Mining Graph Pattern Association Rules

Xin Wang$^{(\boxtimes)}$ and Yang Xu

Southwest Jiaotong University, Chengdu, China
xinwang@swjtu.cn, xuyang@my.swjtu.edu.cn

Abstract. We propose a general class of graph-pattern association rules (GPARs) for social network analysis, *e.g.,* discovering underlying relationships among entities in social networks. Despite the benefits, GPARs bring us challenges: conventional support and confidence metrics no longer work for GPARs, and discovering GPARs is intractable. Nonetheless, we show that it is still feasible to discover GPARs. We first propose a metric that preserves *anti-monotonic* property as support metric for GPARs. We then formalize the GPARs mining problem, and decompose it into two subproblems: frequent pattern mining and GPARs generation. To tackle the issues, we first develop a parallel algorithm to construct *DFS code graphs*, whose nodes correspond to frequent patterns. We next provide an efficient algorithm to generate GPARs by using *DFS code graphs*. Using real-life and synthetic graphs, we experimentally verify the performance of the algorithms.

1 Introduction

Association rules have been studied for discovering regularities between items in relational data [4]. They have a traditional form $X \Rightarrow Y$, where X and Y are disjoint itemsets. There have been recent interests in studying associations between entities in social graphs, *e.g.,* a special kind of graph pattern association rules are introduced in [7]. While, as these rules have consequents taking only a single edge, they are not capable enough to model even more complicated associations among entities in social networks. Nonetheless, GPARs are more involved with generalized patterns as antecedents and consequents. This highlights the need for studying how to discover generalized GPARs on social graphs.

Example 1. A fraction of a social network G is shown in Fig. 1(a), where each node denotes a person with name 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. One can easily infer the 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. As shown in Fig. 1(b), the antecedent and consequent of the rule, which are represented as graph patterns, *i.e.,* Q_l and Q_r, specify conditions on various entities in a social graph in terms

S. Hartmann et al. (Eds.): DEXA 2018, LNCS 11030, pp. 223–235, 2018.
https://doi.org/10.1007/978-3-319-98812-2_19

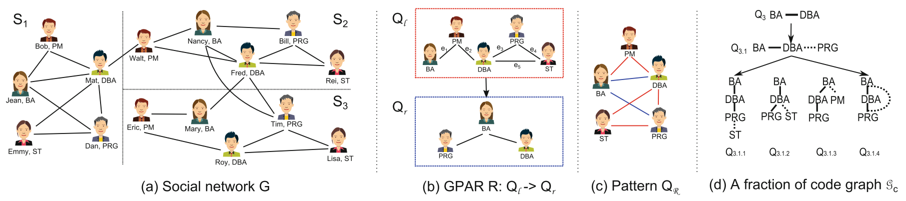


Fig. 1. Graph, association as graph patterns and code graph

of topological constraints, *e.g.,* friendship. With the rule, one can infer social relationships, and recommend friends to others who are most likely interested in, *e.g.,* recommend `Mary` to `Dan`, and `Roy` to `Mary`. □

Though, GPARs have wide applications, they bring several challenges. (1) Conventional support and confidence metrics no longer work for GPARs. (2) Prior techniques cannot be directly applied to discover generalized GPARs. (3) Social graphs are often big and distributively stored, these make mining computation even harder.

Contributions. The paper provides methods to discover GPARs.

(1) We propose generalized GPARs to capture complex social relations among social entities (Sect. 2.2). We define support and confidence metrics for GPARs, decompose the GPARs mining problem into two subproblems, *i.e.,* frequent pattern mining and rules generation, and outline an algorithm for the problem (Sect. 2.3).
(2) We first study the frequent pattern mining problem (Sect. 3.1). We develop a parallel algorithm, that outputs a *DFS code graph* $\mathcal{G}_c$, with nodes corresponding to frequent patterns. The algorithm has desirable performance: it computes support for a node at k-th level in $\mathcal{G}_c$ in $O(|E_f|((k+1)2^{k+1})^{k-1})$ time, where $|E_f|$ is the number of crossing edges. We also study how to generate GPARs with *DFS code graph* $\mathcal{G}_c$ (Sect. 3.2). Given $\mathcal{G}_c = (V_c, E_c)$ and bound η, we then develop an algorithm to produce GPARs with confidence above η in $O(|V_c|(|V_c| + |E_c|))$ time, which is independent of the size of the underlying big graph G.
(3) Using real-life and synthetic graphs, we experimentally verify the performance of our algorithms, and find the following: (a) our mining algorithm scales well with the increase of processors; and (b) they work reasonably well on large graphs (Sect. 4).

