Efficient Anomaly Detection in Property Graphs

Jiamin Hou, Yuhong Lei, Zhe Peng, Wei Lu(✉), Feng Zhang, and Xiaoyong Du(✉)

School of Information and DEKE, MOE, Renmin University of China, Beijing, China
{jiaminhou,leiyuhong,pengada,lu-wei,fengzhang,duyong}@ruc.edu.cn

Abstract. Property graphs are becoming increasingly popular for modeling entities, their relationships, and properties. Due to the computational complexity, users are seldom to build complex user-defined integrity constraints; worse, the systems often do not have the capabilities of defining complex integrity constraints. For these reasons, violation of the implicit integrity constraints widely exists and leads to various data quality issues in property graphs. In this paper, we aim to automatically extract abnormal graph patterns and efficiently mine all matches in large property graphs to the abnormal patterns that are taken as anomalies. For this purpose, we first propose a new concept namely *CGPs(Conditional Graph Patterns)*. CGPs have the capability of modeling anomalies in the property graph by capturing both abnormal graph patterns and the attribute (i.e., property) constraints. All matches to any abnormal CGP are taken as anomalies. To mine abnormal CGPs and their matches automatically and efficiently, we then propose an efficient parallel approach called *ACGPMiner (Abnormal Conditional Graph Pattern Miner)*. ACGPMiner follows the generation-and-validation paradigm and does the anomaly detection level by level. At each level i, we generate CGPs with i edges, validate whether CGPs are abnormal, and mine all matches to any abnormal CGPs. Further, we propose two optimizations, pre-search pruning to reduce the search space of match enumerations and a two-stage strategy for balancing the workload in distributed computing settings. Using real-life graphs, we experimentally show that our approach is feasible for anomaly detection in large property graphs.

Keywords: graph · abnormal data · parallel

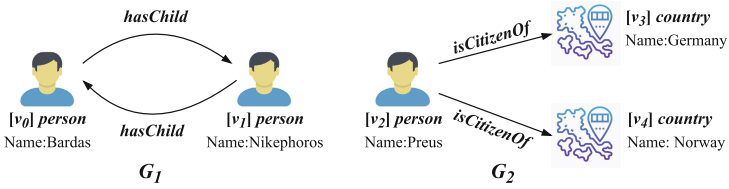
1 Introduction

The graph model has been showing its effectiveness in modeling entities and their relationships in a wide spectrum of applications scenarios, like knowledge bases, transportation graphs, social networks, etc. Due to the computation complexity of graph models, it is seldom to build complex user-defined integrity constraints

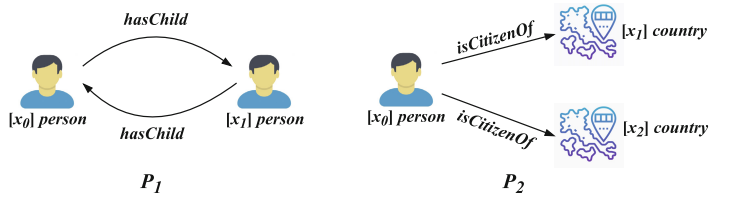
X. Wang et al. (Eds.): DASFAA 2023, LNCS 13945, pp. 120–136, 2023.
https://doi.org/10.1007/978-3-031-30675-4_9

in large graphs [1]. Worse still, some integrity constraints are not trivial to be expressed, e.g., a person is not allowed to have two nationalities with one as Chinese, but is allowed to have two nationalities with one as "the United States" and the other as "England". Due to the above two reasons, violation of the implicit integrity constraints widely exists and leads to various data quality issues in the graph.

Example 1 **(Motivation Example).** Consider two real subgraphs G_1, and G_2 in Fig. 1(a), which are extracted from a classic knowledge graph YAGO [2]. G_1 shows that two persons Bardas and Nikephoros are each other's children, and this modeling is obviously contradictory. G_2 demonstrates that a person named Preus holds both German and Norwegian nationality at the same time. This modeling is also not correct because Norway does not allow dual citizenship. Interestingly, although Norway does not support dual citizenship, some other countries support dual citizenship, such as the United States and England. □



(a) Abornomal subgraphs in YAGO



(b) Abstracted graph patterns behind abnormal subgraphs

Fig. 1. Abnormal subgraphs in YAGO and abstracted graph patterns

Thus far, most of the existing works [3–6] follow the defining-and-identifying paradigm to do graph anomaly detection. Specifically, they first define abnormal graph patterns and then perform graph pattern matching to identify subgraphs that are graph isomorphic to any of the patterns. These subgraphs are considered anomalies. For example, we first define two graph patterns that are P_1 and P_2 in Fig. 1(b), and then identify subgraphs that are graph isomorphic to either P_1 or P_2 in a large graph, e.g., G_1 and G_2, which are considered as anomalies. However, the defining-and-identifying paradigm has two drawbacks. On the one hand, modeling anomalies simply using graph patterns is not adequate.

For example, it is not correct that any matches to P_2 are modeled as anomalies. This is because, as repeatedly discussed, a subgraph G is considered as an anomaly only when G is graph isomorphic to P_2, and the attribute values (resp. country) of vertices of G are confined to a pre-defined set of values, e.g., Norway. On the other hand, the graph patterns need to be defined a priori. In reality, defining the abnormal graph patterns is not trivial, and often, they are modeled after collecting quite a few "manually-reported" anomalies. For example, patterns P_1 and P_2 are modeled by the abstraction of manually-reported anomalies G_1, G_2, and other isomorphic subgraphs can be identified by performing graph pattern matching. Unfortunately, this manually-reported paradigm is incomplete, leaving many more anomalies to be unrevealed.

