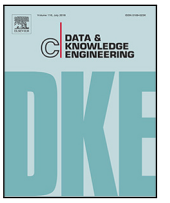
Data & Knowledge Engineering 150 (2024) 102274

Contents lists available at ScienceDirect

Data & Knowledge Engineering

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



# Mining Keys for Graphs

Morteza Alipourlangouri, Fei Chiang *

*McMaster University, 1280 Main St W, Hamilton, L8S 4L8, ON, Canada*

ARTICLE INFO

*Keywords:*
Graph keys
Key mining
Recursive keys

ABSTRACT

Keys for graphs are a class of data quality rules that use topological and value constraints to uniquely identify entities in a data graph. They have been studied to support object identification, knowledge fusion, data deduplication, and social network reconciliation. Manual specification and discovery of graph keys is tedious and infeasible over large-scale graphs. To make GKeys useful in practice, we study the GKey discovery problem, and present GKMiner, an algorithm that mines keys over graphs. Our algorithm discovers keys in a graph via frequent subgraph expansion, and notably, identifies *recursive* keys, i.e., where the unique identification of an entity type is dependent upon the identification of another entity type. We introduce the key properties, *minimality* and *support*, which effectively help to reduce the space of candidate keys. GKMiner uses a set of auxillary structures to summarize an input graph, and to identify likely candidate keys for greater pruning efficiency and evaluation of the search space. Our evaluation shows that identifying and using recursive keys in entity linking, lead to improved accuracy, over keys found using existing graph key mining techniques.

## 1. Introduction

Keys are a fundamental integrity constraint defining the set of properties to uniquely identify an entity. Keys serve an important role in relational, XML and graph databases to maintain data quality standards to minimize redundancy and to prevent incorrect insertions and updates. In addition, keys are helpful for deduplication (also referred to as entity resolution) and have been widely studied for entity identification [1–3]. While keys are often defined by a domain analyst according to application and domain requirements, manual specification of keys is expensive and laborious for large-scale datasets. Furthermore, as changes occur, existing keys may become outdated, and automated methods are needed to refine existing keys, or to discover new keys. Existing techniques have explored mining keys in relational data (as part of functional dependency discovery) [4], and in XML data [5].

The expansion of graph databases has lead to the study of integrity constraints over graphs, including functional dependencies [6,7], graph entity dependencies [6], association rules [8,9], keys [10], and their ontological invariant [11]. The theoretical foundation of these constraints have been studied, and there has been wide application of key constraints towards deduplication, citation of digital objects, data validation and knowledge base expansion [1,12]. Graphs such as knowledge bases and citation graphs require keys to uniquely identify objects to ensure reliable and accurate deduplication and query answering. There is a need to automatically discover keys from such graphs. While RDF graphs structure data as a directed edge-labeled multi-graph, they are widely adopted in the semantic web and linked open data. In contrast, property graph models represents data as a directed, attributed multi-graph. Vertices and edges are objects with labels and key–value pairs called properties. Although recent work has proposed techniques to find keys over RDF data [3], these techniques do not support: (i) topological constraints; and (ii) recursive keys (a distinct feature in graph keys). Consider the following example showing how keys help to identify entities in a graph.

* Corresponding author.
*E-mail addresses:* alipoum@mcmaster.ca (M. Alipourlangouri), fchiang@mcmaster.ca (F. Chiang).

https://doi.org/10.1016/j.datak.2023.102274
Received 10 February 2023; Received in revised form 4 December 2023; Accepted 21 December 2023
Available online 27 December 2023

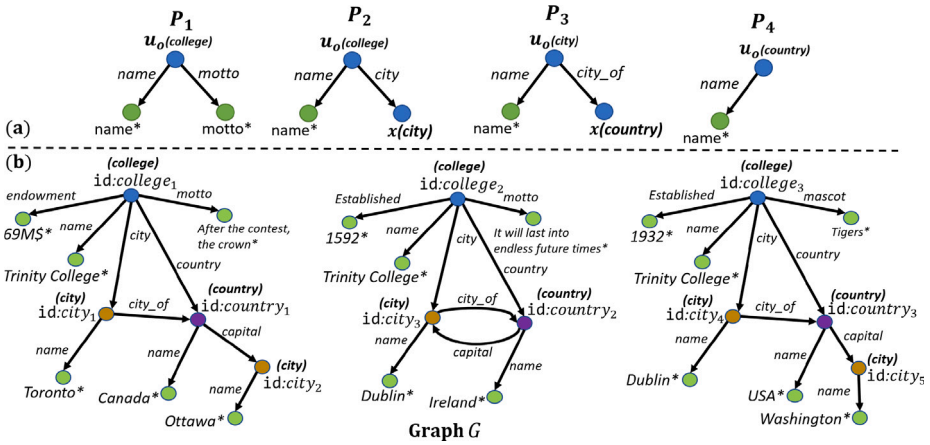



**Fig. 1.** Sample graph from DBpedia.

**Example 1.** Consider a knowledge graph consisting of triples (subject, predicate, object) where the subject and object are nodes, and the predicate is an edge connecting a subject to an object. Fig. 1 shows a sample graph from the DBpedia dataset [13] of three colleges, five cities, and three countries along with the attributes of each entity. Consider graph keys with patterns $P_1$ and $P_2$ in Fig. 1. $P_1$ states that *if two colleges share the same name and motto, then they refer to the same college*. $P_2$ states that *if two colleges share the same name and city, then they refer to the same college.* Similarly, a city can be identified by its name and country as it is shown in $P_3$. Moreover, $P_4$ states that *a country can be identified by its name*. Note that $P_2$ is dependent upon $P_3$, and $P_3$ is dependent on $P_4$, reflecting the recursiveness of graph keys [10].