Related Work. We categorize related work as follows.

Graph Pattern Mining. The problem has two branches. (1) Algorithms for pattern mining in graph databases are given in [9,10]. (2) Mining techniques over single large graphs are also studied in, *e.g.,* [6]. To improve efficiency, parallel technique is proposed in [13]. Our work differs with [13] in the following: we leverage partial evaluation and asynchronous message passing to identify potential matches in distributive scenario, moreover frequent patterns are our intermediate results.

GPARs *Mining.* A special kind of GPARs and its mining techniques are introduced in [7], where consequents of the GPARs are defined as pattern graphs with a single edge. Another closer work is about mining GPARs over stream data [11]. Our work differs with them 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 introduce graph-pattern association rules.

2.1 Preliminary Concepts

We start with preliminary concepts.

Graph and Subgraph. A *graph* is defined as $G = (V, E, L)$, where (1) V is a set of nodes; (2) $E \subseteq V \times V$ is a set of *undirected* edges ; and (3) each node v in V carries $L(v)$ indicating its label or content *e.g.,* name, job title, as found in social networks. 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 moreover, for each $v \in V'$, $L'(v) = L(v)$. A *directed* graph is defined similarly, but with each edge (v, v') denoting a directed edge from v to v'.

Pattern and Sub-pattern. A *pattern* Q is a graph (V_p, E_p, f), where V_p and E_p are the set of nodes and edges, respectively; each u_p in V_p has a label $f(u_p)$, specifying search condition, *e.g.,* job title. A pattern $Q' = (V'_p, E'_p, f')$ is *subsumed by* another pattern $Q = (V_p, E_p, f)$, denoted by $Q' \sqsubseteq Q$, if (V'_p, E'_p) is a subgraph of (V_p, E_p), and function f' is a restriction of f.

Isomorphism and Subgraph Isomorphism. An *isomorphism* is a bijective function h from the nodes of Q to the nodes of G, such that (1) for each node $u \in V_p$, $f(u) = L(h(u))$, 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* [5] 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, we say G *matches* Q, and denote G_s as a *match* of Q in G. Abusing notations, we say v in G_s as a *match* of u in Q, when $h(u) = v$.

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 $\{(u, \mathsf{img}(u)) | u \in E_p\}$, 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.

DFS Code and DFS Code Tree. The definitions are introduced in [14]. To make the paper self-contained, we cite them as follows (rephrased).

Given a graph $G = (V, E)$, its DFS tree T_G can be built by performing a depth first search in G from a node. Given T_G, a *DFS code* α of G is an edge sequence $(e_0, e_1, \cdots, e_m)$, that is constructed based on the binary relation $\prec_{E,T}$, such that $e_i \prec_{E,T} e_{i+1}$ for $i \in [0, |E| - 1]$. We refer readers to [14] for more details about binary relation $\prec_{E,T}$. A graph G can have multiple DFS trees and a set of *DFS codes*. While one can sort them by *DFS lexicographic order*, then the minimum one, denoted by $\mathsf{min}(G)$, can be chosen as the canonical label of G, and graph isomorphism can be determined by comparing $\mathsf{min}(G)$ [14].

A *DFS code tree* $\mathcal{T}_c$ is a directed tree with a single root, 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 rooted *DFS code graph* $\mathcal{G}_c$ is a directed graph with a single root as virtual node, and non-root node corresponding to a *DFS code*.

Distributed Data Graphs. 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 $(V_i \cup F_i.O,\ E_i,\ L_i)$ 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 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) $(V_i \cup F_i.O, E_i, L_i)$ is a subgraph of G induced by $V_i \cup F_i.O$. In $\mathcal{F}$, we denote $V_f = \bigcup_{i \in [1,n]} F_i.O$ as the set of virtual nodes, E_f as the set of crossing edges, and $|\mathcal{F}|$ as the number of fragments.