To mine anomalies from a large property graph automatically and efficiently, in this paper, we first propose a concept namely the conditional graph pattern (a.b.a. CGP) that is able to model the anomalies properly. CGP captures the graph topology as well as the attribute value constraints and hence compared with graph patterns, it takes a more expressive capability to model anomalies. Take G_2 in Fig. 1(a) for example. If we add an attribute constraint $x_1.name =$'Norway' on any match to the graph pattern P_2, then no false positives of anomalies over Norway nationality are produced. We introduce a quantifiable metric namely *abnormality*, based on which we are able to mine CGPs that are considered as anomalies. Furthermore, to make CGPs useful in practice, we propose *ACGPMiner (Abnormal Conditional Graph Pattern Miner)*, an efficient parallel approach to identifying all subgraphs in the property graph that satisfy the requirements of CGPs. ACGPMiner combines pattern mining and attribute discovery in a single process, designs various effective pruning strategies to reduce the search space, and balances the workload in the distributed compute settings using a two-stage strategy. Extensive experiments are conducted on real-life graphs and the results show that our approach is feasible for anomaly detection in large property graphs.

2 Problem Definition

Definition 1 ***(Property graph).*** *A property graph G is modeled as a quadtuple (V, E, L, F_A). (1) V is a set of vertices; (2) E is a set of edges, and $E \subseteq V \times V$; (3)each $v \in V$ is labeled $L(v) \in \Theta$ and each $e \in E$ is labeled $L(e) \in \Theta$, where Θ is an alphabet of the node and edge labels in graphs; (4) each vertex $v \in V$ is associated with a set $A = \{A_1, \ldots, A_n\}$ of attributes (i.e., properties). Attribute values of v are denoted as $F_A(v) = (A_1, c_1), (A_2, c_2), ..., (A_n, c_n)$, where c_i $(1 \leq i \leq n)$ is the attribute value of v over A_i.*

Definition 2 ***(Subgraph, $\in$).*** *Given two graphs $G_1 = (V_1, E_1, L_1, F_{A_1})$ and $G_2 = (V_2, E_2, L_2, F_{A_2})$, G_1 is said to be a subgraph of G_2, written as $G_1 \in G_2$, iff (1) $V_1 \subseteq V_2$, $E_1 \subseteq E_2$; (2) for each vertex $v \in V_1$, $L_1(v) = L_2(v)$ and $F_{A_1}(v) = F_{A_2}(v)$; (3) for each edge $(u, v) \in E_1$, $L_1(u, v) = L_2(u, v)$.*

Definition 3 ***(Isomorphism, $\simeq$).*** *Two graphs G_1 and G_2 are said to be isomorphic, written as $G_1 \simeq G_2$, iff there is a bijective function $f : V_1 \rightarrow V_2$ satisfying: (1) for each vertex $v \in V_1$, $L_1(v) = L_2(f(v))$; (2) for each edge $(u, v) \in E_1$, $(f(u), f(v)) \in E_2$ and $L_1(u, v) = L_2(f(u), f(v))$.*

Definition 4 ***(Graph pattern).*** *A graph pattern is a graph $P[\bar{x}] = (V_P, E_P, L_P, u)$, where (1) V_P (resp. E_P) is a set of vertices (resp. edges); (2) L_P is a function that assigns labels to each vertex $v \in V_P$ (resp. edge $e \in E_P$); (3) $\bar{x}$ is a list of variables, and (4) u is a bijective mapping from $\bar{x}$ to V_P that assigns a distinct variable to each vertex $v \in V_P$.*

Definition 5 ***(Graph pattern matching).*** *A match of a graph pattern P in the graph G is a subgraph G_1 of G that is isomorphic to P, i.e., $G_1 \in G$ and $G_1 \simeq P$.*

Example 2 Figure 1 shows two graph patterns: P_1 and P_2. (a) $P_1[x_0, x_1]$ describes two persons x_0 and x_1 who are each other's children. In P_1, 1) V_P are two vertices and E_P are two edges; 2) L_P assigns the label "person" to both two vertices and the label "hasChild" to both two edges; 3) $\bar{x}$ contains two variables x_0 and x_0; and 4) u maps x_0 to the left vertex and x_1 to the right vertex in P_1. A match of the pattern P_1 in G_1 is $x_0 \rightarrow v_0$ and $x_1 \rightarrow v_1$. (b) Similarly, $P_2[x_0, x_1, x_2]$ indicates that a person x_0 is a citizen of both country x_1 and country x_2. A match of pattern P_2 in G_2 is $x_0 \rightarrow v_2$, $x_1 \rightarrow v_3$ and $x_2 \rightarrow v_4$. □

Graph isomorphism verifies whether two (sub)graphs have an identical structure (i.e. topology). Given a set of (sub)graphs $\mathcal{G} = \{G_i, G_2, ..., G_N\}$, the isomorphism relation divides $\mathcal{G}$ into equivalence classes. Each class is abstracted as a *graph pattern* and subgraphs belonging to the same class are graph isomorphic to each other. In this way, a graph pattern can be considered as a template of all isomorphic subgraphs and a subgraph is considered as an instance (match) of its pattern. However, graph patterns do not contain any attribute information which is required in property graphs. To address this issue, we introduce the conditional graph pattern, given in Definition 6, to impose the attribute constraint on the graph pattern.

Definition 6 ***(Conditional graph pattern).*** *A conditional graph pattern (a.b.a. CGP) is defined as $P[\bar{x}](X)$, where $P[\bar{x}]$ is a graph pattern and X is sets of conditions of $\bar{x}$. A condition has the form of $(x.A, c)$, where x is a variable in $\bar{x}$, A denotes an attribute, and c is the attribute value attached to x over A.*

Definition 7 ***(Conditional graph pattern matching).*** *A match of a CGP $P[\bar{x}](X)$ in the graph G is a subgraph G_1 of G that 1) is isomorphic to P and 2) satisfies all conditions in X.*

In a CGP $P[\bar{x}](X)$, X can be $\emptyset$, which can be seen as a particular conditional graph pattern without additional conditions, i.e., a simple graph pattern. Moreover, to reduce excessive conditional literals, we select a set of active attributes from G that are of users' interest or are attributes contained in most entities.

