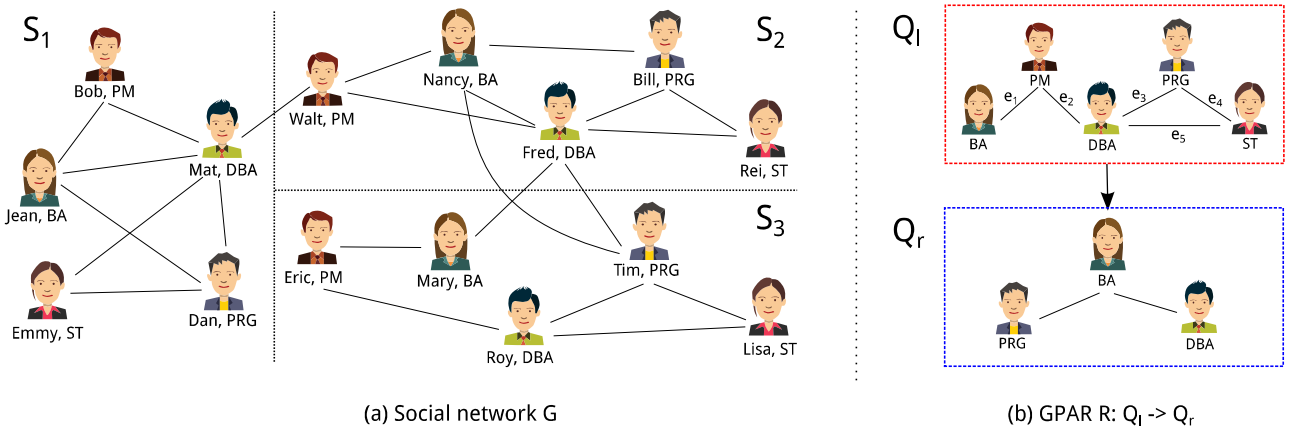


Fig. 1. Graph & Association as graph patterns.

traditional techniques can be trivially applied. However, in the context of a single graph, it is not applicable for mining frequent edge sets along the same line as frequent itemset mining on transactional data. In addition, the prior method for special GPARs is also not feasible because it takes a set of single-edge consequents as input and outputs GPARs with these consequents. (2) Social graphs are often large and distributively stored; hence, centralized techniques are no longer viable. One may easily verify that the rule R, along with its support, may not be correctly computed from G without distributed algorithms. Worse still, mining computations are often cost-prohibitive. With these comes the need for an algorithm to allow a high degree of parallelism and to efficiently discover GPARs.

The practical need for GPARs raises the following fundamental questions. (1) As will be seen shortly, conventional support and confidence metrics no longer work for GPARs. Thus, what metrics can be used to measure support and confidence? (2) Is there any parallel algorithm for GPARs mining over distributed graphs? (3) As there often exist excessive GPARs from a large graph, how can we develop techniques to generate "represented" GPARs to facilitate inspection and interpretation of GPARs?

Contributions. This paper proposes generalized GPARs, which extend association rules with general graph patterns and provides effective algorithms for discovering (representative) GPARs under a distributed scenario.

(1) We first propose generalized graph-pattern association rules to capture complex social relations among social entities (Section 2.3). We next show that the GPARs mining problem can be decomposed into two subproblems, *i.e.,* frequent pattern mining and rule generation. We then outline an algorithm for the problem. (Section 2.4).

(2) We study the frequent pattern mining (FPM) problem under a distributive scenario (Section 3). Inspired by the strategy used for the constraint satisfaction problems, *i.e.,* "look-ahead & backtracking", we develop an algorithm FPMiner to generate a *DFS code graph* $\mathcal{G}_c$, whose nodes correspond to frequent patterns, *i.e.,* patterns with support above a threshold (Section 3.1). The algorithm works in parallel, hence obtains desirable performance: it computes the support of a pattern Q in $O(|E_f|((k+1)2^{k+1})^{k-1})$ time and incurs $O((k+1)|\mathcal{F}||V_f|)$ data shipment, where k indicates the level of the node to which the pattern Q corresponds in $\mathcal{G}_c$, and $|\mathcal{F}|$, $|E_f|$, and $|V_f|$ are the number of sites, crossing edges, and virtual nodes, respectively (see definitions of $\mathcal{F}$, E_f and V_f in Section 2.1). We also provide techniques for optimizing mining computations (Section 3.3).

(3) We study how to generate GPARs by using *DFS code graph* $\mathcal{G}_c$ (Section 4). Given $\mathcal{G}_c = (V_c, E_c)$ and confidence bound η, we develop an algorithm, denoted as RuleGen, to produce GPARs with confidence above η in $O(|V_c|(|V_c| + |E_c|))$ time, which are independent of the size of the underlying big graph G. We then study how to generate "representative" GPARs. We start from a notion of maximal frequent patterns, followed by the problem of maximal frequent pattern mining (MFPM). We show that MFPM does not increase the difficulty: after $\mathcal{G}_c$ is constructed, all the leaf nodes of $\mathcal{G}_c$ correspond to the set of maximal frequent patterns. Using maximal frequent patterns, we provide an algorithm, denoted by RepRuleGen, to generate "representative" GPARs, to reduce excessive GPARs and ease understanding.

(4) Using real-life and synthetic graphs, we experimentally verify the performance of our algorithms (Section 5). We find the following. (a) Our distributive algorithms for frequent pattern mining scale well with the increase of processors (n). For example, our algorithm FPMiner is on average 4.1, 2.9 and 2.5 times faster on three real-world social networks, when n increases from 4 to 20. (b) Our algorithms work reasonably well on large graphs. For example, on a graph with 4 million nodes and 53.5 million edges, FPMiner spends less than 5 minutes (269 s) and ships only 15.9% of the entire graph to discover frequent patterns using 20 processors. (c) The optimization technique for FPMiner is effective. For example, $\text{FPMiner}_{\text{opt}}$, the optimized algorithm of FPMiner, only requires 56.3% of the time, and ships 68% of the data of FPMiner, on average. (d) Our rule generation algorithm RuleGen is very efficient requiring less than 0.4 s, over real-life graphs. (e) Generating GPARs with maximal frequent patterns is very effective, as the "representative" GPARs only account for 1.57% of the entire ruleset, and cover more than 90% of the top-k GPARs, on average. (f) Our GPARs can predict missing relationships with an average accuracy of 43.8% on real-life graphs and outperform existing link prediction methods.

The work provides a full treatment for mining generalized graph-pattern association rules from large social graphs. It provides a parallel algorithm along with an optimization strategy for mining frequent patterns and develops techniques to generate (resp. "representative") GPARs with (resp. maximal) frequent patterns. Compared with earlier works, this work fills one critical void for mining generalized GPARs and yields a promising approach for social network analysis.

Related Work. This paper extends our prior work (Wang & Xu, 2018) by including the following new contributions: (1) Proofs

of Proposition 1 (Section 3.1) and Lemma 1 (Section 3.2); (2) a procedure EvalT and a detailed analysis of algorithm FPMiner (Section 3.2); (3) optimization strategy for local computation in *workers* (Section 3.3); (4) a maximal frequent pattern mining problem (MFPM) and techniques to generate "representative" GPARs (Section 4.2); and (5) a set of new experimental studies (Section 5), an efficiency and scalability test of $\text{FPMiner}_{\text{opt}}$, data shipment of FPMiner vs. $\text{FPMiner}_{\text{opt}}$, a performance evaluation over a new dataset *Amazon*, a performance comparison between RuleGen and RepRuleGen, and case studies over two real-life graphs.

We next categorize the related work as follows.

Association rules. Association rules, which are defined on relations of transaction data, were first introduced in Agrawal et al. (1993). Prior work on association rules for social networks (Schmitz, Hotho, Jäschke, & Stumme, 2006) and RDF knowledge bases mined conventional rules and Horn rules (as conjunctive binary predicates) (Galárraga, Teflioudi, Hose, & Suchanek, 2013) over a set of tuples with extracted attributes from social graphs instead of exploiting graph patterns. A special type of GPARs was proposed in Fan et al. (2015), where the consequents were defined as pattern graphs with a single edge.

Existing research has also shown that traditional techniques can produce too many association rules because for a frequent itemset of size k, there may exist $2^k - 2$ frequent subsets (Zaki, 2000). Therefore, maximal frequent itemsets and their mining algorithm were introduced by Gouda and Zaki (2001), as maximal frequent itemsets can imply and include all information of frequent itemsets.

This work proposes mining techniques for generalized GPARs, which are defined with general graph patterns as antecedent and consequent. Moreover, we show that the set of maximal frequent patterns can be easily obtained from a *DFS code graph* and used to generate "representative" GPARs.

Big graph analysis. In the era of big graphs, the development of algorithms for graph analytics has spawned a large amount of research. Sapountzi and Psannis (2018) categorized social network analysis into three types of tasks: Sentiment analysis & opinion mining, topic detection and collaborative recommendation, and depicted various methods and their associated frameworks for the tasks. A recent survey regarding data mining and processing frameworks for big graphs was conducted by Aridhi and Nguifo (2016), in which the problem of frequent pattern mining was formulated as two problems: graph transaction based and single graph based. This classification coincides with Jiang, Coenen, and Zito (2013).

As will be seen shortly, single graph based pattern mining is a subtask of GPARs mining, and the patterns discovered need to be organized well for rule generation. Although graph analytical platforms exist for single graph-based pattern mining, *e.g.,* Teixeira et al. (2015), their output may still need to be reorganized to facilitate ("representative") rule generalization.

Graph-pattern mining. The generalization from item sets to graph patterns also requires effective graph-pattern mining techniques to discover GPARs in big graphs beyond traditional rule mining techniques (Zhang & Zhang, 2002). The topic of graph-pattern mining has been intensively studied (see Jiang et al. (2013) for a survey). Two technical routes developed according to the essential differences in graph data.

(1) Algorithms for graph databases. The algorithms can be further categorized into two types: (i) Apriori methods (Inokuchi, Washio, & Motoda, 2000) that start with small graphs, and generate substructures by joining similar but slightly different frequent subgraphs, in a bottom-up manner; and (ii) pattern-growth methods (Huan, Wang, Prins, & Yang, 2004), which extend a frequent graph by adding new nodes and edges in every possible position. (2) Algorithms for single large graphs. For example, Elseidy, Abdelhamid, Skiadopoulos, and Kalnis (2014) proposed using a minimum image that preserves anti-monotonic property as a support metric to measure pattern frequency. To further improve efficiency, parallel techniques were studied in Talukder and Zaki (2016) and Teixeira et al. (2015). Talukder and Zaki (2016) developed a level-wise approach and effective pruning strategy under the distributive scenario. Teixeira et al. (2015) introduced a Pregel-inspired distributed platform Arabesque for frequent pattern mining.

More recently, de J. Costa, Bernardini, Artigas, and Viterbo (2019) proposed methods for mining direct acyclic patterns to identify retention patterns in undergraduate programmes. To identify periodic patterns in dynamic networks, Halder, Samiullah, and Lee (2017) proposed SPPMiner, a single pass supergraph based periodic pattern mining technique. SPPMiner stores all entities of the dynamic network only once and calculates common sub-patterns only once at each timestamp, thus achieving high efficiency.

However, techniques for frequent pattern mining over graph databases (Huan et al., 2004; Inokuchi et al., 2000) cannot be directly used to mine GPARs, as their anti-monotonic property does not hold in a single graph (Jiang et al., 2013). Our settings are quite different from de J. Costa et al. (2019) and Halder et al. (2017), as our method can mine general patterns rather than DAG patterns, and we are coping with static graphs rather than dynamic graphs. Our work differs from distributed techniques (Talukder & Zaki, 2016; Teixeira et al., 2015) as follows: we leverage both partial evaluation and asynchronous message passing to identify matches of candidate patterns in a distributive environment instead of level-wise evaluation or Pregel-based computation; Moreover, frequent patterns are our intermediate results, and our distributed technique organizes all frequent patterns in a graph structure, which facilitates ("representative") rule generation.

GPARs *mining*. Special GPARs and their mining techniques were introduced in Fan et al. (2015), where consequents of the GPARs are defined as pattern graphs with a single edge, and the algorithm is specifically developed to mine diversified GPARs with fixed consequent. Fan, Wu, and Xu (2016) next introduced quantified graph association rules, which extend GPARs with quantifiers to identify potential customers in social media marketing. Another work addressed mining GPARs over stream data (Namaki, Wu, Song, Lin, & Ge, 2017).

Our work differs from these studies in the semantics, *i.e.,* we are mining generalized GPARs, with antecedent and consequent represented by general graph patterns.

2. Graph-pattern association rules

In this section, we first review graphs, patterns, subgraphs, (subgraph) isomorphism and distributed graphs. We then introduce *DFS Code, DFS Lexicographic Order* and *DFS Code Tree*, followed by the definition of graph-pattern association rules.

2.1. Graphs, patterns, isomorphism & distributed graphs

We start with the notion of graphs, patterns, subgraphs, (subgraph) isomorphism, and distributed graphs.

Data graph. A data *graph* (or simply *graph*) is defined as $G = (V, E, L)$, where (1) V is a set of nodes and (2) $E \subseteq V \times V$ is a set of *undirected* edges, in which (v, v') or (v', v) denotes an undirected edge between node v and v'; and (3) each node v in V carries a tuple $L(v) = (A_1 = a_1, \cdots, A_n = a_n)$, where $A_i = a_i$ ($i \in [1, n]$) represents that the node v has a value a_i for the attribute A_i, and is denoted as $v.A_i = a_i$, *e.g., v*.name="Bob", *v*.job_title="PM". A *directed* graph is defined similarly, but with each edge (v, v') denoting a directed edge from v to v'.