We will use the following notations. (1) Pattern Q is *connected* if for each pair of nodes in Q, there exists a path between them. (2) Given a DFS tree, we denote the node being visited lastly via preorder search as the *rightmost node*, then the direct path from root to the rightmost node is named as the *rightmost path*. (3) Given pattern Q, we refer edges that are in the DFS tree as *forward edges*, and remaining edges as *backward edges*. (4) We refer *DFS code*, *DFS code tree* and *DFS code graph* as *code*, *code tree* and *code graph*, respectively, when it is clear from context. (5) Given code α, we refer its corresponding pattern as $Q(\alpha) = (V(\alpha), E(\alpha))$. (6) The *size* $|G|$ of G (resp. $|Q|$ of Q) is $|V| + |E|$ (resp. $|V_p| + |E_p|$), the total number of nodes and edges in G (resp. Q). (7) Given a directed graph G, node v' is a *descendant* of v if there is a directed path from v to v'. (8) Given a directed graph G with a single root v_R, we denote node v as the k-th level node of G, if there exists a path from v_R to v with k edges on the path; and denote the longest path from v_R to a node v in G as the height of G.

2.2 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, 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.

We model a GPAR R as *a graph pattern* Q_R by extending Q_l with edge set of Q_r. We consider nontrivial GPARs by requiring that (1) Q_R, Q_l and Q_r are *connected*; and (2) Q_l and Q_r are *nonempty*, *i.e.,* each of them has at least one edge.

2.3 GPARs Mining

We define support and confidence for GPARs, followed by a mining algorithm.

Support. The support of a pattern Q in a single graph G, denoted by $\mathsf{supp}(Q,G)$, indicates the appearance frequency of Q in G. Several *anti-monotonic* support metrics for graph patterns are introduced. Following "minimum image" [6], which preserves *anti-monotonic* property, we define support of Q as $\mathsf{supp}(Q,G) = \mathsf{min}\{|\mathsf{img}(u)| \mid u \in V_p\}$. Then the support of a GPAR R can be defined as $\mathsf{supp}(Q_R,G)$.

Confidence. To find how likely Q_r holds when Q_l holds, we define the *confidence* of a GPAR R in G as $\mathsf{conf}(R,G) = \frac{\mathsf{supp}(Q_R,G)}{\mathsf{supp}(Q_l,G)}$.

Mining Algorithm. Following traditional strategy for association rules mining, we outline an algorithm, denoted by $\mathsf{GPARMiner}$ (not shown) for mining GPARs. The algorithm first mines frequent patterns Q_R with support above threshold from graph G, then generates a set of GPARs satisfying confidence threshold with Q_R. In Sect. 3, we will introduce how to mine frequent patterns and generate GPARs.

3 Graph Pattern Association Rules Mining

We first introduce the *frequent pattern mining* (FPM) problem, followed by distributed technique for the problem. We then develop method to generate GPARs.

3.1 Distributed Frequent Pattern Mining Algorithm

The FPM problem can be stated as follows: given a graph G, and support threshold θ, it is to find a set S of frequent patterns Q such that $\mathsf{supp}(Q,G) \geq \theta$ for any Q in S. However, the problem is nontrivial. Its decision problem is verified NP-hard by reduction from the NP-complete subgraph isomorphism problem [5]. Despite the hardness, one can leverage parallel computation to improve mining processing. Motivated by this, we develop a distributive algorithm for the FPM problem.