The example highlights two points: (1) several keys are possible for an entity, with varying topology and attributes. This requires defining key properties to identify *frequent* (well-supported) keys that are *simple* (involving the fewest number of literals). Similar notions have been explored in the discovery of relational keys. (2) The scale and complexity of existing property graphs exhibit embedded entity relationships, and requirements to capture not only attribute values, but structural requirements as part of the key definition. This necessitates new techniques for mining graph keys to include the topology and the ability to consider recursive keys. In the example, $P_1$ uses the *name* and *motto* to uniquely identify the college. However, not all colleges have a motto, resulting in NULL values for some colleges, and leading to poor support and representation across all colleges. This motivates the need to define new key properties and metrics that represent and quantify desirable entity characteristics in the data, and efficient algorithms for the discovery of such keys over graphs.

**Contributions.** We make the following contributions.

1. We define new properties for graph keys (*support* and *minimality*), and formalize the graph key discovery problem.
2. We introduce GKMiner, an algorithm that mines all recursive graph keys by using novel auxiliary structures and optimizations to prune unlikely key candidates.
3. We evaluate GKMiner over real data graphs, and show its scalability with varying parameters. We also compare the effectiveness of (recursive) keys (discovered by GKMiner) against keys found by an existing baseline for an entity linking task. Our results show that GKMiner outperforms SAKey w.r.t. faster runtimes, and using recursive keys lead to improved recall and F1 scores.

**Organization.** We discuss related work in Section 2, and preliminaries in Section 3. In Section 4, we define the key properties, and introduce the GKMiner algorithm. We present our experimental evaluation in Section 5, and conclude with future work discussion in Section 6.

## 2. Related work

Our work finds similarity to several related areas. We first discuss different notions of keys over relations, XML, RDF-based keys, and existing definitions over property graphs. Given our problem to mine keys over graphs, we discuss related work over dependency mining including other types of integrity constraints beyond keys, such as mining graph functional dependencies. We extend the discussion to consider association rule mining over graphs which share similar graph pattern (topology) specifications as GKeys. Lastly, given our problem context to identify keys for entity matching, we discuss existing approaches for entity matching over knowledge graphs.

**Keys and Dependencies.** Keys are defined to uniquely identify entities in a database. For relational data, keys are defined as a set of attributes over a schema [14], or by using unique column combinations [15,16] to uniquely identify the tuples. While the

intuition of recursiveness can be captured via foreign keys in relational data, key discovery algorithms in relational data focus on mining keys over single relations, and do not need to consider topology distinctions among keys, which exist over graphs.

For XML data, keys are defined based on path expressions in the absence of schema [5]. Traditional keys are also defined over RDFs [17–19] in the form of a combination of object properties and data properties defined over OWL ontology. Recent works have studied functional dependencies for graphs (GFDs) that define value constraints on entities that satisfy a topology constraint [6,20]. Temporal extensions to this have included temporal graph functional dependencies (TGFDs) that model topological and attribute value consistency over a set of temporal, graph snapshots [7]. Keys for graphs (GKeys) aim to uniquely identify entities represented by vertices in a graph, using the combination of recursive topological constraints and value equality constraints. GKeys are a special case of GFDs [10]. The recursiveness of GKeys makes it more complex compare to the relational and RDF-based counterparts. Graph matching keys, referred to as GMKs, are an extension of graph keys using similarity predicates on the values, and support approximation entity matching [21].

Graph keys impose topological constraints along with attribute value bindings that are needed to identify entities. Existing techniques do not consider topological constraints, and discover keys as a set of attribute values over RDF data. In our work, we mine keys by considering both topology and attribute values in the form of a graph pattern [10]. Since not all GKeys will be useful, i.e., some may be redundant, we propose a set of properties and optimizations to discover a set of *minimal* keys *efficiently*. A distinction of our work is the discovery of *recursive* keys, where the key for an entity $u_0$ is dependent on the (unique) identification of one or more other entities $u_i, i \in \{1 \dots m\}$.