In an undirected graph G, a *path* is a finite or infinite sequence of edges that connect a sequence of nodes that are all distinct from one another. A *path* in a directed graph is defined similarly, but with directed edges. A *tree* is an undirected graph in which any two nodes are connected by exactly one *path*. A *rooted tree* is a *tree* in which one particular node is designated as the *root*. The edges of a *rooted tree* are either away from or towards the *root*, in which case the structure becomes a *directed rooted tree*.

In the following, we refer to a graph as an undirected graph without specification and a directed graph, otherwise.

Pattern graph. A *pattern* graph (or simply *pattern*) Q is a graph (V_p, E_p, f), where V_p and E_p are the set of nodes and edges, respectively; f is a function defined on V_p such that for each node $u \in V_p f(u)$ is a predicate defined as a conjunction of atomic formulas of the form of '$A = a$' such that A denotes an attribute of the node u, and a is a value of A. Intuitively, $f(u)$ specifies search conditions imposed by u.

Subgraph. A graph $G' = (V', E', L')$ is a *subgraph* of $G = (V, E, L)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and for each $v \in V'$, $L'(v) = L(v)$. Similarly, a pattern $Q' = (V'_p, E'_p, f')$ is a *subgraph* of $Q = (V_p, E_p, f)$, denoted by $Q' \subseteq Q$, if $V'_p \subseteq V_p$, $E'_p \subseteq E_p$, and for each $u \in V'_p$, $f'(u) = f(u)$.

Isomorphism & subgraph isomorphism. Consider graph G and pattern Q, a node v in G satisfies the search conditions of a pattern node u in Q, denoted as $v \sim u$, if for each atomic formula '$A = a$' in $f(u)$, there exists an attribute A in $L(v)$ such that $v.A = a$. An *isomorphism* is a bijective function h from the nodes of Q to the nodes of G, such that (1) for each node $u \in V_p$, $h(u) \sim u$, and (2) (u, u') is an edge in Q if and only if $(h(u), h(u'))$ is an edge in G. A *subgraph isomorphism* (Cordella, Foggia, Sansone, & Vento, 2004) is an isomorphism from Q to a subgraph G_s of G. When an isomorphism h from pattern Q to a subgraph G_s of G exists, then G *matches* Q, and G_s is denoted as a *match* of Q in G. Abusing notations, we say that v in G_s is a *match* of u in Q, when $v \sim u$.

We denote by $Q(G)$ the set of matches of Q in G. We also denote the image $\mathsf{img}[Q, G]$ of Q in G by a set $\{(u, \mathsf{img}(u)) | u \in V_p\}$, where $\mathsf{img}(u)$, referred to as the image of u in G, consists of distinct nodes v in G as matches of u in Q.

Example 2. Consider pattern Q_l given in Fig. 1 (b) and graph G in Fig. 1 (a). The match set $Q_l(G)$ contains four entries, which are listed in the table below. The image $\mathsf{img}[Q, G]$ is a set with the following elements: (PM, {Bob,Walt,Eric}), (BA, {Jean,Nancy,Mary}), (DBA, {Mat,Fred,Roy}), (PRG, {Dan,Bill,Tim}), (ST, {Emmy,Rei, Lisa}).

Matches	PM	BA	DBA	PRG	ST
1	Bob	Jean	Mat	Dan	Emmy
2	Walt	Nancy	Mat	Dan	Emmy
3	Walt	Nancy	Fred	Bill	Rei
4	Eric	Mary	Roy	Tim	Lisa

Distributed data graphs. In practice, a graph G is often fragmented into a collection of subgraphs and stored in different sites (Husain, Doshi, Khan, & Thuraisingham, 2009; Rowe, 2009). A *fragmentation* $\mathcal{F}$ of a graph $G = (V, E, L)$ is $(F_1, \ldots, F_n)$, where each *fragment* F_i is specified by $(V_i \cup F_i.O, E_i \cup cE_i, L_i)$ such that

(a) $(V_1, \ldots, V_n)$ is a partition of V,
(b) $F_i.O$ is the set of nodes v' such that there exists an edge $e = (v, v')$ in E, $v \in V_i$ and node v' is in *another fragment*; we refer to v' as a *virtual node*, e as a *crossing edge*, and cE_i as the set of crossing edges; and
(c) $(V_i \cup F_i.O, E_i \cup cE_i, L_i)$ is a subgraph of G induced by $V_i \cup F_i.O$.

We assume *w.l.o.g.* that each F_i is stored at site S_i for $i \in [1, n]$. For multiple fragments residing in the same site, they are simply treated as a single fragment.

Here, F_i contains "local" nodes in V_i and virtual nodes in $F_i.O$ from other fragments. The edge set consists of (a) edges in E_i and (b) *crossing edges* in cE_i, *i.e.,* edges between local nodes in V_i and virtual nodes.

We use the following notations: (a) $V_f = \bigcup_{i \in [1,n]} F_i.O$ is the set of all virtual nodes in $\mathcal{F}$; (b) E_f is the set of all crossing edges in $\mathcal{F}$; and (c) $|\mathcal{F}|$ denotes the number of fragments in $\mathcal{F}$.

Example 3. As shown in Fig. 1 (a), a fragmentation of graph G is (F_1, F_2, F_3), where F_1, F_2 and F_3 are stored in sites S_1, S_2 and S_3, respectively. Take F_2 as an example, $F_2.O$ consists of virtual nodes Mat, Mary and Tim, and the crossing edges are (Walt, Mat), (Fred, Mary), (Fred, Tim) and (Nancy, Tim). One may verify that there exists a subgraph G_s with node set {Walt, Nancy, Mat, Dan, Emmy} of G, matches Q_l. To identify G_s, data must be transferred between S_1 and S_2.

Remarks. To enrich the semantics of node matching, we use multiple-labelled graphs to model our data (resp. pattern) graphs, *i.e.,* a node in the graph (resp. pattern) can be associated with multiple attribute-value pairs. However, to simply discussion, we choose an attribute *A, e.g.,* job title, from the attribute set, and use the value of *v.A, e.g.,* PM, as node label of v, for the mining computation. Then, the chosen attribute is denoted as A_m, and the domain of A_m in G is denoted as $\mathcal{L}_{\mathcal{A}}$.

2.2. DFS code, DFS lexicographic order & DFS code tree

The notions of *DFS code* and *DFS code tree* are introduced in Yan and Han (2002). To make the paper self-contained, we cite them as follows (rephrased).

We start from *DFS Tree* and *Edge Ordering*.

DFS tree. Given a graph G, its *DFS tree* T_G can be built via a depth-first search in G from a node. The traversal sequence of nodes during T_G construction forms a linear order.

When using subscripts, *e.g.,* $i \in [1, n]$, to label the linear order according to their visiting sequence, v_i is considered to be visited before v_j, if $i < j$. Thus, v_0 and v_n are viewed as the *root* and *rightmost node*, respectively; the straight path from v_0 to v_n is denoted as the *rightmost path*; and an edge $e = (v_i, v_j)$ can be represented as an ordered pair (i, j), based on the subscripts of endpoints v_i and v_j and their visiting sequence. The graph with associated subscripts is denoted as G_l.

Edge ordering. Based on G_l, which encodes the linear order of T_G, a binary relation $\prec_{E,T}$ can be constructed on the edge set E with below rules. Assume $e_1 = (i_1, j_1)$, $e_2 = (i_2, j_2)$, (a) if $i_1 = i_2$ and $j_1 < j_2$, $e_1 \prec_{E,T} e_2$; (b) if $i_1 < j_1$ and $j_1 = i_2$, $e_1 \prec_{E,T} e_2$; and (c) if $e_1 \prec_{E,T} e_2$ and $e_2 \prec_{E,T} e_3$, $e_1 \prec_{E,T} e_3$. The relation $\prec_{E,T}$ specifies the *edge ordering w.r.t.* T_G.

DFS code. Given *DFS tree* T_G along with $\prec_{E,T}$ of a graph G, the *DFS code* of graph G *w.r.t.* T_G, denoted as $\alpha(T_G)$, is an edge sequence such that $e_i \prec_{E,T} e_{i+1}$ for $i \in [0, |E| - 1]$.

DFS lexicographic order. For a graph G, there exists a set $\mathcal{F}$ of *DFS Trees*. Let $\mathcal{Z} = \{\alpha(T) | T \in \mathcal{F}\}$, *i.e.,* $\mathcal{Z}$ be a set consisting of all the *DFS code* of G. Let $\prec_L$ denote the linear order of the entries in $\mathcal{L}_{\mathcal{A}}$. Then, the lexicographic combination of $\prec_{E,T}$, and $\prec_L$ is a linear order $\prec_e$ on the set $E \times \mathcal{L}_{\mathcal{A}} \times \mathcal{L}_{\mathcal{A}}$ (see Yan and Han (2002) for more details), and the *DFS Lexicographic Order* is a linear order, which is defined as follows.

If $\alpha(T_1) = (a_0, a_1, \cdots, a_m)$, $\alpha(T_2) = (b_0, b_1, \cdots, b_n)$ and $\alpha(T_1)$, $\alpha(T_2) \in \mathcal{Z}$, then $\alpha(T_1) \leq \alpha(T_2)$ *if and only if* one of the following is true:

(i) $\exists t, 0 \leq t \leq \min(m, n)$, $a_k = b_k$ for $k < t$, $a_t \prec_e b_t$; and
(ii) $a_k = b_k$, for $0 \leq k \leq m$ and $n \geq m$.

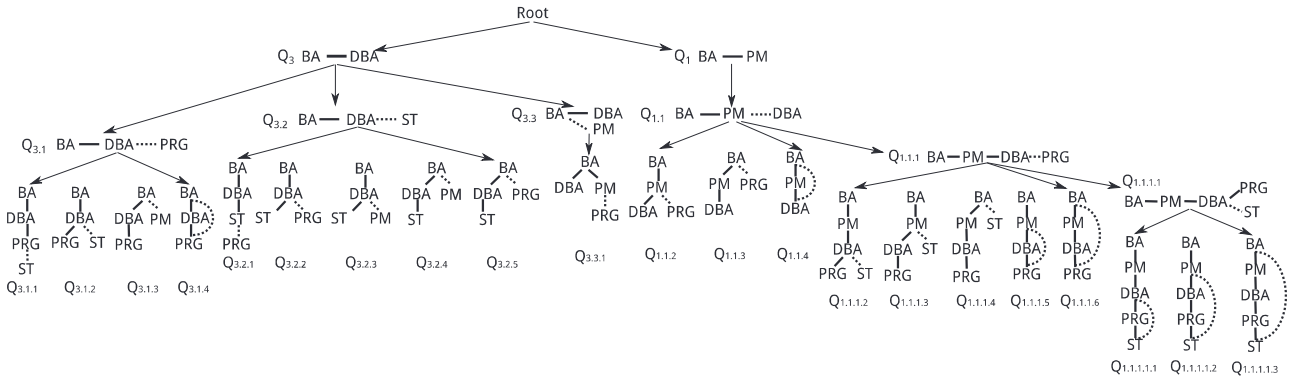


Fig. 2. A fraction of code tree $\mathcal{T}_c$.

Given a set $\mathcal{Z}$ of *DFS code*, one can sort them via *DFS lexicographic order*; then, the minimum *DFS code*, denoted by $\min(G)$, can be chosen as the canonical label of G, and graph isomorphism can be determined by comparing $\min(G)$ (Yan & Han, 2002).

DFS code tree. A *DFS code tree* $\mathcal{T}_c$ is a rooted directed tree, where (a) the root is a virtual node, (b) each non-root node, denoted as v_α, corresponds to a *DFS code* α, (c) for a node v_α with *DFS code* $(e_0, \cdots, e_k)$, its child must have a valid *DFS code* in the form of $(e_0, \cdots, e_k, e')$, and (d) the order of the *DFS code* of v_α's siblings satisfies the *DFS lexicographic order*.

Similarly, a *DFS code graph* $\mathcal{G}_c$ is a directed graph with a single root as a virtual node, and non-root nodes correspond to *DFS code*.

Example 4. Consider pattern Q_l of Fig. 1 (b). Among its *DFS trees*, T_G, which is rooted at BA and with edge set $\{e_i | i \in [1, 4]\}$, is a special *DFS tree* because its minimum *DFS code* $\min(Q_l) = (e_1, e_2, e_3, e_4, e_5)$, is generated based on it. Furthermore, Q_l can be represented by its canonical label $\min(Q_l)$ and corresponds to the node, labelled by $Q_{1.1.1.1.1}$ on the *DFS code tree* $\mathcal{T}_c$, shown in Fig. 2. One may verify that $\mathcal{T}_c$ is rooted at a virtual node, and each non-root node corresponds to a *DFS code*, represented by its corresponding pattern. Observe that each directed edge (v_α, v'_α) in $\mathcal{T}_c$ indicates that the *DFS code* of v'_α is generated from the *DFS code* of v_α by including a new edge from nodes on the rightmost path of α. For example, an edge from Q_3 to $Q_{3.1}$ indicates that the *DFS code* of $Q_{3.1}$ is generated from the *DFS code* of Q_3 by inserting a new edge, represented by a dashed line connecting the rightmost node DBA and a new node labelled by PRG.

Notations. (1) Pattern Q is *connected* if, for each pair of nodes in Q, there exists a path in Q between them. (2) Given a pattern Q along with its DFS tree T_G, we refer edges that are in T_G as *forward edges* and the remaining edges as *backward edges*. (3) We refer to *DFS code, DFS code tree* and *DFS code graph* as *code, code tree* and *code graph*, respectively, when it is clear from the context. (4) Given code α, we refer to its corresponding pattern as $Q(\alpha) = (V(\alpha), E(\alpha))$. (5) The *size* $|G|$ of G (resp. $|Q|$ of Q) is $|V| + |E|$ (resp. $|V_p| + |E_p|$), *i.e.,* the total number of nodes and edges in G (resp. Q). (6) Given a directed graph G, node v' is a *descendant* of v if there is a directed path from v to v'. (7) Given a directed graph G with a single root v_R, we denote node v as the k-th level node of G, if the shortest path from v_R to v is k; we also denote the longest shortest path from v_R to a node v in G as the height of G.