Example 3 (1) Consider a CGP $P_1[x_0, x_1](\emptyset)$. The condition of this CGP is empty thus it can be targeted as a graph pattern. (2) Consider a CGP $P_2[x_0, x_1, x_2]$ $(x_1.name = 'Norway')$. The condition of this CGP is attached to the variable x_1, which limits the attribute "name" to the value 'Norway'. Compared to P_2 depicting the structure that one person has two nationalities, This CGP additionally requires attribute constraint that the person's one nationality is Norway. With the above CGPs, we can capture the abnormal subgraphs G_1 and G_2 in Fig. 1. □

In order to measure the degree of anomaly of CGPs, we put forward the concept of *abnormality*. Before introducing the concept of abnormality, we first introduce the concept of support to better explain the abnormality.

Definition 8 ***(Support).*** *Consider a graph G, and a CGP $\varphi = P[\bar{x}](X)$, where P has a pivot [7] $z \in \bar{x}$. We define the support of φ as $supp(\varphi, G) = |P(G, X, z)|$, where $P(G, X, z)$ is the set of unique vertices corresponding to the variable z for all matches of φ.*

Given a graph G, a CGP φ and a support threshold λ, we say φ is frequent in G if $supp(\varphi, G) \geq \lambda$. It is intuitive to arise a simple solution that infrequent CGPs can capture abnormal data. But if we consider the entire graph as an instance of a CGP, it cannot occur more than once. It is not enough to simply look for infrequent CGPs. In this paper, we propose the *abnormality* to mine abnormal CGPs as follows.

Definition 9 ***(Abnormality).*** *We define the abnormality of a CGP φ as:*

$$abn(\varphi, G) = \frac{supp(\varphi, G)}{supp\,(\varphi', G)} \tag{1}$$

Here, φ is generated by adding an edge or a condition to a CGP φ'.

If the support of φ' is very large, while the extended newly CGP φ has little even no support, then φ most likely extends some unreasonable/abnormal information. The abnormality describes the ratio of these two support degrees. What's more, abnormality is an additional condition that is applied in infrequent CGPs. So formula (1) has an implicit condition that $supp(\varphi, G) \leq \lambda$.

The Problem Statement. Given a property graph G, a support threshold $\lambda \geq 0$, and an abnormality threshold $\varepsilon \geq 0$, the anomaly detection problem is to extract abnormal CGPs φ over G with $supp(\varphi, G) \leq \lambda$ and $abn(\varphi, G) \leq \varepsilon$ and then find all matches to any abnormal CGPs.

3 ACGPMiner Approach

3.1 Overview

To mine anomalies from a large property graph automatically and efficiently, we propose *ACGPMiner (Abnormal Conditional Graph Pattern Miner)*, a parallel approach to identifying abnormal subgraphs in the property graphs that

satisfy the requirements of CGPs. As shown in Fig. 2, ACGPMiner employs the master-worker paradigm in a multithreaded shared-nothing environment. Communication occurs between the master and workers. The master manages the underlying cluster resources and coordinates the execution of tasks. Workers perform actual specific data processing tasks and report the status of tasks to the master. Each worker in $\{w_0, ..., w_{n-1}\}$ is a process running on multiple cores in $\{c_0, ..., c_{m-1}\}$.

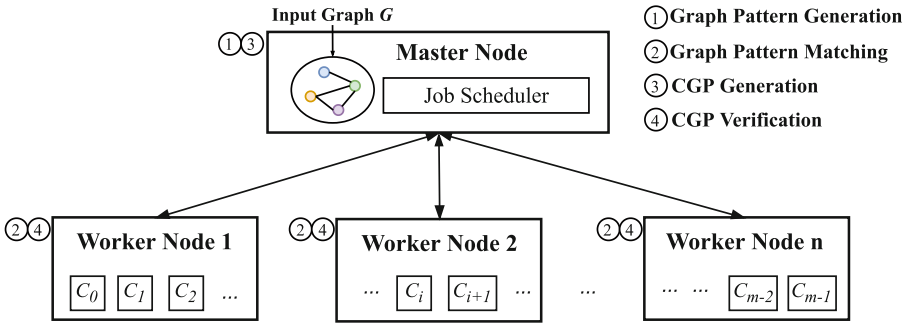


Fig. 2. An overview of ACGPMiner

The core technique of ACGPMiner is to discover abnormal CGPs, based on which we can detect all abnormal subgraphs that match them. To mine abnormal CGPs, ACGPMiner first finds graph patterns P in G, then generates CGPs with P by adding conditions, and finally verifies whether CGPs are abnormal. Specifically, ACGPMiner discovers abnormal CGPs level by level, from smaller CGPs to larger ones. At each level i, it digs out abnormal CGPs with i edges through four significant steps: (1) graph pattern generation to obtain a set of graph patterns; (2) graph pattern matching to find matches for all graph patterns that contribute to CGP candidates; (3) CGP generation to attach conditions to graph patterns for producing candidate CGPs based on matches, and (4) CGP verification for validating whether a CGP is abnormal. In our design, we model graph pattern generation and CGP generation as generating tasks, and model graph pattern matching and CGP verification as computing tasks. Considering that the generating tasks are typically lightweight while computing tasks are prohibitively expensive, ACGPMiner performs generating tasks in the master node and conducts computing tasks in worker nodes in parallel, i.e., assigning multiple workers to compute and execute together computing tasks in parallel.

3.2 Detail Design

We now elaborate on four major steps of ACGPMiner, as shown in Fig. 3.