Theorem 1. *There exists a parallel algorithm for* 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 k-th level in $\mathcal{G}_c$ 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.

```
Algorithm FPMiner /* executed at the coordinator */
Input: Fragmented graph $\mathcal{F} = \{F_1, \cdots, F_n\}$ of data graph $G$, support threshold $\theta$.
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 $\alpha$ from QSet if $\mathsf{supp}(Q(\alpha), G) < \theta$;
4.  for each code $\alpha$ in QSet do PatExt$(R, \alpha, \theta, \mathcal{G}_c)$;
5.  return $\mathcal{G}_c$.

Procedure PatExt
Input: $\alpha_p$, $\alpha$, $\theta$ and $\mathcal{G}_c$.
Output: Updated $\mathcal{G}_c$.
1.  if $\alpha \neq \mathsf{min}(Q(\alpha))$ then
2.    connect $v_\alpha$ with its parent $v_{\alpha_p}$ in $\mathcal{G}_c$;
3.    return ;
4.  initialize a new node $v_\alpha$, connect $v_\alpha$ with its parent $v_{\alpha_p}$ in $\mathcal{G}_c$;
5.  generate a set cSet of code, corresponding to a set of candidate patterns;
6.  for each code $\alpha_c$ in cSet do
7.    M := M$\cup$ LocalMine $(F_i, \theta, \alpha, \alpha_c)$;
8.  for each $\alpha_c$ 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. 2. Algorithm FPMiner

The algorithm, denoted as FPMiner and shown in Fig. 2, 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. Before illustrating 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 these match fragments at *workers* 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 is a match of a pattern node u, and send $X_{(u,v)}$, to neighbor sites for local evaluation. When v has not been determined a match of u, $X_{(u,v)}$ is set as unknown; while once v is confirmed 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 frequency of $Q(\alpha)$; and (b) at each *worker*, indexes $\mathsf{M_F}$, $\mathsf{M_S}$ as hash-table, that map codes corresponding to frequent patterns to their match set and images, respectively.

Algorithm. Below, we describe details of the algorithm.

Initialization. The initialization has following 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 local frequency of patterns from all *workers.*
(2) Upon receiving requests from S_c, all *workers* S_i compute local support for single edge patterns, in parallel as following. (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 to a code α in $\mathsf{M_F}$, 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 S_j to generate a match of $Q(\alpha)$ and update 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 will not be included in $\mathsf{img}(u)$ at local fragment, instead, v will be 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 local supports, and sends M to the *coordinator.*
(3) The *coordinator* S_c assembles sets M from all *workers*, initializes QSet by summing up all local supports (line 2). It next eliminates *code* α from QSet if $Q(\alpha)$ is not frequent, *i.e.,* frequency is less than threshold θ (line 3), and repeatedly invokes procedure PatExt to mine larger patterns by using frequent single edge patterns (line 4).

Frequent Pattern Mining. Starting from code α that corresponds to a single edge pattern, S_c invokes procedure PatExt to mine frequent patterns. In a nutshell, the mining procedure consists of three stages. Below we describe 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. Expansion from a code that's generated before may cause unnecessary computation. To avoid this, one can determine whether a code is redundant by applying the rule given by Lemma 1.

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

Lemma 1 tells us that one can determine whether code α is redundant by checking whether α is the *minimum code* of the pattern $Q(\alpha)$. With the lemma, PatExt applies the strategy, given in [14], to verify whether a code α is minimum (line 1). The strategy iteratively (a) expands α'; and (b) compares α with α' during expansion, and terminates current iteration as soon as α is already greater than α'.

Note that PatExt uses a *code graph* $\mathcal{G}_c$ to maintain frequent patterns. As opposed to the growth of *code trees* that only includes a new edge between a pair of nodes that corresponding to *minimum codes*, given code α and its parent

```
Procedure LocalMine /* executed at each site $S_i$ in parallel */
Input: Fragment $F_i$, frequency threshold $\theta$, $\alpha$, $\alpha_c$.
Output: A set $\mathsf{M}$.
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 corresponding variable associated with $v_v$ as true;
7.        invoke EvalT$(id, \alpha, \alpha_c)$ 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. 3. Procedure LocalMine

code α_p, if α is redundant, $\mathcal{G}_c$ will also be expanded 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 otherwise 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 next round pattern extension (Fig. 3).

(2) *Candidate Generation.* To avoid generate duplicate patterns following naive pattern extension strategy, 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 DFS tree of $Q(\alpha)$ as backward edges, or is grown from rightmost node of $Q(\alpha)$ as forward edges; and (b) any node label of G, whose appearance frequency is above support threshold θ can be used to label new node in the updated pattern (line 5).

Example 2. Given a pattern $Q_3 = (\mathsf{DBA}, \mathsf{BA})$, three candidates $Q_{3.1}$, $Q_{3.2}$ and $Q_{3.3}$, shown in Fig. 1(d), 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 $Q_{3.1.2}$: $\{(\mathsf{BA}, \mathsf{DBA}), (\mathsf{DBA}, \mathsf{PRG}), (\mathsf{DBA}, \mathsf{ST})\}$ is already a minimum code. When candidate $Q_{3.2.2}$ is generated, it is considered a duplicate pattern as 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 *codes*, the *coordinator* iteratively sends a candidate code α_c to *workers* to request local support of $Q(\alpha_c)$ (lines 6–7 of PatExt).