2.3. Graph-pattern association rules

We now define graph-pattern association rules.

GPARs. A *graph-pattern association rule* (GPAR) R is defined as $Q_l \Rightarrow Q_r$, where Q_l and Q_r (1) are both patterns that are connected and include at least one edge, and (2) share nodes but have no edge in common. We refer to Q_l and Q_r as the *antecedent* and *consequent* of R, respectively.

The rule states that in a graph G, if there is an isomorphism mapping h_l from Q_l to a subgraph G_1 of G, then there likely exists another mapping h_r from Q_r to another subgraph G_2 of G, such that for each $u \in V_l \cap V_r$, if it is mapped by h_l to v in G_1, then it is also mapped by h_r to the same v in G_2, *i.e.,* there is a node in G that is a match of pattern node u in Q_l and Q_r, simultaneously, for all the common nodes u of Q_l and Q_r.

We model a GPAR R as *a pattern* Q_R by extending Q_l with the edge set of Q_r. We consider practical and nontrivial GPARs by requiring that (1) Q_R, Q_l and Q_r are connected; (2) Q_l and Q_r are *nonempty, i.e.,* each of them has at least one edge; and (3) Q_l and Q_r do not have edges in common.

Example 5. As shown in Fig. 1 (b), the association rule described in Example 1 can be expressed as a GPAR R: $Q_l \Rightarrow Q_r$, with Q_l and Q_r as its antecedent and consequent, respectively. The GPAR R can be modelled as a pattern graph Q_R by extending Q_l with the edge set of Q_r.

2.4. GPARs mining

We first define the support and confidence for GPARs. We then outline an algorithm for mining GPARs from big social graphs G.

Support. The support of a pattern Q in a single graph G, denoted by $\text{supp}(Q, G)$, indicates the appearance frequency of Q in G. Similar to the association rules for itemsets, the support measure for patterns should be *anti-monotonic, i.e.,* for patterns Q and Q', if $Q' \sqsubseteq Q$, then $\text{supp}(Q', G) \geq \text{supp}(Q, G)$ for any G, as an *anti-monotonic* support helps to prune the search space. Several *anti-monotonic* support metrics for patterns exist, *e.g.,* minimum image (MNI) (Elseidy et al., 2014), harmful overlap (Fiedler & Borgelt, 2007), and maximum independent sets (Gudes, Shimony, & Vanetik, 2006). As MNI can be more efficiently computed, we adopt MNI as a support metric of patterns, and define it as follows:

$$\text{supp}(Q, G) = \min\{|\text{img}(u)| \mid u \in V_p\},$$

where $\text{img}(u)$ is the image of pattern node u in G. One can easily verify that this support measure is *anti-monotonic*.

Similarly, we define support of a GPAR R as

$$\text{supp}(R, G) = \text{supp}(Q_R, G),$$

by treating R as a pattern Q_R.

Confidence. To determine the likelihood of Q_r holding when Q_l holds, we follow the conventional confidence metric for association

Algorithm Miner

Input: Graph G, support θ, confidence η.
Output: A set of GPARs R with $\mathsf{supp}(R, G) \geq \theta$, $\mathsf{conf}(R, G) \geq \eta$.
1. mine patterns R with $\mathsf{supp}(R, G) \geq \theta$ from G;
2. generate GPARs R with $\mathsf{conf}(R, G) \geq \eta$ from Q;
3. **return** GPARs;

Fig. 3. Algorithm Miner.

rules and define the confidence of a GPAR R in G as follows:

$$\mathsf{conf}(R, G) = \frac{\mathsf{supp}(Q_R, G)}{\mathsf{supp}(Q_l, G)}.$$

Example 6. Recall Example 2. Although there are four matches of Q_l in G, the support $\mathsf{supp}(Q_l, G)$ of Q_l in G is 3 because $|\mathsf{img}(u)| = 3$ for each u in Q_l. Similarly, the image $\mathsf{img}(Q_R, G)$ of Q_R in G is a set $\{(\mathtt{PM}, \{\mathtt{Bob}, \mathtt{Walt}\}), (\mathtt{BA}, \{\mathtt{Jean}, \mathtt{Nancy}\}), (\mathtt{DBA}, \{\mathtt{Mat}, \mathtt{Fred}\}), (\mathtt{PRG}, \{\mathtt{Dan}, \mathtt{Bill}\}), (\mathtt{ST}, \{\mathtt{Emmy}, \mathtt{Rei}\})\}$, hence $\mathsf{supp}(Q_R, G) = 2$. As a result, the confidence $\mathsf{conf}(R, G)$ of R is $\frac{2}{3}$.

Mining algorithm. A common strategy adopted by traditional association rule mining algorithms is to decompose the problem into two major subtasks: frequent itemset mining, and rule generation, which suggests mining GPARs as follows: (1) Mine frequent patterns Q_R from big social graphs G; and (2) generate GPARs with frequent patterns Q_R.

Following this strategy, we outline an algorithm, denoted by Miner and shown in Fig. 3. The algorithm takes a graph G, support threshold θ, and confidence threshold η as input and returns a set of GPARs as output. In a nutshell, it first mines frequent patterns Q_R with support above the threshold from graph G (line 1). It then generates a set of GPARs satisfying the confidence constraints with frequent patterns Q_R (line 2). Finally, Miner returns qualified GPARs as the result (line 3). In Sections 3 and 4, we introduce how to mine frequent patterns Q_R and how to generate GPARs with Q_R, respectively.

Remark. Our technique can be readily extended to graphs with edge labels. Indeed, an edge-labelled graph can be converted into a node-labelled graph as follows: for each edge e, add a "dummy" node carrying the label of e, along with two unlabelled edges.

3. Frequent pattern mining

In this section, we first investigate the frequent pattern mining problem (Section 3.1); we then develop a parallel algorithm for the problem (Section 3.2) and provide an optimization strategy to improve performance (Section 3.3).

3.1. Frequent pattern mining problem

To mine GPARs from graphs G, one first needs to find a set of frequent patterns from G. This leads to the *frequent pattern mining* (FPM) problem.

Problem. The problem can be stated as follows.

- *Input*: A graph G and support threshold $\theta \leq \max\{\mathsf{sum}(a) | a \in \mathcal{L}_A\}$.
- *Output*: A set S of frequent patterns Q of G such that $\mathsf{supp}(Q, G) \geq \theta$ for any Q in S.

Here, $\mathsf{sum}(a)$ is the appearance number of nodes with $A_m = a$, and $\max\{\mathsf{sum}(a) | a \in \mathcal{L}_A\}$ indicates the upper bound of support for a graph G. A reasonable support, *e.g.*, $\theta \leq \max\{\mathsf{sum}(a) | a \in \mathcal{L}_A\}$ can avoid trivial results, *i.e.*, $\mathsf{S} = \emptyset$.

However, the FPM problem is nontrivial. Its decision problem is to determine, given graph G, support threshold θ and integer k, whether there exists a set S of frequent patterns Q in G with $\mathsf{supp}(Q, G) \geq \theta$ for any Q in S, and size $|\mathsf{S}| \geq k$ (k is trivially no larger than $|V|^{|\mathcal{L}_A|}$). One can show the following by reduction from the NP-complete subgraph isomorphism problem (Cordella et al., 2004).

Proposition 1. *The* FPM *decision problem is* NP*-hard.*

Proof. We show that the decision problem of FPM is NP-hard by reduction from the *subgraph isomorphism problem* (ISO), which is known as an NP-complete problem (Cordella et al., 2004). Given a graph G_1 and a pattern Q_1, ISO determines whether there exists a subgraph of G_1 that is isomorphic to Q_1. Given any instance of ISO, we construct an instance of an FPM problem by setting $G = G_1$, $k = 1$, and $\theta = 1$. □

The construction is obviously in PTIME. We next verify that there exists a subgraph of G_1 that is isomorphic to Q_1, if and only if there exists a set S of frequent patterns, such that $\mathsf{supp}(Q_i, G_1) \geq \theta$ for each $Q_i \in \mathsf{S}$, and $|\mathsf{S}| \geq k$.

(1) Assume that there exists a subgraph G_s of G_1 that is isomorphic to Q_1. Then, Q_1 must be a frequent pattern because $\mathsf{supp}(Q_1, G_1) \geq 1$, and $|\mathsf{S}| \geq 1$ as S contains at least one element, *i.e.*, Q_1.
(2) Conversely, if there is a pattern Q_1 with $\mathsf{supp}(Q_1, G_1) \geq 1$. Then, there must exist a subgraph of G_1 that is isomorphic to Q_1, since otherwise $\mathsf{supp}(Q_1, G_1)$ cannot be above 1.
As ISO is known to be NP-complete, the FPM problem is NP-hard. □

3.2. Distributed frequent pattern mining algorithm

The NP-hardness of the FPM problem indicates that no polynomial-time algorithm exists for the problem. Moreover, real-life social graphs are often very large, with billions of nodes and edges (Grujic, Bogdanovic-Dinic, & Stoimenov, 2014). These results together create extremely high computational costs for the problem. Nevertheless, consider that social graphs are often distributively stored, and parallel processing is verified to be effective for large-scale computations; it is necessary to develop parallel algorithms for the problem such that pattern mining can be more efficiently processed. Motivated by this, we develop a parallel algorithm to mine frequent patterns.

Theorem 1. *There exists a parallel algorithm for the* FPM *problem that computes a code graph $\mathcal{G}_c$ of a fragmented graph $\mathcal{F}$ such that (a) each node in $\mathcal{G}_c$ corresponds to a code representing a frequent pattern, and (b) the support computation for a node at the k-th level in $\mathcal{G}_c$ incurs $O((k+1)|\mathcal{F}||V_f|)$ data shipment, and is in $O(|E_f|((k+1)2^{k+1})^{k-1})$ time.*

Proof. We show Theorem 1 by presenting an algorithm as a constructive proof. □

The algorithm, denoted as FPMiner and shown in Fig. 4, takes a fragmented graph $\mathcal{F} = (F_1, \ldots, F_n)$ and support threshold θ as input, works with a *coordinator* S_c and a set of *working sites* (*a.k.a. workers*) S_i, mines frequent patterns *in parallel* by following bulk synchronous processing, and tracks frequent patterns in a graph structure $\mathcal{G}_c$ (details about $\mathcal{G}_c$ will be provided.). Before presenting the algorithm, we first introduce a notion of *partial matches* and auxiliary structures used by FPMiner.

Partial matches. Consider that some matches of a pattern may cross over multiple sites, and each *worker* may only have a part of these matches. We refer to these match fragments at each *worker* as *partial matches* and associate a unique id with them if they belong to the same match. At each *worker*, we associate a Boolean variable $X_{(u, v)}$ with a virtual node v to indicate whether v matches

Algorithm FPMiner /* executed at the *coordinator* */

Input: Fragmented graph $\mathcal{F} = \{F_1, \cdots, F_n\}$ of data graph G, and support threshold θ.
Output: A *code graph* $\mathcal{G}_c$.
1. initialize a *code graph* $\mathcal{G}_c$ rooted at v_R;
2. collect partial results from *workers*; initialize a set QSet;
3. remove α from QSet if $\mathsf{supp}(Q(\alpha), G) < \theta$;
4. **for each** *code* α in QSet **do** PatExt$(R, \alpha, \theta, \mathcal{G}_c)$;
5. **return** $\mathcal{G}_c$.

Procedure PatExt
Input: α_p, α, θ and $\mathcal{G}_c$.
Output: Updated $\mathcal{G}_c$.
1. **if** $\alpha \neq \mathsf{min}(Q(\alpha))$ **then** ,
2. connect v_α with its parent v_{α_p} in $\mathcal{G}_c$;
3. **return** ;
4. initialize a new node v_α, connect v_α with its parent v_{α_p} in $\mathcal{G}_c$;
5. generate a set cSet of code, corresponding to a set of candidate patterns;
6. **for each** code α_c in cSet **do**
7. M := M∪ LocalMine $(F_i, \theta, \alpha, \alpha_c)$;
8. **for each** α_c in M **do**
9. compute global support of $Q(\alpha_c)$;
10. **if** $\mathsf{supp}(Q(\alpha_c), G) \geq \theta$ **then** PatExt$(\alpha, \alpha_c, \theta, \mathcal{G}_c)$;
11. **return** $\mathcal{G}_c$;

Fig. 4. Algorithm FPMiner.

a pattern node u, and send $X_{(u, v)}$ instead of partial matches taking v as a match of u to neighbouring sites for local evaluation. When v has not been determined, a match of u, $X_{(u, v)}$ is set as unknown, and once v is confirmed as a valid (resp. invalid) match of u, $X_{(u, v)}$ is evaluated as true (resp. false).

Auxiliary structures. The algorithm maintains the following: (a) at the *coordinator*, a *code graph* $\mathcal{G}_c$ in which a node denoted as v_α corresponds to a *code* α and the support of the corresponding pattern $Q(\alpha)$. The *code graph* $\mathcal{G}_c$ differs from *code tree* in that it may contain edges from nodes v_{α_p} to nodes v_α, where α is the minimum *code* of the pattern $Q(\alpha)$ and is generated from α_p; and (b) at each *worker*, indexes $\mathsf{M_F}$, $\mathsf{M_S}$ as a hash-tables, that map *codes* α to match sets and images of $Q(\alpha)$, respectively.

Algorithm. Below, we describe the details of the algorithm.

Initialization. The initialization of FPMiner follows three steps.

(1) The *coordinator* first initializes a *code graph* $\mathcal{G}_c$ with a single node v_R as its root (line 1). It next requests the local support of patterns from all *workers*.