Graph Pattern Generation. First, we perform graph pattern generation to generate various graph patterns that are candidates for future CGP discovery.

As stated previously, ACGPMiner runs level by level. At each level i, it first generates new graph patterns with i edges. Each graph pattern P' expands a level $i-1$ graph pattern P by adding a new edge (possibly with new nodes). The main techniques of this step have been extensively discussed in numerous works [8–11]. In this paper, we employ the CAM code [8] to guarantee the uniqueness of graph patterns and the FFSM-Join and FFSM-Extend search strategies [11] to generate candidate graph patterns quickly.

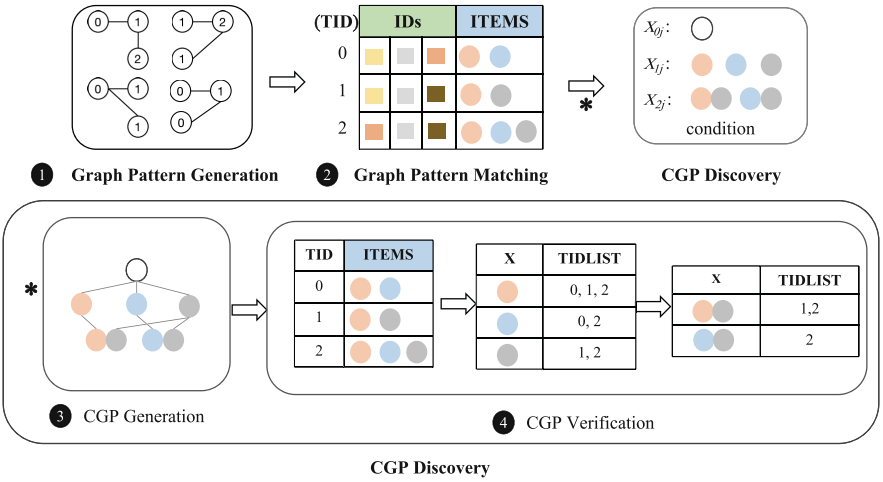


Fig. 3. Four major steps of ACGPMiner

Graph Pattern Matching. Next, we perform graph pattern matching to identify matches for all graph patterns contributing to CGP discovery. ACGPMiner depicts matches in a materialized table view consisting of three sections: *TID*, *IDs*, and *ITEMS*. *TID* is the order of matches; *IDs* characterize the match as a list of unique IDs of vertices; *ITEMS* represent attribute information of each match's vertices as a list of attribute-value pairs. For instance, G_1 is a match of P_1 in Fig. 1. It can be converted to a table view containing *TID* with [0] (assuming it is the first match of P_1), *IDs* with $[v_0, v_1]$ and *ITEMS* with $[(x_0 - name, Bardas), (x_1 - name, Nikephoros)]$. Furthermore, ACGPMiner provides an incremental method, which extends the stored matches to obtain matches of a larger graph pattern. To obtain matches of P, ACGPMiner performs a join operation $Matches(P') \bowtie Matches(e)$, where P is generated by adding a frequent edge e to P'. When doing the join operator, we also perform an isomorphism check and an automorphism check to reduce the exploration space. Since matches may involve large graph data and are therefore computationally expensive, we perform them in parallel. Specifically, we decompose matches across multiple workers, with each worker computing a portion of graph pattern matches.

CGP Generation. Based on matches of the graph pattern, ACGPMiner performs abnormal CGP discovery. CGP discovery is comprised of two subtasks: CGP generation and verification. We first perform CGP generation to obtain a set of candidate CGPs. Recall that a CGP $P[\bar{x}](X)$ is a graph pattern P coupled with conditions X. To introduce conditions to graph patterns, we utilize the semantic attribute information of matches, i.e., the *ITEMS*. We build conditions by starting the search from singleton X collected from the *ITEMS* part and progressing to a larger X through the set combination level by level. For instance, we can add a singleton $(x_0 - name, Bardas)$ on P_1 to generate a candidate CGP $P_1(x_0.name = `Bardas')$.

CGP Verification. After gathering a set of candidate CGPs, ACGPMiner performs CGP verification to check whether a CGP is abnormal. ACGPMiner applies a vertical data format for fast computation. We convert the original data to the vertical-layout data with the format *{X: TIDLIST}*. Here, X is a singleton item, and *TIDLIST* is a list of *TIDs* containing X. For example, the original data of G_1's matches can be converted to $\{\{X : [(x_0 - name, Bardas)]$, *TIDList:* [0]$\}$;$\{X : [(x_1 - name, Nikephoros)]$, *TIDList:* [0]$\}\}$. Based on this design, for getting matches of larger conditions X, we only need to collect the intersection of *TIDLIST* of X's subsets. The support of a CGP with conditions X is the size of *TIDLIST*. Once we get the support of each CGP, we can check whether it is abnormal based on the formula (1).

Pruning. To reduce the search space, we apply the below prunings: (1) Pruning graph patterns: If $supp(P, G) \leq \lambda$, we cease expanding a graph pattern P to produce a larger graph pattern. It is based on the fact that the support of graph patterns is anti-monotonic, meaning that as the graph pattern is expanded, the support decreases. Similarly, no CGP discovery is made if $supp(P, G) \leq \lambda$. (2) Pruning candidate CGPs: ACGPMiner prunes candidate condition sets of length k containing infrequent subsets of length $k - 1$. If X is a frequent condition set, then all of its subsets must also be frequent.

3.3 Discussion

Two computing tasks dominate the cost of ACGPMiner. Both tasks require efficiently generating matches, and there is potential for improvement.

First, there may exist a huge storage overhead in graph pattern matching since we materialize matches of all graph patterns into table views. However, we do not need to materialize matches for infrequent graph patterns since we prune them without making CGP discovery. Inspired by this consideration, we use a pre-search pruning strategy to only materialize matches of frequent graph patterns. A detailed discussion is given in Sect. 4.1.