Step II: Upon receiving α_c, all *workers* compute matches and *image* of pattern $Q(\alpha_c)$ with procedure LocalMine, in parallel. Specifically, 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 last round computation and are stored in $\mathsf{M_F}$. If $G_{Q(\alpha_c)}$ is verified a valid match of $Q(\alpha_c)$ (line 3), *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"truth" value of each virtual node in $G_{Q(\alpha_c)}$ via procedure EvalT (lines 5–7). While if $G_{Q(\alpha_c)}$ can not be determined a match of $Q(\alpha_c)$, which indicates that $e_i = (u_1, u_1')$ and $e_1 = (v_1, v_1')$ are backward edge and crossing edge, respectively, then EvalU (not shown) is invoked at S_j, where v_1' locates, to verify whether $G_{Q(\alpha_c)}$ is a true match or not (lines 8–9). After verification, if $X_{(u_1,v_1)} = \mathsf{true}$, representing that v_1 is a valid match of u_1, is returned, EvalU 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). Lastly, 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 (not shown) propagates truth value of variables as follows. Upon receiving *id*, α and α_c, it checks 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 current fragment are marked as "dummy" nodes; otherwise if the partial match $G_{Q(\alpha)}$ exists, EvalT simply extends $G_{Q(\alpha)}$ with a new edge corresponding to the new pattern edge in $Q(\alpha_c)$. It then updates $\mathsf{M_S}(\alpha_c)$ and $\mathsf{M_F}(\alpha_c)$ by including the new match. For each virtual node v_v in $G_{Q(\alpha_c)}$, if the variable X_{v_v} has not been evaluated, then EvalT sets corresponding variable associated with v_v as true and invokes EvalT at neighbor site where v_v locates for further propagation.

Observe that if pattern $Q(\alpha_c)$ is grown from $Q(\alpha)$ with a backward edge (u_1, u_1'), then any candidate $G_{Q(\alpha)}$ should be expanded with 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 a valid match of $Q(\alpha_c)$ with local information, then data needs to be shipped to neighbor sites for further verification. In light of this, procedure EvalU is invoked at site where virtual node v_1' locates to check 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 truth variables.

Example 3. After receiving a candidate pattern $Q_{3.1}$ from S_c, the *worker* S_2 computes its local frequency as following. S_2 first extends matches of Q_3 and generates two matches $G_{3.1}^1$ and $G_{3.1}^2$ with node set $\{\mathsf{Nancy}, \mathsf{Fred}, \mathsf{Bill}\}$ and $\{\mathsf{Nancy}, \mathsf{Fred}, \mathsf{Tim}\}$, respectively, and updates $\mathsf{M_S}(\alpha_c))$ and $\mathsf{M_F}(\alpha_c)$ by including $G_{3.1}^1$ and $G_{3.1}^2$, respectively. As $G_{3.1}^2$ has a virtual node Tim as a match of PRG, S_2 evaluates $X_{(\mathsf{PRG},\mathsf{Tim})}$ as true, marks Tim in $G_{3.1}^2$ as dummy node, sends a tuple with $X_{(\mathsf{PRG},\mathsf{Tim})} = \mathsf{true}$ to S_3, and set $\mathsf{M} = \{(\mathsf{BA}, 1), (\mathsf{DBA}, 1), (\mathsf{PRG}, 1)\}$ to the *coordinator*. After receiving tuple from S_2, *worker* S_3 initializes a new match of $Q_{3.1}$ with node set $\{*, \mathsf{Fred}*, \mathsf{Tim}\}$. □

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 is verified frequent. When all codes are processed, PatExt returns *code graph* $\mathcal{G}_c$ as the result (lines 9–11 of PatExt).

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

Analyses. To analyze 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 to k. One may verify the following to see the complexity.

Complexity. When computing support for 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 neighbor site $S_j O(2^{|V_k|}|V_k| + |V_k||F_j.O|)$ time to verify whether v_1' is a match of u_1'. Moreover, the verification request will be 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. □

3.2 Rule Generation

Below we present an algorithm to generate GPARs with *code graph* $\mathcal{G}_c$.

Algorithm. The algorithm, denoted as RuleGen (not shown), takes as input a *code graph* $\mathcal{G}_c = (V_c, E_c)$ and confidence bound η. It first initializes empty set S and queue q. It next repeatedly traverses $\mathcal{G}_c$ from each node v_α by reverse breadth first search. 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. 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. Lastly, RuleGen returns the set S as the final result after all the nodes are processed.

Analyses. To see the complexity of RuleGen, observe 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. □

4 Experimental Study

Using real-life and synthetic data, we conducted following experiments to evaluate (1) the scalability of algorithm FPMiner, and (2) the efficiency of algorithm RuleGen.