(2) Upon receiving requests from the *coordinator*, all *workers* S_i compute the local support for single edge patterns, in parallel as follows. (a) Each *worker* S_i constructs a graph G_e with a single edge $e = (v, v')$, and generates its *minimum code* $\mathsf{min}(G_e)$. (b) If $\mathsf{min}(G_e)$ equals a code α in $\mathsf{M_F}$, *i.e.,* G_e matches pattern $Q(\alpha)$ with edge (u, u'), and S_i (i) checks whether the edge e is a crossing edge or not. If e is a crossing edge, S_i marks v' in G_e as a "dummy" node, notifies *worker* S_j to generate a match of $Q(\alpha)$ and updates the indexes by sending $X_{(u', v')} = \mathsf{true}$ to S_j, where v' locates; and (ii) updates $\mathsf{M_S}(\alpha)$ by extending $\mathsf{img}(u)$ with $h(u)$ for each pattern node u, where h is the isomorphism mapping from $Q(\alpha)$ to G_e, and expands $\mathsf{M_F}(\alpha)$ with G_e. Note that if a pattern node u is mapped to a "dummy" node v, v is viewed as a "virtual" match of u and is not included in $\mathsf{img}(u)$ at the local fragment; instead, v is included in $\mathsf{M_S}(\alpha)$ at the site where it locates. Finally, S_i generates a set $\mathsf{M} = \{(u, |\mathsf{img}(u)|) | u \in V(\alpha)\}$ for each α in $\mathsf{M_S}$ as the local support and sends M to the *coordinator*.

(3) The *coordinator* S_c assembles sets M from all *workers* and initializes set QSet by summing the local supports (line 2). It next eliminates *code* α from QSet if $Q(\alpha)$ is not frequent, *i.e.,* support is less than the threshold θ (line 3), and repeatedly invokes procedure PatExt to discover larger frequent patterns (line 4).

Example 7. Recall graph G in Fig. 1. After initialization, the index $\mathsf{M_F}$ at S_2 is shown in the table below:

Pattern	Matches	Pattern	Matches
Q_1 = (PM, BA)	{(Walt, Nancy)}	Q_2 = (PM, DBA)	{(Walt, Fred), (Walt, Mat∗)}
Q_3 = (DBA, BA)	{(Fred, Nancy), (Fred, Mary∗)}	Q_4 = (BA, PRG)	{(Nancy, Bill), (Nancy, Tim∗)}
Q_5 = (DBA, PRG)	{(Fred, Bill), (Fred, Tim∗)}	Q_6 = (DBA, ST)	{(Fred, Rei)}
Q_7 = (PRG, ST)	{(Bill, Rei)}		

Note that nodes marked with * are "dummy nodes" and are not considered when computing local support. After assembling M from the *workers*, the *coordinator* finds that the support of Q_1, Q_2, Q_5, Q_6 and Q_7 are all 3, while Q_3 and Q_4 have support 2.

Frequent pattern mining. Starting from a code α, whose corresponding pattern takes a single edge, the *coordinator* invokes procedure PatExt to mine frequent patterns. In summary, the mining procedure consists of three stages. In the first stage, PatExt verifies whether the given code is redundant and expands the *code graph* with the given code (lines 1–4). Second, PatExt generates a set of code that is verified to be minimum (line 5). Last, PatExt invokes procedure LocalMine to compute the support for each candidate code at all sites, in parallel; assembles the results and invokes PatExt to mine larger patterns (lines 6–10). Below, we describe the details of each stage.

(1) *Redundancy elimination.* A pattern may have numerous *codes*, and each of them may generate different patterns that might be generated before from other codes. Expansion from a code that was generated previously may cause unnecessary computations. To avoid this, one can determine whether a code is redundant by applying the rule given by Lemma 1.

Procedure LocalMine /* executed at each working site S_i in parallel */

Input: Fragment F_i, frequency threshold θ, α, α_c.
Output: A set M.
1. **for each** $G_{Q(\alpha)}$ (with id=id) in $\mathsf{M_F}(\alpha)$ as a match of $Q(\alpha)$ **do**
2. generate $G_{Q(\alpha_c)}$ by extending $G_{Q(\alpha)}$ with $e_1 = (v_1, v_1')$;
3. **if** $G_{Q(\alpha_c)}$ is a match of $Q(\alpha_c)$ **then**
4. assign id_c to $G_{Q(\alpha_c)}$; update $G_{Q(\alpha_c)}$, $\mathsf{M_S}(\alpha_c)$, $\mathsf{M_F}(\alpha_c)$;
5. **for each** undetermined virtual node v_v (locating at S_j) in $G_{Q(\alpha_c)}$ **do**
6. set the corresponding variable associated with v_v as true;
7. invoke EvalT(id, α, α_c) at S_j;
8. **if** $G_{Q(\alpha_c)}$ is a possible match of $Q(\alpha_c)$ and v_1' is at site S_j **then**
9. invoke EvalU(id, u_1, v_1, u_2, v_2) at S_j;
10. **if** $X_{(u_1', v_1')}$=true **then**
11. update $G_{Q(\alpha_c)}$, $\mathsf{M_S}(\alpha_c)$, $\mathsf{M_F}(\alpha_c)$;
12. **return** $\mathsf{M} = \{(\alpha_c, \langle u, |\mathsf{S}(u)|\rangle)|u \in V(\alpha_c), \alpha_c \in \mathsf{M_S}\}$;

Fig. 5. Procedure LocalMine.

Lemma 1. *If the code α of a pattern $Q(\alpha)$ is not minimum, then α must be redundant, and any descendant of v_α on the code graph must also be redundant.*

Proof. To see the correctness of the lemma, observe that when α is not minimum, its child α_c on the *code graph* must correspond to a pattern $Q(\alpha_c)$ that is not the same as the pattern generated from $\mathsf{min}(Q(\alpha))$. Indeed, $Q(\alpha_c)$ corresponds to *code* $\mathsf{min}(Q(\alpha_c))$. Thus, all the patterns generated from α can be generated from other minimum *codes*, which shows that α is "redundant". By induction, when α is "redundant", any of its descendants must also be "redundant". □

Lemma 1 tells us that one can determine whether code α is redundant by determining whether α is the *minimum code* of the pattern $Q(\alpha)$. Following the lemma, PatExt applies the strategy, given in Yan and Han (2002), to verify whether a code α is minimum (line 1). The strategy iteratively (a) expands current code α', (b) compares α with α' during expansion, and (c) terminates the current iteration when α is greater than α'.

Note that PatExt uses a *code graph* $\mathcal{G}_c$, rather than a *code tree*, to maintain frequent patterns. As opposed to the growth of *code trees* that only include a new edge between a pair of nodes that correspond to the *minimum codes*, given code α and its parent code α_p, if α is verified to be redundant, $\mathcal{G}_c$ will also expand by inserting an edge from node v_{α_p} to node $v_{\alpha'}$, where α' is the *minimum code* of $Q(\alpha)$, even though pattern extension will no longer proceed (lines 2–3). If α is a minimum code, PatExt expands $\mathcal{G}_c$ with node v_α and edge (v_{α_p}, v_α) (line 4) and starts the next round of pattern extension.

(2) *Candidate generation.* To avoid generating excessive duplicate patterns following a naive pattern extension strategy, the below rule is applied by PatExt. Specifically, given *code* α, (a) the new edge to be inserted in $Q(\alpha)$ either connects nodes on the rightmost path of the DFS tree of $Q(\alpha)$ as backward edges, or grows from the rightmost node of $Q(\alpha)$ as forward edges; and (b) any node label of G, whose appearance frequency is above the support threshold θ, can be used to label new nodes in the updated pattern (line 5).

Example 8. Recall pattern Q_3 in Example 7. Three candidate patterns $Q_{3.1}$, $Q_{3.2}$ and $Q_{3.3}$, shown in Fig. 2, are generated by expanding Q_3 with forward edges (marked with dotted lines), while candidate $Q_{3.1.4}$ extends $Q_{3.1}$ with a backward edge (marked with dotted lines). Moreover, the *code* of $Q_{3.1.2}$: {(BA, DBA), (DBA, PRG), (DBA, ST)} is already a minimum code. When candidate $Q_{3.2.2}$ is generated, it is considered a duplicate pattern because its *minimum code* is the same as that of $Q_{3.1.2}$.

(3) *Support computation.* The support of candidate patterns is computed in parallel with the following three steps.

Step I: Given a set cSet of candidate *codes*, the *coordinator* repeatedly sends α_c in cSet to *workers* to request the local support of $Q(\alpha_c)$ via procedure PatExt (lines 6–7).

Step II: Upon receiving α_c, all *workers* compute matches and *images* of pattern $Q(\alpha_c)$ with procedure LocalMine (Fig. 5), in parallel. Below, we illustrate how procedure LocalMine works at each *worker* in more details.

Each *worker* S_i identifies the edge $e_i = (u_1, u_1')$ that is in $Q(\alpha_c)$ but not in $Q(\alpha)$ and generates a candidate match $G_{Q(\alpha_c)}$ by extending $G_{Q(\alpha)}$ with an edge $e_1 = (v_1, v_1')$, whose endpoints have the same node labels as that of edge e_i for each match $G_{Q(\alpha)}$ of $Q(\alpha)$ (line 2). Note that the set of matches $G_{Q(\alpha)}$ are generated in the last round of computations and are stored in $\mathsf{M_F}$. If $G_{Q(\alpha_c)}$ is verified as a valid match of $Q(\alpha_c)$ (line 3), the *worker* S_i (a) assigns an id id_c to $G_{Q(\alpha_c)}$, (b) marks v_1' in $G_{Q(\alpha_c)}$ as a "dummy" node if v_1' is a virtual node, and (c) updates indexes $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ (line 4). It next propagates the truth value of each virtual node in $G_{Q(\alpha_c)}$ via procedure EvalT (lines 5–7). If $G_{Q(\alpha_c)}$ cannot be determined as a match of $Q(\alpha_c)$, which indicates that $e_i = (u_1, u_1')$ and $e_1 = (v_1, v_1')$ are the backward edge and crossing edge, respectively, then procedure EvalU (not shown) is invoked at S_j, where v_1' is located, to verify whether $G_{Q(\alpha_c)}$ is a true match (lines 8–9). After verification, if a boolean value $X_{(u_1, v_1)} = \mathsf{true}$, indicating that v_1 is a valid match of u_1, is returned, LocalMine updates $G_{Q(\alpha_c)}$, $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ in the same way as it does before (lines 10–11). Finally, LocalMine sends the set $\mathsf{M} = \{(\alpha_c, \langle u, |\mathsf{S}(u)|\rangle)|u \in V(\alpha_c), \alpha_c \in \mathsf{M_S}\}$ to the *coordinator* for support computation (line 12).

Procedure EvalT, shown in Fig. 6, propagates the truth value of variables as follows. Upon receiving *id*, α and α_c, it determines whether there exists a partial match with id=*id* in local index $\mathsf{M_F}(\alpha)$; if there does not exist such a partial match, then a new match $G_{Q(\alpha_c)}$ of $Q(\alpha_c)$ is generated, where all the nodes in $G_{Q(\alpha_c)}$ that is not in the current fragment are marked as "dummy" nodes; otherwise, EvalT simply extends $G_{Q(\alpha)}$ with a new edge corresponding to the pattern edge, which is last, inserted in $Q(\alpha_c)$ (line 1), and updates $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ by including the new match (line 2). It then propagates the truth value (lines 3–5). To be more specific, for each virtual node v_v in $G_{Q(\alpha_c)}$, if the value of the variable X_{v_v} has not been determined as true, EvalT simply sets $X_{v_v} = \mathsf{true}$ and invokes EvalT at the neighbour site where v_v locates for further propagation.

Observe that if pattern $Q(\alpha_c)$ grows from $Q(\alpha)$ with a backward edge (u_1, u_1'), then any candidate $G_{Q(\alpha)}$ should be expanded with

Procedure EvalT
Input: id, α, α_c.
Output: truth.
1. generate a new match of $Q(\alpha_c)$;
2. update $\mathsf{M_S}(\alpha_c)$, $\mathsf{M_F}(\alpha_c)$;
3. **for each** undetermined virtual node v_v (located at S_k) in $G_{Q(\alpha_c)}$ **do**
4. set the corresponding variable associated with v_v as true;
5. invoke EvalT(id, α, α_c) at S_k;
6. **return** truth;

Fig. 6. Procedure EvalT.

the edge $e_b = (h(u_1), h(u'_1))$, where h is the isomorphism mapping from $Q(\alpha)$ to $G_{Q(\alpha)}$. When the edge e_b is a crossing edge connecting v_1 and v'_1, $G_{Q(\alpha_c)}$ may not be determined as a valid match of $Q(\alpha_c)$ with local information, and then, the data need to be shipped to neighbour sites for further verification. Thus, procedure EvalU (not shown) is invoked at the site where virtual node v'_1 is located to determine whether there exists a partial match with id=id and contains v'_1 as a match of u'_1. If so, it next invokes procedure EvalT to update indexes and propagate the truth variables.

Example 9. After receiving a candidate pattern $Q_{3.1}$ from the *coordinator*, the *worker* S_2 computes its local support as follows. It first extends matches of Q_3 and generates two matches $G^1_{3.1}$ and $G^2_{3.1}$ with node set {Nancy, Fred, Bill} and {Nancy, Fred, Tim}, respectively, and updates $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ by including image {(BA, {Nancy}), (DBA, {Fred}), (PRG, {Tim, Bill})} and match set $\{G^1_{3.1}, G^2_{3.1}\}$, respectively. As $G^2_{3.1}$ has a virtual node Tim as a match of PRG, S_2 sets $X_{(\mathtt{PRG,Tim})}$ as true, marks Tim in $G^2_{3.1}$ as a "dummy" node, sends a message with $X_{(\mathtt{PRG,Tim})} = \mathsf{true}$ to S_3, and $\mathsf{M} = \{(\mathtt{BA}, 1), (\mathtt{DBA}, 1), (\mathtt{PRG}, 1)\}$ to the *coordinator*. After receiving messages from S_2, S_3 initializes a new match of $Q_{3.1}$ with node set {∗, Fred∗, Tim}, where * indicates a "dummy" node.