Second, it is necessary to balance the workload for computing matches in parallel with multiple workers. As previously mentioned, we decompose matches onto multiple workers, where each worker computes a portion of the matches. A basic parallelization strategy allocates all workers to compute matches for a graph pattern. However, not all workers are required to compute each graph

pattern's matches. The matches may be small and solvable by a few workers. For this case, assigning all workers could incur significant communication and synchronization costs and waste valuable resources. To solve it, we design a two-stage strategy for load balancing. The discussion is elaborated in Sect. 4.2.

4 Optimizations

4.1 Pre-search Pruning

We use pre-search pruning to reduce the search space of ACGPMiner. The key insight is that we only get all matches when the support is greater than the threshold λ. To do so, we design a *domain structure* and employ a heuristic search strategy for graph pattern matching on this structure.

Domain Structure. For a graph pattern $P[\bar{x} = x_0, x_1, ..., x_k]$, its domain structure of matches can be formalized as $D = (D_{x_0}, D_{x_1}, ..., D_{x_k})$, where D_{x_i} is the set of vertices that match x_i induced by $m(x_i)$ for all matches m of P in G. For each vertex in D_i, we store its unique *ID* and attribute information *ITEM*. In addition, we store its adjacent vertex for searching matches of larger graph patterns. Semantically, D_{x_i} represents the set of domains of the variable x_i. With the domain structure, we can directly get the support of a graph pattern, i.e., the size of D_z where z is the variable representing the pivot pattern node.

Example 4 To better explain domain structure, we show the representation of a graph pattern $P[\bar{x} = x_0, x_1, ..., x_k]$ in Fig. 4(a). Here, (1) $\{x_0, x_1..x_k\}$ are a set of variables of $\bar{x}$; (2) v_x is a vertex of D_{x_1}, and (3) $\{D_{x_k} : v_y\}$ is v_x's adjacent vertex, indicating that v_x is connected to v_y, which is in the domain D_k. As shown in Fig. 4(b), the domain structure can represent P_2's match G_2. □

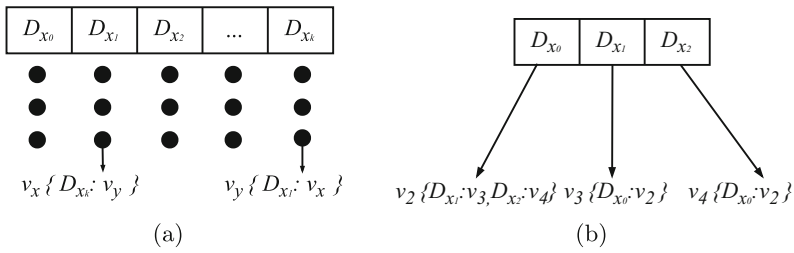


Fig. 4. Domain structure

Graph Pattern Matching. Based on the domain structure, we employ a heuristic search strategy in graph pattern matching to reduce the search space. Specifically, for each vertex $v \in D_z$, we only search for one match containing v. This is due to the fact that our support only considers the size of the vertex specified in the pivot pattern variable. Only if the support meets the threshold

Algorithm 1: Graph pattern matching using domain structure

```
   // get candidate domain structure
1  D(P)=D(P') ⋈ D(e);
   // calculate the support
2  for each vertex u of D_z where z is the pivot do
3  |  find a match m that assigns u to x ;
4  |  if not find then
5  |  |  Remove u from the domain D
6  support=size of D_z ;
   // get all matches
7  if support ≥ λ then
8  |  for each vertex u of D_z do
9  |  |  get all matches consisting u;
```

during the pre-search are all matches containing the v searched. Otherwise, we stop the discovery.

Algorithm 1 details how we use domain structure for incremental graph pattern matching. For a candidate graph pattern P, we first get its candidate domain storage structure by joining matches of P's parent graph pattern P' and its frequent extended edge e (Line 1). We stitch together domain structures D_i corresponding to the connected pattern vertex. Then we apply heuristics to get support on the domain structure (Line 2–6). We iterate over each vertex $u \in D_z$ and search for a match that assigns u to z (Line 3). Note that we perform isomorphism and automorphism checks through searching matches. If the search is unsuccessful, u is removed from D_z (Line 5). Hence, if these vertices are considered in the later pattern matching for getting all matches (Line 7–9), it precludes any further search and reduces the search space. After traversing all vertices of D_z, it is easy to get P's support, which is equal to the size of D_z (Line 6). Only when the support is greater than the threshold λ, do we materialize all matches (Line 7–9).

4.2 Load Balancing

We now discuss the two-stage load balancing strategy, as shown in Fig. 5. The key insight is that candidates with high computational costs can be assigned as many workers as possible, while candidates with low computational costs can be assigned only a few workers. To do so, the first stage builds an approximate computational cost model for deciding the number of workers to compute graph pattern matches. The second stage generates efficient execution plans and assigns workers to perform actual parallel computing. The detailed introduction is as follows.

The First Stage. ACGPMiner generates a pool of jobs, where each job computes matches of one specific graph pattern. In this stage, our goal is to decide

how many workers should be assigned to process a job. To do so, we build an approximate computational cost model which predicts the size of the matches of the newly generated graph pattern. Recall that a graph pattern P's matches are generated by a join $Matches(P') \bowtie Matches(e)$. Drawing on the cost estimation of natural joins in relational databases, we build the approximate model in the following.

$$C(P) = \frac{T(P')T(e)}{Max(V(P', c), V(e, c))} \tag{2}$$

Here, $T(P')$ (resp. $T(e)$) means the count of matches of the candidate graph pattern P' (resp. e). c is the connected pattern node between P' and e. $V(P', c)$ (resp. $V(e, c)$) means the count of distinct values in P' (resp. e) for the variable c. On the basis of the predicted statistics, we can divide one job among n workers for parallel processing. If the predicted cost $C(P)$ is more than a given maximum cost θ, we need $n = max(1, C(P)/\theta)$ workers. Intuitively, if a worker bears data that exceeds the threshold in the future, then the current data will be distributed to other workers. More expensive jobs are assigned to more workers.