Experimental Setting. We used three real-life graphs: (a) *Pokec* [2], a social network with 1.63 million nodes taking 269 different node types, and 30.6 million edges; (b) *Google+* [8], a social graph with 4 million entities of 5 types and 53.5 million links; and (c) *Web* [3], a snapshot of Web graph with 12.1 million Web pages labeled by its domain or country using a label set $\mathcal{L}$ with $|\mathcal{L}| = 100$ and 103.6 million links. We designed a generator to produce synthetic graphs $G=(V,E,L)$, controlled by the numbers of nodes $|V|$ and edges $|E|$, where L is taken from an alphabet of $1K$ labels.

Algorithms. We implemented the following, all in Java. (1) Algorithm FPMiner, compared with (a) $\mathsf{GRAMI_{ND}}$, which is a naive distributed algorithm, that ships all fragments to the *coordinator*, and applies centralized FPM tool GRAMI [1]; and (b) $\mathsf{GRAMI_{D}}$, another distributed FPM algorithm. $\mathsf{GRAMI_{D}}$ first computes support of single edge patterns in the same way as FPMiner does, and broadcasts frequent patterns to each *worker*. All *workers* then invoke GRAMI [6] to compute local supports parallelly, and ship both local supports and those matches with *virtual nodes* to the *coordinator*. The *coordinator* finally assembles results and identifies frequent patterns. (2) Algorithm RuleGen.

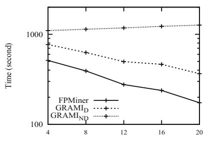
Graph Fragmentation and Distribution. We used the algorithm of [12] to partition graph G into n fragments, and distributed them to n sites ($n \in [1, 20]$). Each site is powered by 8 cores Intel(R) Xeon(R) 2.00 GHz CPU with 128 GB of memory and 1 TB hard disk, using Debian Linux 3.2.04 system. Each experiment was run 5 times and the average is reported.

Experimental Results. We next report our findings.

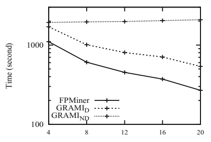
Exp-1: Scalability of FPMiner. In this set of experiments, we evaluated the scalability of FPMiner by varying the number of fragments n. We use *logarithmic scale* for the y-axis in Fig. 4(a)–(b). We started the tests with three real-life datasets.

Varying n. Fixing $\theta = 3K$, $1K$, and $4K$ for *Pokec*, *Google+* and *Web*, respectively, we varied n from 4 to 20 and evaluated efficiency of FPMiner. Figure 4(a), (b) and (c) report the results of FPMiner on *Pokec*, *Google+* and *Web*, respectively, which tell us the following. Algorithm FPMiner allows a high degree of parallelism: The more sites are available, the less time it takes. It is, on average, 4.1, 3.3, and 2.9 times faster when n increases from 4 to 20 on *Google+*, *Web* and *Pokec*, respectively, and scales best among three algorithms. Moreover, it takes 749 s for FPMiner over *Web*, with 20 sites.

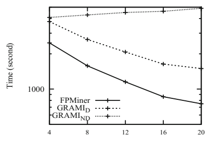
Varying θ. Fixing $n = 4$, we varied support θ from $2K$ to $4K$ in $0.5K$ increments, $0.6K$ to $1.0K$ in $0.1K$ increments, and $3K$ to $5K$ in $0.5K$ increments on *Pokec*,



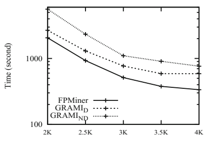
(a) Varying n (Pokec)



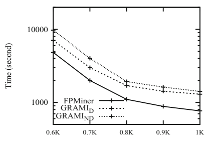
(b) Varying n(Gootle+)



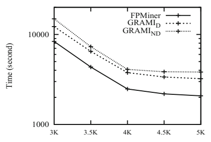
(c) Varying n (Web)



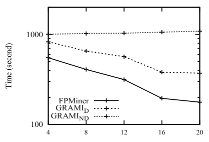
(d) Varying θ (Pokec)



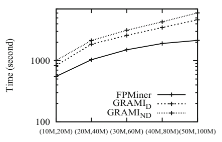
(e) Varying θ (Google+)



(f) Varying θ (Web)



(g) Varying n(Synthetic)



(h) Varying $|G|$ (Synthetic)

Fig. 4. Performance evaluation

Google+, and *Web*, respectively. Figure 4(d), (e) and (f) tell us the following. (1) All algorithms take longer with small θ, because more candidate patterns and their matches need to be verified. (2) FPMiner outperforms $\mathsf{GRAMI_D}$ in all cases, and is less sensitive to the increment of θ. This is because FPMiner maximizes parallelism during support computation, hence is most efficient; $\mathsf{GRAMI_D}$ assembles matches that cross multiple sites, and verifies support with costly centralized method; and $\mathsf{GRAMI_{ND}}$ is essentially a centralized algorithm, and has worst performance, which is as expected.