When another candidate pattern $Q_{3.1.4}$ is received, S_3 extends a match of $Q_{3.1}$, which contains nodes {∗, Fred∗, Tim}, with edge (Tim, Nancy). As the match after extension cannot be determined, a match of $Q_{3.1.4}$, S_3 sends message $X_{(\mathtt{BA,Nancy})} = \mathsf{unknown}$ to S_2 for further verification. Upon receiving the message, S_2 determines that $G^2_{3.1}$ with node Nancy as a match of BA can be a match of $Q_{3.1.4}$ after extension because it has node Tim as a match of PRG and generates a new match of $Q_{3.1.4}$ with edges {(Nancy, Fred), (Fred, Tim∗), (Tim∗, Nancy)} and sends $X_{(\mathtt{BA,Nancy})} = \mathsf{true}$ to S_3. S_3 finally generates a match of $Q_{3.1.4}$ with nodes {Nancy∗, Fred∗, Tim} and updates indexes with the new match.

Step III : At the *coordinator*, procedure PatExt receives sets M from all *workers*, assembles them, computes global support for each candidate pattern $Q(\alpha_c)$, and invokes itself to further mine patterns with $Q(\alpha_c)$ that are verified to be frequent. When all the *codes* are processed, PatExt returns *code graph* $\mathcal{G}_c$ as the result (lines 8–11 of PatExt).

Result collection. When all the *codes* corresponding to the single edge patterns are processed, the *code graph* $\mathcal{G}_c$ is built. The *coordinator* then returns $\mathcal{G}_c$ as the final result because each node on the *code graph* corresponds to a frequent pattern (line 5 of FPMiner).

Analyses. To complete the proof of Theorem 1, it remains to verify the correctness and performance of FPMiner.

Correctness. The node set of the *code graph* $\mathcal{G}_c$ corresponds to the complete set of frequent patterns for the following reasons. (1) FPMiner applies PatExt to iteratively mine frequent patterns and expands $\mathcal{G}_c$ with the *codes* corresponding frequent patterns starting from the single edge patterns. (2) During the process, PatExt generates all possible candidate patterns with $n+1$ edges from each frequent n-edge pattern, verifies support of candidates and identifies frequent patterns. This guarantees that no frequent pattern is missed. Moreover, PatExt does not eliminate any non-redundant code ensured by Lemma 1. These together warrant that the node set of $\mathcal{G}_c$ corresponds to the set of frequent patterns.

To analyse the performance of the algorithm, we denote candidate patterns corresponding to k-th level nodes of *code graph* $\mathcal{G}_c$ as $Q_k = (V_k, E_k)$. Then, $|E_k|$ trivially equals k. One may verify the following to see the performance.

Total network traffic. To compute the support for a candidate pattern, two types of data, which are variables exchanged among *workers* and local supports sent to the *coordinator* from all *workers*, need to be shipped. Taking the support computation for pattern Q_k at site S_i as an example, (a) initially, at most $|V_k||F_i.O|$ true variables and $|V_k||cE_i|$ unknown variables are sent from site S_i. In the following, unknown variables and true variables are broadcast among multiple sites. Observe that the value of each unknown variable associated with a virtual node depends on the value of at most $|V_k||F_i.O|$ variables associated with virtual nodes in the same fragment; in addition, the distance between v_i and v_1 must be no longer than $|E_k| - 1$ because only if v_i and v_1 are in the same match of $Q(\alpha_c)$, the unknown variable $X_{(u_i, v_i)}$ can be evaluated to true. Thus, the total number of unknown variables to be shipped is $\Sigma_{i\in[1,|E_k|-1]}|V_k||F_i.O|$, which is bounded by $O(|V_k||V_f|)$, where V_f is the set of all virtual nodes of graph G. As responses to unknown variables, at most, the same number of variables taking truth value are sent. Therefore, the total number of variables that need to be shipped is bounded by $O(|\mathcal{F}||V_k||V_f|)$ in all rounds of communications in the worst case. (b) As the size of set M about local support of pattern Q_k is $|V_k|$, the total size of sets M received by the *coordinator* is bounded by $O(|\mathcal{F}||V_k|)$. Putting (a) and (b) together, FPMiner ships at most $O((k+1)|\mathcal{F}||V_f|)$ data to compute the support for a candidate pattern at k-th level of $\mathcal{G}_c$.

Computational cost. When computing support for a candidate Q_k, the number of matches that need to be expanded is bounded by $O(2^{|V_k|})$. For each crossing edge (v_1, v'_1) marked as unknown at S_i, it takes neighbour site $O(2^{|V_k|}|V_k| + |V_k||F_j.O|)$ time to verify whether v'_1 is a match of u'_1. Moreover, the verification request is propagated within $|E_k| - 1$ steps from S_i. Hence, it takes $O(2^{|V_k|} + |F_i.O| + |cE_i|(2^{|V_k|}|V_k| + |V_k||F_i.O|)^{|E_k|-1})$ time, which is bounded by $O(|E_f|((k+1)2^{k+1})^{k-1})$ time, to compute support for Q_k.

This completes the proof of Theorem 1. □

Remark. (1) After the local support of candidate patterns are evaluated at each *worker*, all variables associated with virtual nodes are recovered to their original value, *i.e.*, unknown. (2) Though the computational complexity of FPMiner is exponential, FPMiner is shown to be rather efficient. As shown in our experimental study, it only takes FPMiner 269 s to mine frequent patterns on a graph with 4 million nodes and 53.5 million edges.

3.3. Optimization strategies

Due to the high computational cost of FPM, it is necessary to develop strategies to optimize the mining process. Observe that

Procedure $\mathsf{LocalMine}_{\mathsf{opt}}$ /* executed at each working site S_i in parallel */

Input: Fragment F_i, frequency threshold θ, α, α_c.
Output: A set M.

1. **for each** $G_{Q(\alpha)}$ in $\mathsf{M_F}(\alpha)$ as a match of $Q(\alpha)$ **do**
2. generate $G_{Q(\alpha_c)}$ by extending $G_{Q(\alpha)}$ with $e_1 = (v_1, v_1')$;
3. **if** $G_{Q(\alpha_c)}$ is a match of $Q(\alpha_c)$ **then**
4. **if** v_1' is a virtual node **then**
5. update $G_{Q(\alpha_c)}$; $X_{(u_1', v_1')} :=$ true;
6. update $\mathsf{M_S}(\alpha_c)$, $\mathsf{M_F}(\alpha_c)$;
7. **if** $G_{Q(\alpha_c)}$ is a possible match of $Q(\alpha_c)$ and v_1' is at site S_j **then**
8. mark e_1 as unknown;
9. **for each** virtual node v_1' with $X_{(u_1', v_1')} =$ true **do**
10. invoke $\mathsf{EvalT}_{\mathsf{opt}}(\mathsf{M_F}(\alpha), id, \alpha, \alpha_c)$ at S_j;
11. **for each** crossing edge $e_1 = (v_1, v_1')$ marked as unknown **do**
12. invoke $\mathsf{EvalU}_{\mathsf{opt}}(\mathsf{M_F}(\alpha), id, u_1, v_1, u_2, v_2)$ at S_j;
13. **if** $X_{(u_1', v_1')}$=true **then**
14. update $G_{Q(\alpha_c)}$, $\mathsf{M_S}(\alpha_c)$, $\mathsf{M_F}(\alpha_c)$;
15. **return** $\mathsf{M} = \{(\alpha_c, \langle u, |\mathsf{S}(u)|\rangle) | u \in V(\alpha_c), \alpha_c \in \mathsf{M_S}\}$;

Fig. 7. Procedure $\mathsf{LocalMine}_{\mathsf{opt}}$.

the MNI based support metric treats the size of the smallest image of pattern nodes as the frequency of the pattern, we hence do not need to enumerate all the matches of candidate patterns; instead, we only need to guarantee that for each pattern node, all its matches can be found during the mining process. Based on this observation, we next introduce an optimization technique that significantly improves procedure LocalMine. To ease the presentation of the procedure, we denote a virtual node v_o (resp. v_i), and a *dominating* (resp. *dependent*) node of v_i (resp. v_o) if the value of variable $X_{(u_i,v_i)}$ depends on the value of variable $X_{(u_o,v_o)}$.

As shown in Fig. 7, procedure $\mathsf{LocalMine}_{\mathsf{opt}}$ opts to invoke remote procedures in a more efficient manner. Specifically, for each match $G_{Q(\alpha)}$ of $Q(\alpha)$, if $G_{Q(\alpha)}$ is expanded with a crossing edge $e_1 = (v_1, v_1')$, $\mathsf{LocalMine}_{\mathsf{opt}}$ either evaluates the boolean variable associated with virtual node v_1' as true if $G_{Q(\alpha_c)}$ is a true match of $Q(\alpha_c)$ (line 5), or marks e_1 as unknown if $G_{Q(\alpha_c)}$ cannot be verified a true match (line 8). After all the matches of $Q(\alpha)$ are processed, $\mathsf{LocalMine}_{\mathsf{opt}}$ invokes $\mathsf{EvalT}_{\mathsf{opt}}$ (not shown) at neighbour site S_j for each virtual node v_1' whose associated variable has been evaluated as true, to propagate variables that have been evaluated as true (lines 9–10). For each crossing edge marked as unknown, $\mathsf{LocalMine}_{\mathsf{opt}}$ invokes $\mathsf{EvalU}_{\mathsf{opt}}$ (not shown) at site S_j where v_1' is located to verify whether $G_{Q(\alpha_c)}$ is a match. If a truth value is returned, $\mathsf{LocalMine}_{\mathsf{opt}}$ updates $G_{Q(\alpha_c)}$, $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ (lines 11–14). Finally, $\mathsf{LocalMine}_{\mathsf{opt}}$ returns the set M as the result (line 15).

Similar to EvalT, procedure $\mathsf{EvalT}_{\mathsf{opt}}$ is also in charge of propagating variables that have been evaluated as true. It works as follows. At site S_i, when a message with $X_{(u,v)} =$ true is received, $\mathsf{EvalT}_{\mathsf{opt}}$ (a) determines whether the dependent node v_i of v can become a match of u_i by determining whether all dominating nodes of v_i are valid matches of pattern nodes and notifies site S_k where v_i is located via message $X_{(u_i,v_i)} =$ true if v_i is determined a valid match of u_i, and (b) determines whether all variables associated with virtual nodes as dependent nodes are evaluated as true, generates $G_{Q(\alpha_c)}$ by expanding $G_{Q(\alpha)}$ using dummy nodes and edges connecting them, and updates indexes $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_N}(\alpha_c)$ as before if true.

Procedure $\mathsf{EvalU}_{\mathsf{opt}}$ works similar to EvalU. At site S_i, upon receiving a message $X_{(u_i,v_i)} =$ unknown from site S_j, $\mathsf{EvalU}_{\mathsf{opt}}$ conducts the following to verify whether v_i is a match of u_i. First, it iteratively verifies whether each match $G_{Q(\alpha)}$ of $Q(\alpha)$, which includes v_i as a match of u_i in $Q(\alpha)$, contains the node v_1 as a match of u_1. If so, $\mathsf{EvalU}_{\mathsf{opt}}$ generates a new match $G_{Q(\alpha_c)}$ by including a new edge $(h(u_1), h(u_1'))$ in $G_{Q(\alpha)}$, updates the indexes as before, and sets $X_{(u_i,v_i)} =$ true; otherwise, $\mathsf{EvalU}_{\mathsf{opt}}$ associates a variable $X_{(u_o,v_o)}$ with each virtual node v_o as a match of u_o in $G_{Q(\alpha)}$. After all matches of $Q(\alpha)$ are processed, $\mathsf{EvalU}_{\mathsf{opt}}$ either sends a message with $X_{(u_i,v_i)} =$ true to S_j indicating that v_i matches u_i if $X_{(u_i,v_i)}$ is evaluated as true, or $X_{(u_o,v_o)} =$ unknown to site S_k, where v_o is located to request further verification.

Remark. The optimization technique effectively improves the performance of FPMiner. As shown in our experimental study; the computational cost and the network traffic are reduced by 30.5% and 32%, respectively, on average, over real-life graphs.

4. GPARs generation

In this section, we first introduce how to generate a set of GPARs from *code graph* $\mathcal{G}_c$. We then introduce the notion of "representative" GPARs and develop a technique to generate "representative" GPARs.

4.1. Rule generation

Conventional association rules are generated based on frequent itemsets. Similarly, we next show how to generate GPARs by using *code graphs*, which maintains all frequent patterns.

Theorem 2. *Given a code graph* $\mathcal{G}_c = (V_c, E_c)$*, it is in* $O(|V_c|(|V_c| + |E_c|))$ *time to generate the set of* GPARs.

In the following, we show Theorem 2 with a constructive proof.

Algorithm. The algorithm, denoted as RuleGen and shown in Fig. 8, takes a *code graph* $\mathcal{G}_c = (V_c, E_c)$ and confidence bound η as input. It first initializes an empty set S and an empty queue q (line 1). In the following, RuleGen repeatedly traverses $\mathcal{G}_c$ from each node v_α via a reverse breadth-first search and produces GPARs (lines 2–9). Specifically, for each ancestor $v_{\alpha''}$ of v_α encountered during the traversal, it generates another pattern Q_r by excluding edges of $Q(\alpha'')$ from Q_α (line 7). If Q_r is connected and the confidence $\frac{\mathsf{supp}(Q(\alpha),G)}{\mathsf{supp}(Q(\alpha''),G)} \geq \eta$, a new GPAR $Q(\alpha'') \Rightarrow Q_r$ is generated and included in set S (lines 8–9). After all the nodes of $\mathcal{G}_c$ are processed, RuleGen returns the set S, which contains all the GPARs as the final result (line 10).