The Second Stage. In this stage, we utilize the statistics from the first stage to generate fast execution plans with good load balance. Our goal is to utilize workers as much as possible and not keep them idle for too long. To do so, we handle similar jobs that require almost a similar processing time at the same time. The master continually dispatches jobs to available workers until it becomes empty. Dispatched jobs are prioritized by predicted size; smaller and similar jobs are processed first. Once workers are assigned to jobs, they perform actual parallel computations, i.e., parallel graph pattern matching or parallel CGP verification.

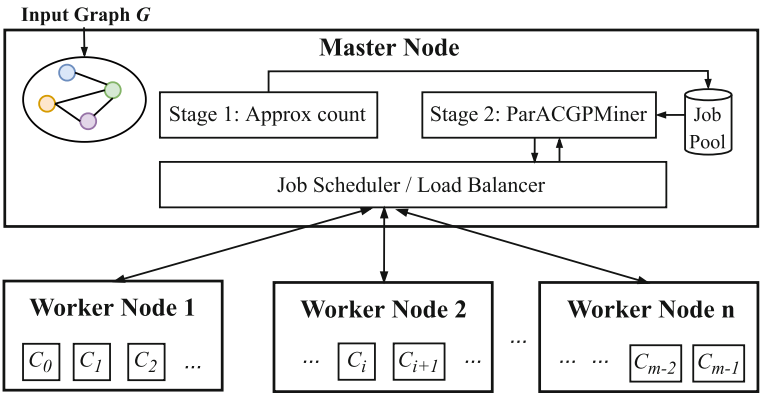


Fig. 5. Two-stage load balancing strategy

5 Evaluation

5.1 Experimental Setup

Implementation. ACGPMiner is implemented in Scala and is built on top of Spark [12]. Spark deals with iterative algorithms in an efficient way and performs the in-memory process to faster the execution. All experiments are conducted on Apache Spark (version 3.1.2). Each executor is set with 5 GB memory and 2 cores. The memory of the driver program is 2 GB. The total cores of each worker are maxed to 8, and the executors of each worker are maxed to 4.

Algorithms. We implement different configurations of ACGPMiner: (1) ParAM: ACGPMiner with the optimizations of pre-search pruning and the two-stage load balancing strategy; (2) DomainAM: ParAM with pre-search pruning but without the two-stage load balancing strategy; (3) TableParAM: ParAM without pre-search pruning but with the two-stage load balancing strategy; and (4) BaseAM: ParAM without pre-search pruning and the two-stage load balancing strategy.

The Compared Method. The CGP is a new proposed definition to mine abnormal data. Existing works were not aimed at mining abnormal CGPs. To compare with other methods, we implement a *Baseline* following the idea proposed in [7], which is one of the most current state-of-the-art methods under the automatic paradigm. We make appropriate changes to fit our topic. Specifically, we run in iterations and discover abnormal CGPs with i edges at each iteration i, which works similarly to our BaseAM algorithm discussed above. The difference is that it uses a brute-force algorithm in attribute discovery. It first lists all candidate CGPs and then iteratively validates each CGP. Our algorithm, however, employs a vertical-data layout for fast validation and does not require scanning through the dataset for each CGP validation. We also implement the Baseline in Scala with Spark for a fair comparison.

Dataset. We use the following two real-life graphs. (a) YAGO: YAGO [2] is a knowledge graph that augments WordNet with common knowledge facts extracted from Wikipedia. We use YAGO with 18 entity types and 36 edge labels. We pick up YAGO with different scales, controlled by the numbers $|V|$ of vertices varying in $\{0.5M, 1M, 1.5M, 2M\}$ and numbers $|E|$ of edges varying in $\{1M, 2M, 3M, 4M\}$. Each entity has an average of 3 attributes in this dataset. (b) DBpedia [13] is another well-known knowledge graph that aims to extract structured content from the information created in Wikipedia. We use DBpedia with 401 entity types, 5M vertices, 268 edge labels, and 14M edges. Each entity in this dataset has an average of 4 attributes.

Hardware Setup. We conduct experiments in an in-house cluster with virtual nodes running CentOS 7.4. Each node is equipped with two Intel(R) Xeon(R) Platinum 8276 CPUs (28 cores $\times$ 2 HT), 8 $\times$ 128GB DRAM, and 3TB NVMe SSDs.