Varying n (Synthetic). Fixing $|G| = (10M, 20M)$, and $\theta = 4K$, we varied n from 4 to 20. The results are shown in Fig. 4(g), and consistent with Fig. 4(a), (b) and (c). In particular, FPMiner takes 177 s with 20 processors on synthetic graph G.

Varying θ (Synthetic). Fixing $n = 4$, $\theta = 4K$, we varied $|G|$ from $(10M, 20M)$ to (50M, 100M) with $10M$ and $20M$ increments on $|V|$ and $|E|$, respectively. As shown in Fig. 4(h), (1) all three algorithms take longer on larger graphs; (2) FPMiner is less sensitive to $|G|$ than others, since when graphs get larger, the increment of its computational time grows slower than other algorithms; and (3) FPMiner outperforms $\mathsf{GRAMI_D}$ and $\mathsf{GRAMI_{ND}}$ by 1.6 and 2 times, respectively, on average, which is consistent with Fig. 4(d), (e) and (f).

Exp-2: Performance of RuleGen. We evaluated efficiency of RuleGen in this test.

Efficiency. Fixing confidence threshold $\eta = 0.6$, we varied support θ from $2K$ to $4K$ in $0.5K$ increments, $0.6K$ to $1.0K$ in $0.1K$ increments, and $3K$ to $5K$ in $0.5K$ increments on *Pokec*, *Google+*, and *Web*, respectively, and evaluated efficiency of RuleGen. We find that (1) RuleGen is very efficient, taking only 754 ms on *Web* when $\theta = 3K$; (2) RuleGen spends more time when θ gets smaller, as the smaller θ is, the bigger *code graph* $\mathcal{G}_c$ is, hence more time is needed for the traversal.

5 Conclusion

We have proposed generalized graph-pattern association rules (GPARs) and viable support, confidence measures, and shown that GPARs can be used to identify association rules among entities in social graphs. We have provided techniques for GPARs mining. Our experimental study has verified the performance of the algorithms. We contend that GPARs yield a promising tool for social network analysis.

Acknowledgments. This work is supported by NSFC 71490722, and Fundamental Research Funds for the Central Universities, China.

References

1. GraMi. https://github.com/ehab-abdelhamid/GraMi
2. Pokec social network. http://snap.stanford.edu/data/soc-pokec.html
3. Web graph. http://law.di.unimi.it/datasets.php
4. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. SIGMOD Rec. **22**(2), 207–216 (1993)
5. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. TPAMI **26**(10), 1367–1372 (2004)
6. Elseidy, M., Abdelhamid, E., Skiadopoulos, S., Kalnis, P.: GRAMI: frequent subgraph and pattern mining in a single large graph. PVLDB **7**(7), 517–528 (2014)
7. Fan, W., Wang, X., Wu, Y., Xu, J.: Association rules with graph patterns. PVLDB **8**, 1502–1513 (2015)
8. Gong, N.Z., et al.: Evolution of social-attribute networks: measurements, modeling, and implications using google+. In: IMC (2012)
9. Huan, J., Wang, W., Prins, J., Yang, J.: Spin: mining maximal frequent subgraphs from graph databases. In: SIGKDD (2004)
10. Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data. In: Zighed, D.A., Komorowski, J., Żytkow, J. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 13–23. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45372-5_2
11. Namaki, M.H., Wu, Y., Song, Q., Lin, P., Ge, T.: Discovering temporal graph association rules. In: CIKM (2017)
12. Rahimian, F., Payberah, A.H., Girdzijauskas, S., Jelasity, M., Haridi, S.: JA-BE-JA: a distributed algorithm for balanced graph partitioning. In: SASO (2013)
13. Talukder, N., Zaki, M.J.: A distributed approach for graph mining in massive networks. Data Min. Knowl. Discov. **30**(5), 1024–1052 (2016)
14. Yan, X., Han, J.: GSPAN: graph-based substructure pattern mining. In: ICDM (2002)