Analyses. To complete the proof of Theorem 2, we show that RuleGen (1) correctly generates the set of GPARs, and (2) is in $O(|V_c|(|V_c| + |E_c|))$ time.

Correctness. One can readily verify the following. (1) Each node v_α of $\mathcal{G}_c$ represents a frequent pattern, and frequent patterns $Q(\alpha'')$

Algorithm RuleGen

Input: A *code graph* $\mathcal{G}_c$, confidence threshold η.
Output: A set of GPARs.

1. set $\mathsf{S} := \emptyset$, queue $\mathsf{q} := \emptyset$;
2. **for each** node v_α in $\mathcal{G}_c$ **do**
3. $\mathsf{q} := \emptyset$; push v_α in q;
4. **while** $\mathsf{q} \neq \emptyset$ **do**
5. node v'_α:=q.pop();
6. **for each** $(v''_\alpha, v'_\alpha) \in E_c$ **do**
7. push v''_α in q; generate Q_r;
8. **if** Q_r is connected and $\frac{\mathsf{supp}(Q(\alpha),G)}{\mathsf{supp}(Q(\alpha''),G)} \geq \eta$ **then**
9. $\mathsf{S} = \mathsf{S} \cup \{(Q(\alpha'') \Rightarrow Q_r)\}$;
10. **return** S;

Fig. 8. Algorithm RuleGen.

that correspond to ancestors $v_{\alpha''}$ of v_α in $\mathcal{G}_c$ must be subsumed by $Q(\alpha)$. Then, treating $Q(\alpha'')$ as Q_l, $Q(\alpha)\backslash Q(\alpha'')$ as Q_r, one can generate a valid rule $Q(\alpha'')\Rightarrow Q_r$, if Q_r is connected and the confidence of the rule is no less than η. (2) For any node v_α of $\mathcal{G}_c$, only patterns represented by ancestors of v_α can be used to generate GPARs $Q(\alpha)$, since the pattern represented by any node that is not an ancestor of v_α cannot be subsumed by $Q(\alpha)$. Hence, no rule of $Q(\alpha)$ will be missed after traversal from v_α.

Complexity. For complexity, it can be easily verified that the number of nodes in $\mathcal{G}_c$ is $|V_c|$, and each round of breadth-first search takes at most $|V_c| + |E_c|$ time; hence, the algorithm RuleGen is in $O(|V_c|(|V_c| + |E_c|))$ time.

These complete the proof of Theorem 2 □

Example 10. Recall Example 9. After *code graph* $\mathcal{G}_c$ is constructed, RuleGen generates GPARs by traversing $\mathcal{G}_c$ from each node. Take $\eta = 0.5$ as an example, and let the traversal start from the node that corresponds to pattern $Q_{3.1.4}$ (with edges {(BA, DBA), (DBA, PRG), (PRG, BA)}). During the traversal, the node's parent, which corresponds to $Q_{3.1}$ with edges {(BA, DBA), (DBA, PRG)} and support 2 is first encountered; then, a GPAR with antecedent (with edges {(BA, DBA), (DBA, PRG)}) and consequent (with edge {(PRG, BA)}) can be generated, the confidence of the GPAR is 1, and it is, therefore, a valid GPAR.

4.2. Representative GPARs

It is known that when all frequent itemsets are used, excessive association rules may be generated, which hinders users' inspection and understanding (Zaki, 2000). Therefore, people propose the notion of *maximal frequent itemsets*, treat them as "representative" itemsets, and develop techniques to generate rules with them (Han, Cheng, Xin, & Yan, 2007; Huan et al., 2004). Similarly, (1) the set of maximal frequent patterns can be orders of magnitude smaller than the complete set of frequent patterns, and (2) non-maximal frequent patterns can be reconstructed from the maximal frequent patterns; then, fewer rules are generated from maximal frequent patterns that are also very "representative".

Motivated by this, we study how to generate GPARs using *maximal frequent patterns*. Before proceeding, we first formalize the *maximal frequent patterns mining* (MFPM) problem.

Problem. The MFPM problem is stated as follows.

- *Input*: A graph G, and support threshold θ.
- *Output*: A set S of frequent patterns Q such that (a) $\mathsf{supp}(Q, G) \geq \theta$ for any Q in S, and (b) there does not exist any other frequent pattern Q' in S with $Q' \sqsubseteq Q$.

One may easily verify that the MFPM problem is also NP-hard because it is at least as hard as the FPM problem. Despite hardness, MFPM does not make the situation harder because after *code graph* $\mathcal{G}_c$ is generated, *maximal frequent patterns* can be easily extracted from it.

Proposition 2. *All the leaf nodes of the code graph* $\mathcal{G}_c$ *correspond to maximal frequent patterns.*

Proof. Each leaf node v_α of $\mathcal{G}_c$ corresponds to a code α that is grown from another code α', whose corresponding node $v_{\alpha'}$ in $\mathcal{G}_c$ is the parent of v_α. Inductively, all the *codes* that correspond to the ancestors of v_α on $\mathcal{G}_c$ are subgraphs of $Q(\alpha)$, indicating that α corresponds to a *maximal frequent pattern*. □

Following Proposition 2, we next provide an algorithm to generate "representative" GPARs with *maximal frequent patterns*.

Algorithm. The algorithm, denoted as RepRuleGen (not shown), differs from RuleGen in that it traverses the *code graph* $\mathcal{G}_c$ from each leaf node v_{α_l}, and generates GPARs with Q_{α_l} and $Q_{\alpha''}$, where $v_{\alpha''}$ is an ancestor of v_{α_l}.

Correctness & complexity. The correctness of RepRuleGen is warranted by Proposition 2. To see the complexity of RepRuleGen, observe that RepRuleGen is the same as RuleGen except for the traversing times. As the number of leaf nodes is at most $|V_c|$, it is in $O(|V_c|(|V_c| + |E_c|))$ time, having the same computational complexity as RuleGen.

Remark. Our experimental study shows that generating GPARs with *maximal frequent patterns* substantially reduces the number of GPARs. Indeed, "representative" GPARs only accounts for 1.05% of the whole set of GPARs, on a graph with 4 million nodes and 53.5 million edges.

5. Experimental study

We next present an experimental study of our algorithms. Using real-life and synthetic data, we conducted two sets of experiments to evaluate: (1) The efficiency and data shipment of our distributed algorithm FPMiner and the effectiveness of optimization techniques for FPMiner; (2) the efficiency of the rule generation algorithm RuleGen and the effectiveness of our "representative" rule generation algorithm RepRuleGen.

Experimental setting. We used three real-life graphs. (a) *Amazon* (Leskovec & Krevl, 2014), a product co-purchasing network with 0.55 million nodes and 1.79 million edges. The total size of *Amazon* is 0.95GB. (b) *Pokec* (social network, 2012), a social network with 1.63 million nodes, and 30.6 million edges. Its size is 2.2GB. (c) *Google+* (Gong et al., 2012), a social graph whose size is 2.6GB, has 4 million entities and 53.5 million links.

We also designed a generator to produce synthetic graphs $G = (V, E, L)$, controlled by the numbers of nodes $|V|$ and number of edges $|E|$, where L is taken from an alphabet Σ of $1K$ labels. We generated synthetic graphs following the evolution model (Garg, Gupta, Carlsson, & Mahanti, 2009): An edge was attached to the high degree nodes with higher probability.

The size of G is up to 20 million nodes and 100 million edges.

Algorithms. We implemented the following, all in Java. (1) Algorithm FPMiner, compared with (a) GRAMI$_{\mathsf{ND}}$, which is a naive distributed algorithm, that ships all fragments to the *coordinator*, and applies centralized FPM tool GRAMI (GraMi, 2015); (b) GRAMI$_{\mathsf{D}}$, another distributed FPM algorithm that works as follows. GRAMI$_{\mathsf{D}}$ first requests each *worker* to compute support of a single-node or single-edge patterns, in parallel. After assembling the results from the *workers*, the *coordinator* sends patterns that take a single node or single edge, and have support below threshold θ back to the *workers*. All *workers* then eliminate infrequent nodes and edges that belong to the local fragment and ship the local fragment to the *coordinator*. Upon receiving a

response, the *coordinator* merges all local fragments together and invokes GRAMI (Elseidy et al., 2014) to compute the supports in a centralized way; and (c) $\text{FPMiner}_{\text{opt}}$, which is an optimized version of FPMiner and employs procedures $\text{LocalMine}_{\text{opt}}$, $\text{EvalT}_{\text{opt}}$ and $\text{EvalU}_{\text{opt}}$. (2) Algorithm RuleGen, compared with RepRuleGen, its counterpart for generating "representative" GPARs. (3) Algorithms Jaccard (Jaccard, 1901) and SimRank (Jeh & Widom, 2002) for link prediction. Here, both Jaccard and SimRank are similarity-based methods (Lü & Zhou, 2011), while they apply different similarity metrics. Given two nodes v_1, v_2, Jaccard defines their similarity by $\text{Sim}_j(v_1, v_2) = \frac{N(v_1) \cap N(V_2)}{N(v_1) \cup N(V_2)}$, where $N(v)$ indicates the set of neighbours of v; while SimRank defines the similarity of v_1 and v_2 by $\text{Sim}_s(v_1, v_2) = C \cdot \frac{\Sigma_{v'_1 \in N(v_1)} \Sigma_{v'_2 \in N(v_2)} \text{Sim}_s(v'_1, v'_2)}{k_{v_1} \cdot k_{v_2}}$, where $C \in [0, 1]$ is the decay factor, and k_v is the degree of node v.

Graph fragmentation and distribution. We used the algorithm of Rahimian, Payberah, Girdzijauskas, Jelasity, and Haridi (2013) to partition graph G into n fragments, and distribute them to n sites ($n \in [1, 20]$). Each site is powered by an 8 core Intel(R) Xeon(R) 2.00 GHz CPU with 128GB of memory and 1 TB hard disk, using a Debian Linux 3.2.04 system. Each experiment is run 5 times, and the average is reported. We report the response time (RT) and data shipment (DS) of the algorithms. As GRAMI_{ND} always ships the entire G, we do not show its DS.

Experimental results. We next report our findings.

Exp-1: Performance of FPMiner. In this set of experiments, we evaluated the performance of algorithm FPMiner compared with $\text{FPMiner}_{\text{opt}}$, GRAMI_{ND} and GRAMI_{D}. We used the *logarithmic scale* for the y-axis in the figures for RT (response time). We started the tests with three real-life graphs.

Varying n. Fixing $\theta = 0.3K$, $3K$ and $0.8K$ for *Amazon, Pokec* and *Google+*, respectively, we varied n from 4 to 20 and evaluated efficiency and data shipment of FPMiner vs. $\text{FPMiner}_{\text{opt}}$, GRAMI_{D} and GRAMI_{ND}.

Fig. 9 (a)–(c) report the RT of all the algorithms on *Amazon, Pokec* and *Google+*, respectively, which shows the following: (1) the more sites (processors)that are available, the less time FPMiner and $\text{FPMiner}_{\text{opt}}$ take because the parallel computation that our algorithms apply effectively improves performance. Take FPMiner as an example; it is, on average, 4.1, 2.9 and 2.5 times faster when n increases from 4 to 20 on *Google+*, *Pokec* and *Amazon*, respectively. Moreover, FPMiner requires only 269 s to identify frequent patterns over *Google+*, which is distributed over 20 sites. (2) $\text{FPMiner}_{\text{opt}}$ performs better than FPMiner, *e.g.,* it is on average 1.43, 1.39 and 1.49 times faster than FPMiner on *Amazon, Pokec* and *Google+*, respectively, and scales best among all the algorithms, which verifies the effectiveness of our optimization strategy. (3) GRAMI_{ND} and GRAMI_{D} are indifferent to n because they ship local fragments from each *worker* to the *coordinator* and apply a centralized algorithm to identify the matches. As GRAMI_{D} reduces the size of the local fragment at each *worker*, it is, on average, 1.74 times faster than GRAMI_{ND}.

Fig. 9 (d)–(f) show the results on the DS (data shipment) of the algorithms over *Amazon, Pokec* and *Google+*, respectively. We find that (1) FPMiner and $\text{FPMiner}_{\text{opt}}$ ship 13.3% and 8.9% (resp. 22% and 15.2%, 23.4% and 15.9%) data of GRAMI_{D}, on *Amazon* (resp. *Pockec* and *Google+*) when n=20; (2) $\text{FPMiner}_{\text{opt}}$ ships, on average, 66%, 68% and 70% data of FPMiner on *Amazon, Pokec* and *Google+*, respectively, which confirms the effectiveness of the optimization strategy; and (3) GRAMI_{D} ships 81%, 74% and 68% of the size $|G|$ of *Amazon, Pokec* and *Google+*, respectively, since it removes redundant nodes and edges.

Varying θ. Fixing $n = 4$, we varied support θ from $0.1K$ to $0.5K$ in $0.1K$ increments, $2K$ to $4K$ in $0.5K$ increments and $0.6K$ to $1.0K$ in $0.1K$ increments on *Amazon, Pokec* and *Google+*, respectively.

Fig. 9 (g)–(i) show results on RT and demonstrate the following. (1) All algorithms take longer with a small θ because more candidate patterns and their matches need to be verified. (2) $\text{FPMiner}_{\text{opt}}$ performs better than FPMiner, with only 69% of the time of that of FPMiner on average, as unnecessary computations are avoided owing to the strategy $\text{FPMiner}_{\text{opt}}$ applied. (3) FPMiner (resp. $\text{FPMiner}_{\text{opt}}$) outperforms GRAMI_{D} and GRAMI_{ND} in all cases and is less sensitive to the increment of θ because FPMiner (resp. $\text{FPMiner}_{\text{opt}}$) maximizes parallelism during support computation, while GRAMI_{D} and GRAMI_{ND} assemble all the fragments and verify support with the costly centralized method.