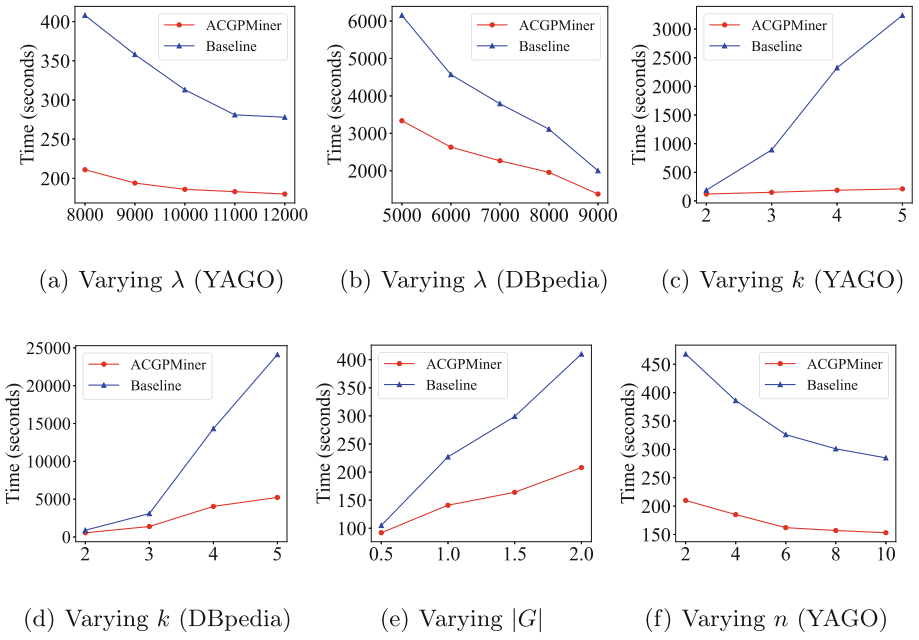
5.2 Experimental Results

We compare the result of our method ACGPMiner (with all optimizations, i.e., ParAM) to the Baseline using a variety of configurations. We next report our findings.

Exp-1: Effect of Support Threshold λ. We first study the performance by varying support threshold λ on YAGO and DBpedia datasets. We set $k = 3$ and $n = 4$ and report the results in Fig. 6(a) and 6(b). First, the running time of both algorithms grows when the support threshold decreases. It is expected that more graph data will be involved with a smaller support threshold. Second, ParAM outperforms Baseline all the time. For YAGO, ParAM is 1.72x faster on average and up to 1.93x than Baseline. For DBpedia, ParAM is 1.69x faster on average and up to 1.84x than Baseline. It confirms that our proposed algorithm is reliable. Third, the support threshold has smaller impacts on ParAM than Baseline. For YAGO, along with the reduction in the support threshold, the running time of Baseline increases by a factor of 1.46 compared to ParAM's 1.1. For DBpedia, Baseline suffers from 3.07x performance loss while ParAM needs 2.4x more running time. Decreasing the support threshold results in an exponential increase in the number of possible candidates and, thus, the exponential decrease in the performance of the mining algorithm. A feasible algorithm should be able to handle a small support threshold. ParAM is more suitable with a low support threshold since ParAM conducts the pre-search pruning which significantly reduces the overhead of materialized pattern matching.

Exp-2: Effect of Pattern Size k. In this experiment, we evaluate the impact of pattern size k. We study the performance on YAGO and DBpedia datasets by varying k from 2 to 5. We set $n = 4$, $\lambda = 2000$ for YAGO, and $\lambda = 9000$ for DBpeida. The results are shown in Fig. 6(c) and 6(d). First, both ParAM and Baseline algorithms need more time to discover abnormal CGPs with larger patterns. Since more CGPs are discovered with larger patterns. Second, matches may be exponentially large since the graph structure is more complex as the number of pattern edges increases. It is challenging to solve such cases. ParAM outperforms Baseline by varying k on both two datasets. ParAM outperforms Baseline by 10 times on average for YAGO and by 3.8 times on average for DBpedia. It again affirms that our method is feasible for property graphs. Furthermore, pattern size k has more negligible impacts on ParAM than Baseline. For YAGO, the running time of ParAM has increased by 1.7x by varying k from 2 to 5. In contrast, the running time of the Baseline has increased by 17.2x. The result is consistent with the DBpedia dataset.

Exp-3: Scalability with $|G|$. We evaluate the scalability by varying the size of graph $|G| = (|V|, |E|)$ from (0.5M, 1M) to (2.0M, 4M). We fix $k = 3$, $n = 3$ and $\lambda = 8000$. As shown in Fig. 6(e), it takes longer to discover abnormal CGPs for larger graphs, as expected. The execution time of ParAM is 1.72x faster than Baseline on average. When the scale of graphs grows to $(2.0M, 4M)$, it takes up to 1.97x less time to discover abnormal CGPs. Moreover, ParAM is



(a) Varying λ (YAGO) (b) Varying λ (DBpedia) (c) Varying k (YAGO)

(d) Varying k (DBpedia) (e) Varying $|G|$ (f) Varying n (YAGO)

Fig. 6. Performance evaluation of ACGPMiner

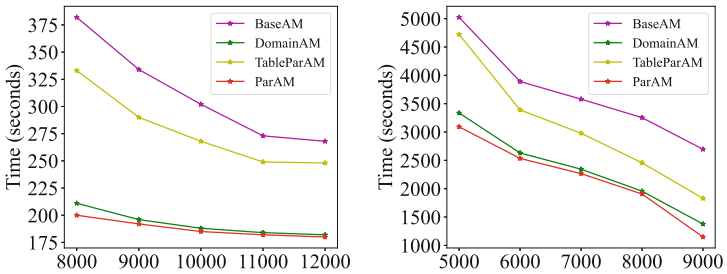
less sensitive to the scale of the graph. As the graph scale increases, the running time of ParAM increases by 2.26x while the time of Baseline increases by 3.9x.

Exp-4: Parallel Scalability. In this experiment, we study the parallel scalability by varying the number n of workers from 2 to 10 on YAGO dataset. We fix $k = 3$ and $\lambda = 8000$. As shown in Fig. 6(f), the running time decreases with the increment of workers. Parallel graph pattern mining and CGP verification dominate the cost. Nonetheless, the parallel costs are reduced when more workers are used. ParAM outperforms Baseline by 2.0x on average and up to 2.3x.

5.3 Optimization Analysis

In this experiment, we study the effect of various optimizations. We compare the performance of various optimizations, i.e., the pre-search pruning and the two-stage load balancing strategy, on YAGO and DBpedia datasets by varying support thresholds. We set $k = 3$ and $n = 3$, and the result is shown in Fig. 7.