The results given in Fig. 9(j)–(l) show the results of DS over *Amazon, Pokec* and *Google+*, respectively. We find that (1) FPMiner, $\text{FPMiner}_{\text{opt}}$ and GRAMI_{D} ship less data when θ increases, as expected; (2) $\text{FPMiner}_{\text{opt}}$ is less sensitive to θ than FPMiner; and (3) GRAMI_{ND} incurs 178% and 298% more DS than FPMiner and $\text{FPMiner}_{\text{opt}}$, on average.

Varying |G| (Synthetic). Fixing $n = 4$, and $\theta = 1K$, we varied $|G|$ from $(10M, 20M)$ to $(50M, 100M)$ with $10M$ and $20M$ increments on $|V|$ and $|E|$. As shown in Fig. 10(a) and (b), (1) all the algorithms take longer time and ship more data on larger graphs, as expected; and (2) $\text{FPMiner}_{\text{opt}}$ is less sensitive to $|G|$ than others, *w.r.t.* RT and DS.

Exp-2: Performance of RuleGen. In this set of tests, we first evaluate the efficiency and effectiveness of RuleGen vs. RepRuleGen. We then show typical GPARs via a case study.

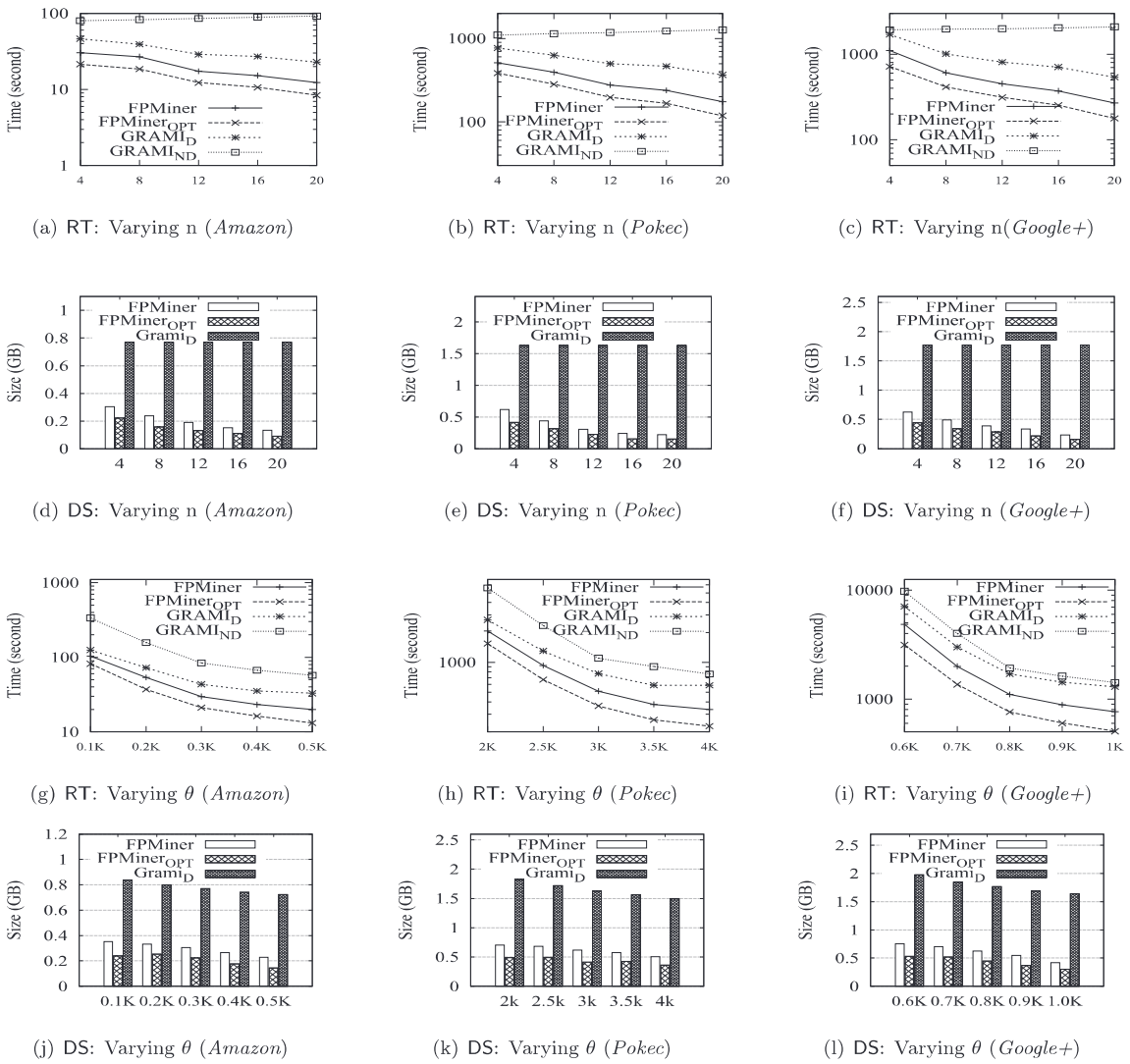
Efficiency. Fixing $\eta = 0.6$, we varied θ from $0.1K$ to $0.5K$ in $0.1K$ increments, $2K$ to $4K$ in $0.5K$ increments, and $0.6K$ to $1.0K$ in $0.1K$ increments on *Amazon, Pokec* and *Google+*, respectively, and evaluated the efficiency of RuleGen and RepRuleGen. Figs. 10(c)–(e) show results on *Amazon, Pokec* and *Google+*, respectively, and tell us the following. (1) It takes more time for both algorithms when θ is smaller because the smaller θ is, the larger *code graph* $\mathcal{G}_c$ is; hence, more time is needed for traversal. (2) RepRuleGen is more efficient than RuleGen. On average, RepRuleGen takes only 35%, 43% and 39% of the time of RuleGen on *Amazon, Pokec* and *Google+*, respectively.

Effectiveness. Following the same setting as the efficiency evaluation of RuleGen and RepRuleGen, we also compared the size of GPARs sets generated by two algorithms. As shown in Fig. 10(f), the rule sets generated by RepRuleGen only accounts for 2.26%, 1.41% and 1.05% of rule sets produced by RuleGen, on *Amazon, Pokec* and *Google+* with $\theta = 0.3K$, $3K$ and $0.8K$, respectively.

To further verify the "representative" property of GPARs generated by RepRuleGen, we evaluate the precision of RepRuleGen *w.r.t.* top-k GPARs as follows. (1) We mine and select top-k GPARs from graphs; and (2) evaluate precision $\text{prec} = \frac{|S_R \cap S_K|}{|S_K|}$, where S_K and S_R indicate top$-k$ GPARs, and GPARs identified by RepRuleGen, respectively.

To do this, metrics that are used to rank the "goodness" of GPARs are needed. We, therefore, choose two typical metrics that are used to measure the interestingness of association rules: *i.e.,* Gain (with $\theta = 0.4$) (Jr. & Agrawal, 1999) and Interest (Brin, Motwani, & Silverstein, 1997) to measure the interestingness of GPARs. The results tell us that (1) the average prec is above 90%, on three real-life graphs, regardless of which metrics are used to identify S_K; and (2) when k grows larger, the precision prec also increases. Due to space constraints, we no longer report detailed results.

Case study. We also manually examined GPARs mined from *Amazon* and *Pokec*. Fig. 10(g) shows three typical GPARs discovered from *Amazon*. Specifically, (1) R_1 shows that if *children's* books b_1 and b_2, and *children's* book b_1 and *literature & fiction* book b_3 are co-purchased books, respectively, then b_1 and *religion & spirituality* book b_4 are also likely to be bought together; and (2) R_2 states that if *children's* books b_1 and b_2 (resp. *children's* book b_2 and *literature & fiction* book b_3) are bought simultaneously, then *business &*



(a) RT: Varying n (*Amazon*) (b) RT: Varying n (*Pokec*) (c) RT: Varying n(*Google+*)

(d) DS: Varying n (*Amazon*) (e) DS: Varying n (*Pokec*) (f) DS: Varying n (*Google+*)

(g) RT: Varying θ (*Amazon*) (h) RT: Varying θ (*Pokec*) (i) RT: Varying θ (*Google+*)

(j) DS: Varying θ (*Amazon*) (k) DS: Varying θ (*Pokec*) (l) DS: Varying θ (*Google+*)

Fig. 9. Performance of FPMiner on real-life graphs.

investing book b_4 will likely be bought when one purchases book b_1; and (3) R_3 states that if *children's* book b_1 and *literature & fiction* book b_2 (resp. *business & investing* book b_3) are viewed together, then b_2 and b_3 may be purchased simultaneously. These rules reveal associations among books that are purchased simultaneously and can be used for book recommendation.

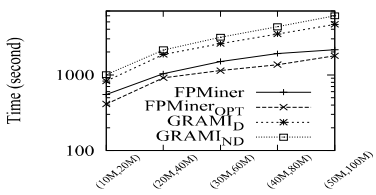
Three typical GPARs on *Pokec*, which are shown in Fig. 10(h), tell us the following. (1) R_4 says that if user u_1 likes *listening to music* and has a friend u_2 with hobby *watching movies*, and one friend u_3 of u_2 likes to *make friends*, then u_1 is likely to have two other friends u_4 and u_5, who like *listening to music* and *watching movies*; and (2) R_5 states that if users u_1, u_2 and u_3 all like *sports*, u_4 likes *swimming*, and u_1 and u_2, u_1 and u_3, u_2 and u_4 are mutual friends, then u_1 is likely to have another friend u_5 with hobby *swimming*; and (3) R_6 shows that if users u_1, u_2 and u_3 all like *sleeping*, and u_1 and u_2, u_1 and u_3 are mutual friends, then u_1 is likely to have one friend with hobby *watching movies*. These rules show that one may recommend friends to people who share similar hobbies.

Exp-3: Prediction accuracy. In this set of tests, we evaluated the prediction accuracy of GPARs vs. Jaccard, SimRank. The settings of the evaluation are as follows.

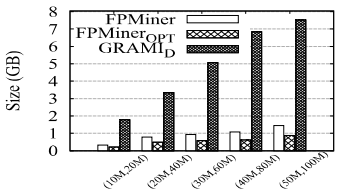
The prediction accuracy of GPARs is tested via cross-validation (Galárraga et al., 2013). That is, given a graph G, we partition it into two fragments F_1 and F_2, mine and select the top 10, 20, 30, 40 and 50 GPARs from F_1, and evaluate the prediction accuracy, which is defined as $\mathrm{Acc}(R) = \frac{\mathrm{supp}(Q_R, F_2)}{\mathrm{supp}(Q_l, F_2)}$, for each GPAR R in F_2. We used metrics Gain and Interest to rank the GPARs, where θ is set as 0.4 for Gain.

The tests of Jaccard, SimRank are conducted via 4-fold cross-validation. Specifically, (1) we extract subgraphs G_F with size (5,000,18,255), (5,000,21,241), (5,000,22,725) from *Amazon*, *Pokec*, and *Google+*, respectively, because it is not applicable for SimRank to run over entire graphs; (2) for each G_F, we randomly divide its edge set into four parts; each time, one part is selected as the testing set, and the remaining 3 parts are chosen to construct a graph as the training graph. The cross-validation process is repeated 4 times, with each of the 4 subsets used exactly once as the testing set; (3) each time, we run Jaccard and SimRank on the training graph; obtain a list of node pairs ranked with their similarity values; select top-L node pairs as node pairs of predicted edges and verify precision $\mathrm{Acc} = \frac{L_r}{L}$, where L_r is the number of correct edges predicted; (4) for SimRank, the parameter C (see Sim_s) is set as 0.8, and the iteration time is set as 5. The average results are reported.

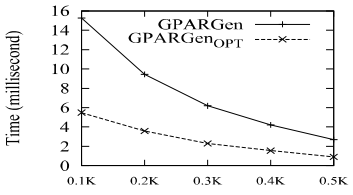
Table 1 shows the prediction accuracy of GPARs, compared with Jaccard and SimRank. We find that (1) using the top-50 GPARs, missing relationships can be predicted with average



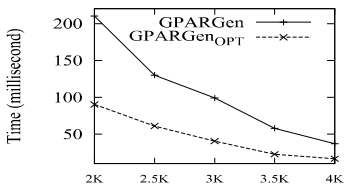
(a) Varying $|G|$ (Synthetic)



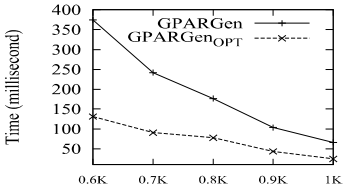
(b) Varying $|G|$ (Synthetic)



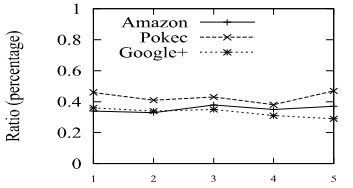
(c) Varying θ ($Amazon$)



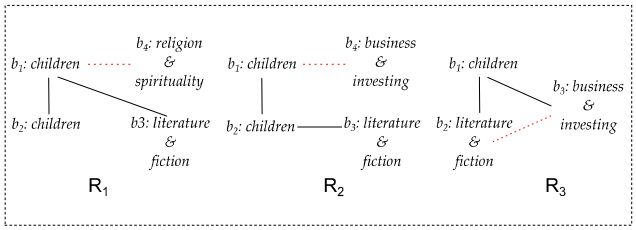
(d) Varying θ ($Pokec$)



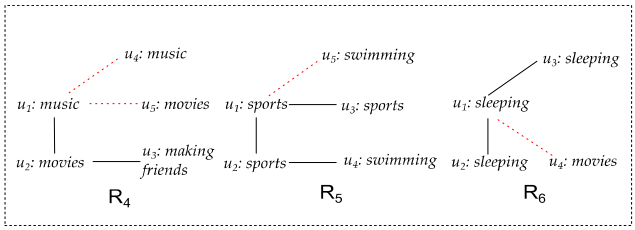
(e) Varying θ ($Google+$)



(f) Varying θ (Real life)



(g) real-life GPARs: $Amazon$



(h) real-life GPARs: $Pokec$

Fig. 10. Scalability of FPMiner, and performance of RuleGen.

Table 1
Prediction accuracy of GPARs vs. Jaccard and SimRank.

		Amazon					*Pokec*					*Google+*				
		top-k (Acc(R), %)					top-k (Acc(R), %)					top-k (Acc(R), %)				
		10	20	30	40	50	10	20	30	40	50	10	20	30	40	50
GPARs	Gain	44.7	44.6	43.8	43.2	38.1	49.8	48.7	48.2	45.1	43.8	42.3	41.6	41.3	39.7	38.8
	Interest	46	44.6	43.7	42.3	38.1	49.4	49	48.4	46.6	44.1	42.5	42.2	41.7	40.9	40.3
		top-L (Acc, %)					top-L (Acc, %)					top-L (Acc, %)				
		10	50	100	150	200	10	50	100	150	200	10	50	100	150	200
Jaccard		32.5	33.2	37.1	46.2	50.3	17.5	14.5	17.2	15.2	13.3	25.4	27.2	23.7	29.6	30.2
SimRank		20.4	13.5	20.1	20.3	19.3	17.5	7.2	9.5	10.3	10.4	18.8	14.3	10.5	12.7	13.3

accuracy up to 40.2% over three real-life graphs, and the higher k is, the lower Acc(R) is, because fewer "interest" GPARs may lead to lower prediction accuracy; (2) our approach outperforms Jaccard and SimRank in terms of prediction accuracy. Taking *Pokec* as an example, when measurement Interest is used, and k is set as 50, the prediction accuracy of GPARs (*i.e.*, 44.1%) is 331.58% and 424.04% of that of Jaccard and SimRank ($L = 200$), respectively. It should be noted that (a) GPARs are used to predict pattern graphs, which may take more than one edge, while both Jaccard and SimRank can only predict single edges. Although the comparison is not very fair, we can still see that GPARs achieves higher overall prediction accuracy; (b) we investigated the prediction process of Jaccard and SimRank and found that the low prediction accuracy of Jaccard and SimRank are partially caused by the sparse characteristics of the graphs used; (c) owing to the parallel computation, our technique is far more efficient than Jaccard and SimRank, *e.g.,* it takes SimRank 1234.7 s, 1830.5 s and 2036.3 s to predict links over a small fraction of *Amazon, Pokec*, and *Google+*, respectively. These further show advantages in terms of the efficiency and accuracy of our technique.

Summary. We find the following from extensive experimental studies.

(1) Our distributed frequent pattern mining algorithms FPMiner and FPMiner$_{opt}$ are not only very efficient but also with low data shipment. This advantage is mainly due to the parallel computation that our algorithms apply. Take *Google+*, which includes 4 million nodes and 53.5 million edges, as an example, FPMiner (resp. FPMiner$_{opt}$) spends fewer than 5 (resp. 3) minutes and ships only 0.234 GB (resp. 0.159 GB) data to discover frequent patterns when the graph is distributed to 20 sites.
(2) Our algorithms FPMiner and FPMiner$_{opt}$ scale well with the number of sites n, support threshold θ and size of graph $|G|$. For example, on *Google+*, FPMiner is 4.1 (resp. 6.3) times faster, when n increases from 4 to 20 (resp. when θ rises from 0.6K to 1.0K); on synthetic graphs G, the time used by FPMiner increases only 3.89 times when $|G|$ increases from (10M, 20M) to (50M, 100M). Because FPMiner$_{opt}$ has

the same trend as FPMiner, we do not report its performance here.

(3) Our optimization technique for FPMiner is quite effective: $\text{FPMiner}_{\text{opt}}$ is, on average, 1.44 times faster than FPMiner and scales better than FPMiner with the increase in the number of sites and support threshold θ, on real-life graphs. This improvement is mainly caused by the smart strategy applied by $\text{FPMiner}_{\text{opt}}$ for match verification.

(4) Our rule generation algorithms RuleGen and RepRuleGen are very efficient, *e.g.,* with no more than 1 second in all cases. Moreover, representative GPARs, generated by RepRuleGen, are indeed "representative", since they only account for, on average, 1.57% of the entire rule set and cover more than 90% of top-k GPARs on real-life graphs, which verifies the effectiveness of RepRuleGen and may facilitate inspection and interpretation of GPARs.

(5) The GPARs can predict missing relationships with an average accuracy of 43.8% on real-life graphs. Compared with existing link prediction methods, Jaccard and SimRank, our method is much more efficient and obtains higher prediction accuracy, *e.g.,* 331.58% and 424.04%, of that of Jaccard and SimRank, respectively, on *Pokec*.

6. Conclusion

We have proposed generalized graph-pattern association rules (GPARs) and viable support and confidence measures for the discovery of GPARs. Compared with special rules introduced in Fan et al. (2015), the generalized GPARs are capable to model even more complicated associations among social entities. We have provided techniques to efficiently mine GPARs. In particular, our technique for frequent pattern mining follows the "look-ahead & backtracking" strategy, that is widely employed for the constraint satisfaction problems. Our experimental study has verified the efficiency and effectiveness of the algorithms. These results extend the study of GPARs. We hence contend that GPARs yield a promising tool for social network analysis.

The study of GPARs is still in its infancy. One topic for future work is to extend GPARs by incorporating different semantics of graph pattern matching such as graph simulation. Another topic concerns GPARs with quantifier and their mining techniques. The third topic is to develop metrics to measure distance of GPARs and efficiently identify diversified top-k GPARs based on the distance metric. Finally, how to efficiently maintain GPARs mined from frequently updated graphs needs further exploration.

Author statement

Xin Wang conceived of the presented idea, developed the theory and algorithms. Yang Xu verified the methods, and implemented algorithms with the help of Xin Wang. Yang Xu and Huayi Zhan designed experiments, and conducted tests. Xin Wang wrote the manuscript with support fromYang Xu and Huayi Zhan.

Xin Wang agreed to be accountable for all aspects of the work in ensuring that problems appearing in any part of the work are appropriately investigated and resolved.

Declaration of Competing Interest

The authors whose names are listed immediately below certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

The authors whose names are listed immediately below report the following details of affiliation or involvement in an organization or entity with a financial or non-financial interest in the subject matter or materials discussed in this manuscript. Please specify the nature of the conflict on a separate sheet of paper if the space below is inadequate.

Acknowledgments

This work is supported in part by the NSFC 71490722, 71490725, 71671146, Sichuan Provincial Major Project 2017JY0225 and Fundamental Research Funds for the Central Universities, China.

References

Agrawal, R., Imielinski, T., & Swami, A. N. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on management of data* (pp. 207–216).

Aridhi, S., & Nguifo, E. M. (2016). Big graph mining: Frameworks and techniques. *Big Data Research, 6*, 1–10.

Brin, S., Motwani, R., & Silverstein, C. (1997). Beyond market baskets: Generalizing association rules to correlations. In *Proceedings ACM SIGMOD international conference on management of data* (pp. 265–276).

Chaturvedi, A., Tiwari, A., & Spyratos, N. (2019). minstab: Stable network evolution rule mining for system changeability analysis. *IEEE Transactions on Emerging Topics in Computational Intelligence*.

Cheng, Y., Chen, L., Yuan, Y., & Wang, G. (2018). Rule-based graph repairing: Semantic and efficient repairing methods. In *2018 ieee 34th international conference on data engineering (icde)* (pp. 773–784). IEEE.

Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI, 26*(10), 1367–1372.

Ebisu, T., & Ichise, R. (2019). Graph pattern entity ranking model for knowledge graph completion. In *Proceedings of the 2019 conference of the north american chapter of the association for computational linguistics: Human language technologies, NAACL-HLT 2019, volume 1* (pp. 988–997).

Elseidy, M., Abdelhamid, E., Skiadopoulos, S., & Kalnis, P. (2014). GRAMI: Frequent subgraph and pattern mining in a single large graph. *PVLDB, 7*(7), 517–528.

Fan, W., Wang, X., Wu, Y., & Xu, J. (2015). Association rules with graph patterns. *PVLDB, 8*(12), 1502–1513.

Fan, W., Wu, Y., & Xu, J. (2016). Adding counting quantifiers to graph patterns. In *Proceedings of the 2016 international conference on management of data, SIGMOD conference* (pp. 1215–1230).

Fiedler, M., & Borgelt, C. (2007). Subgraph support in a single large graph. In *Workshops proceedings of the 7th IEEE international conference on data mining (ICDM)* (pp. 399–404).

Galárraga, L. A., Teflioudi, C., Hose, K., & Suchanek, F. M. (2013). AMIE: Association rule mining under incomplete evidence in ontological knowledge bases. In *22nd international world wide web conference, WWW* (pp. 413–422).

Garg, S., Gupta, T., Carlsson, N., & Mahanti, A. (2009). Evolution of an online social aggregation network: An empirical study. In *Proceedings of the 9th ACM SIGCOMM internet measurement conference, IMC* (pp. 315–321).

Gong, N. Z., Xu, W., Huang, L., Mittal, P., Stefanov, E., Sekar, V., & Song, D. (2012). Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *Proceedings of the 12th ACM SIGCOMM internet measurement conference, IMC* (pp. 131–144).

Gouda, K., & Zaki, M. J. (2001). Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE international conference on data mining* (pp. 163–170).

GraMi (2015). https://github.com/ehab-abdelhamid/GraMi.

Grujic, I., Bogdanovic-Dinic, S., & Stoimenov, L. (2014). Collecting and analyzing data from e-government facebook pages. *ICT Innovations*, 86–96.

Gudes, E., Shimony, S. E., & Vanetik, N. (2006). Discovering frequent graph patterns using disjoint paths. *IEEE Transactions of Knowledge Data Engineering, 18*(11), 1441–1456.

Halder, S., Samiullah, M., & Lee, Y.-K. (2017). Supergraph based periodic pattern mining in dynamic social networks. *Expert Systems with Applications, 72*, 430–442.

Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: Current status and future directions. *Data Mining Knowledge Discovery, 15*(1), 55–86.

Huan, J., Wang, W., Prins, J. F., & Yang, J. (2004). SPIN: Mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 581–586).

Husain, M. F., Doshi, P., Khan, L., & Thuraisingham, B. M. (2009). Storage and retrieval of large RDF graph using hadoop and MapReduce. In *Cloudcom* (pp. 680–686).

Inokuchi, A., Washio, T., & Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of data mining and knowledge discovery* (pp. 13–23).

de J. Costa, J., Bernardini, F., Artigas, D., & Viterbo, J. (2019). Mining direct acyclic graphs to find frequent substructures - an experimental analysis on educational data. *Information Science, 482*, 266–278.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat, 37*, 547–579.

Jeh, G., & Widom, J. (2002). Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 538–543).

Jiang, C., Coenen, F., & Zito, M. (2013). A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review, 28*(01), 75–105.

Jr. , R. J. B., & Agrawal, R. (1999). Mining the most interesting rules. In *Proceedings of the fifth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 145–154).

Leskovec, J., & Krevl, A. (2014). Amazon dataset. http://snap.stanford.edu/data/index.html.

Lin, P., Song, Q., Shen, J., & Wu, Y. (2018). Discovering graph patterns for fact checking in knowledge graphs. In *International conference on database systems for advanced applications* (pp. 783–801). Springer.

Lü, L., & Zhou, T. (2011). Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications, 390*(6), 1150–1170.

Namaki, M. H., Wu, Y., Song, Q., Lin, P., & Ge, T. (2017). Discovering graph temporal association rules. In *Proceedings of the 2017 ACM on conference on information and knowledge management, CIKM* (pp. 1697–1706).

Rahimian, F., Payberah, A. H., Girdzijauskas, S., Jelasity, M., & Haridi, S. (2013). JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *7th IEEE international conference on self-adaptive and self-organizing systems, SASO* (pp. 51–60).

Rowe, M. (2009). Interlinking distributed social graphs. In *Proceedings of the WWW workshop on linked data on the web, LDOW*.

Sapountzi, A., & Psannis, K. E. (2018). Social networking data analysis tools & challenges. *Future Generation Comp System, 86*, 893–913.

social network, P. (2012). http://snap.stanford.edu/data/soc-pokec.html.

Schmitz, C., Hotho, A., Jäschke, R., & Stumme, G. (2006). Mining association rules in folksonomies. In *Data science and classification* (pp. 261–270).

Talukder, N., & Zaki, M. J. (2016). A distributed approach for graph mining in massive networks. *Data Mining Knowledge Discovery, 30*(5), 1024–1052.

Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., & Aboulnaga, A. (2015). Arabesque: a system for distributed graph mining. In *Proceedings of the 25th symposium on operating systems principles, SOSP* (pp. 425–440).

Wang, X., & Xu, Y. (2018). Mining graph pattern association rules. In *Database and expert systems applications - 29th international conference, DEXA* (pp. 223–235).

Yan, X., & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE international conference on data mining (ICDM)* (pp. 721–724).

Zaki, M. J. (2000). Generating non-redundant association rules. In *Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 34–43).

Zhang, C., & Zhang, S. (2002). Association Rule Mining, Models and Algorithms. *Lecture Notes in Computer Science*: 2307. Springer.