We next report our findings. First, for both YAGO and DBpedia datasets, ACGPMiner performs best with all optimizations (denoted by ParAM) and performs worst when no optimization is involved (denoted by BaseAM), as expected. ParAM outperforms BaseAM 1.66x on average for YAGO and 1.68x on average for DBpedia. Second, for both datasets, the optimization of the pre-search pruning is more effective than the two-stage load balancing strategy. Compared to BaseAM, DomainAM outperforms 1.62x while TableParAM outperforms 1.12x.



(a) Optimizations (YAGO) (b) Optimizations (DBpedia)

Fig. 7. The effect of optimizations

It is because we reduce much more search space by applying the pre-search pruning and thus do not need to require computing. Third, we can also observe that the running time takes longer on DBpedia dataset compared to YAGO dataset. It again confirms that the impact of $|G|$ is consistent: the larger G is, the longer ACGPMiner takes.

6 Related Work

Graph Pattern Matching. Given a pre-defined graph pattern, graph pattern matching finds subgraphs in the data graph that are similar (graph isomorphism) to the pattern. Graph pattern matching has been extensively studied in the past decades. A straightforward way is to find matches based on subgraph isomorphism [3]. However, it suffers from huge computational overhead when the graph is big. To reduce time complexity, other methods of a family of graph simulations have been proposed, such as an incremental simulation method [4], a bounded simulation method [5], and a distributed simulation method [6]. However, those methods suffer from either poor expression or in a manual way. First, graph patterns focus on graph structures and ignore the rich attribute information of property graphs. Furthermore, they are required to define graph patterns in advance, which is not trivial and often is developed with a lag after multiple occurrences of abnormal data.

Automatic Discovery of Abnormal Data. Under the automatic paradigm, there exist a few works applying data dependencies to mine abnormal data. Data dependencies are traditionally used to enforce data quality in relations [14–16], and more recently in graphs [1,7,17,18]. As opposed to data dependencies implied in relations, graph dependencies impose the functional dependencies on graph typologies. These works aim to discover laws behind the *normal* graph data. Abnormal data is considered data that does not satisfy these dependencies, which is not intuitive. Also, those works suffer from poor performance since they either use brute force algorithms or do not provide parallel approaches for supporting large-scale graphs.

7 Conclusion

In this paper, we define CGPs to formalize abnormal graph data. CGPs specify the graph structure and attribute conditions in a uniform manner, which can provide a fine-grained paradigm as opposed to graph patterns. We also define the abnormality to measure the degree of exception of abnormal CGPs. Based on the above two notions, we formalize the discovery problem for mining abnormal graph data. To make CGPs useful in practice, we propose a parallel approach, ACGPMiner, for efficiently and automatically mining abnormal CGPs in large-scale graphs. Moreover, we present various optimizations: (1) we design a domain structure and employ a heuristic search strategy for pre-search pruning to reduce search space; (2) we provide a two-stage strategy for load balance. We implement our approach in Scala and build it on top of Spark. Using real-life graphs, we experimentally confirm the effectiveness of our approach.

Acknowledgement. This work was partially supported by National Natural Science Foundation of China under Grant 61972403 and 62072459.

References

1. Fan, W., Lu, P.: Dependencies for graphs. ACM Trans. Database Syst. (TODS) **44**(2), 1–40 (2019)
2. Mahdisoltani, F., Biega, J., Suchanek, F.: Yago3: a knowledge base from multilingual wikipedias. In: 7th Biennial Conference on Innovative Data Systems Research, CIDR Conference (2014)
3. Gou, G., Chirkova, R.: Efficient algorithms for exact ranked twig-pattern matching over graphs. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 581–594 (2008)
4. Fan, W., Wang, X., Wu, Y.: Incremental graph pattern matching. ACM Trans. Database Syst. (TODS) **38**(3), 1–47 (2013)
5. Cao, Y., Fan, W., Huai, J., Huang, R.: Making pattern queries bounded in big graphs. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 161–172, IEEE (2015)
6. Ma, S., Cao, Y., Huai, J., Wo, T.: Distributed graph pattern matching. In: Proceedings of the 21st International Conference on World Wide Web, pp. 949–958 (2012)
7. Fan, W., Wu, Y., Xu, J.: Functional dependencies for graphs. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1843–1857 (2016)
8. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: Third IEEE International Conference on Data Mining, pp. 549–552. IEEE (2003)
9. Miyoshi, Y., Ozaki, T., Ohkawa, T.: Frequent pattern discovery from a single graph with quantitative itemsets. In: 2009 IEEE International Conference on Data Mining Workshops, pp. 527–532. IEEE (2009)
10. Jiang, X., Xiong, H., Wang, C., Tan, A.-H.: Mining globally distributed frequent subgraphs in a single labeled graph. Data Knowl. Eng. **68**(10), 1034–1058 (2009)
11. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: Third IEEE International Conference on Data Mining, pp. 549–552, IEEE (2003)

12. Karau, H., Konwinski, A., Wendell, P., Zaharia, M.: Learning spark: lightning-fast big data analysis. O'Reilly Media, Inc. (2015)
13. DBpedia. http://wiki.dbpedia.org/Datasets
14. Kolahi, S., Lakshmanan, L.V.: On approximating optimum repairs for functional dependency violations. In: Proceedings of the 12th International Conference on Database Theory, pp. 53–62 (2009)
15. Chiang, F., Miller, R.J.: Discovering data quality rules. Proc. VLDB Endowment **1**(1), 1166–1177 (2008)
16. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. IEEE Trans. Knowl. Data Eng. **23**(5), 683–698 (2010)
17. He, B., Zou, L., Zhao, D.: Using conditional functional dependency to discover abnormal data in rdf graphs. In: Proceedings of Semantic Web Information Management on Semantic Web Information Management, pp. 1–7 (2014)
18. Alipourlangouri, M., Chiang, F.: Keyminer: discovering keys for graphs. In: VLDB workshop TD-LSG (2018)