PG-Keys propose a modular and flexible model to formalize keys for property graphs [22]. PG-Keys are used in conjunction with a property graph query language, ISO Graph Query Language (GQL) project. PG-Keys define constraints over nodes, edges, and properties in a property graph. PG-keys define a set of requirements for keys over property graphs including: (i) a scope that defines the set of objects to which the key applies; (ii) a descriptor that specifies the object in the key scope; and (iii) a key value, which must be exclusive, mandatory, and be unique (singleton). The latter combination of these key values defines varying key types imposed over property graphs. GKeys, however, focus on uniquely identifying entities (*i.e.,* nodes), which may contain recursive key definitions within an entity, e.g., the unique identification of a country relies on a dependent, capital city. For property graphs, a uniqueness constraint is a set of attributes whose values uniquely identify an entity in the collection. Neo4j keys [23] are based on uniqueness constraints and require the existence of such constraints for all vertices the graph. A new principled class of constraints called embedded uniqueness constraints have been proposed that separates uniqueness from existence dimensions, and are used in property graphs to uniquely identify entities [24]. GKeys, however, are different than these constraints by supporting topological constraints through a query graph pattern.

Recent work by a subset of the authors define a new class of temporal graph functional dependencies (TGFDs) that enforce topological and attribute value consistency over a series of snapshots on temporal graphs [7]. While both GKeys and TGFDs utilize a graph pattern to specify topological constraints, their semantics are different, i.e., TGFDs enforce attribute value dependencies over a time interval, and do not allow variable nodes in query graph patterns to model recursiveness.

**Dependency Discovery**. Key mining approaches have been studied for relational databases as data-driven [25] and schema-based [26] techniques. TANE [4] proposed a level-wise, schema-based approach to mine functional dependencies (including keys) in relational data, and over RDF data [17]. Recent approaches have proposed mining techniques for new classes of dependencies such as Ontology Functional Dependencies (OFDs) [27,28], and TGFDs [29].

KD2R [18] extends the relational data-driven approach of Sismanis et al. [26] by exploiting axioms (such as the subsumption relation) and considers multi-valued attributes. SAKey [30] extends K2DR by introducing additional pruning techniques to discover approximate keys with exceptions. VICKEY [31] has extended SAKey to mine conditional keys over RDFs. To avoid scanning the entire dataset, all three techniques (*i.e.,* K2DR, SAKey, and VICKEY) first discover the maximal non-keys and then derive the keys from this set. Non-keys are the set of attributes that are not keys and maximal non-keys are super-sets of all other non-keys. Instead of exploring all combinations of attributes, the main idea behind these techniques is to find those combinations that are not keys, and then derive the keys from this set. These approaches prune the large space of candidates by returning only keys satisfying desirable support, and conditional constants. However, the discovered keys do not include sub-entities within its key definition (e.g., the key for a university relies on a key for its city), and do not permit specification of topological constraints. We select SAKey as a baseline to compare GKMiner performance on "semantically equivalent" keys, which may not be completely satisfied by the entire graph $G$. That is, GKMiner provides tunable support levels thereby discovering "approximate" keys, a feature that SAKey inherently provides.

Fan et al. develop a parallel algorithm to discover GFDs in graphs [32]. Although GKeys are a special case of GFDs, their technique is not able to mine GKeys. In order to model GKeys as GFDs, we need to have a graph pattern consist of two connected components to define equality over pairs of matches. However, the recent GFD discovery algorithm [32] only mines GFDs with a single connected component pattern, making it unable to mine GKeys. To the best of our knowledge, we are only aware of the work from Alipour et al. to mine GKeys [33]. We extend this work to define new metrics to mine GKeys, propose an efficient algorithm with optimizations, and perform new comparative evaluations, including with SAKey [30].

**Association Rule Mining**. Graph temporal association rules (GTARs) detect regularities between complex events in temporal graphs. It differs from conventional association rules in syntax and semantics [34]. Similar to GKeys, GTARs use graph patterns to express topological requirements, but also to represent associations that should hold over a specific time window. The authors define the notion of support and confidence for GTARs by using a minimal occurrence of the pattern within a time window. A discovery algorithm is proposed that combines techniques from event mining and rule discovery, where the algorithm first identifies events at different timestamps, and then dynamically appends the promising events within a window to generate candidate GTARs, and

then performs validation. Fan et al. propose graph-pattern association rules (GPARs), which extend association rules for item sets. GPARs are useful to discover regularities between entities in social graphs [8]. A GPAR is defined as $Q(x, y) \Rightarrow q(x, y)$ where $Q(x, y)$ is a graph pattern in which $x$ and $y$ are two designated nodes, and $q(x, y)$ is an edge between $x$ and $y$ with the label $q$. Association rules have an inherently different semantics than dependencies and keys, that is non-functional.

Evolution rules (ERs) detect local changes occurring in an evolving graph [35]. ER mining uses a frequency-based approach to identify changes between a graph pattern and its sub-pattern. The mining algorithm decomposes the graph pattern into a *body* and *head*, and defines rules as *body* $\rightarrow$ *head*, where the head is a single-edge emerging event between two nodes in body. Our GKeys mining takes a level-wise approach to generate candidate keys of increasing number of nodes by expansion of previous (unsatisfying) candidates. This process ensures minimal GKeys are returned, which are non-overlapping from previously discovered keys.

**Graph and Entity Matching**. Entity matching over knowledge graphs can be classified into two approaches: (1) value-based methods that compute attribute-based similarity across instances; and (2) record-based techniques that adopt similarity, learning and rule-based methods [36]. Entity matching using GKeys, which are dependency rules defining the conditions for unique identification, requires the use of graph matching techniques to identify entity instances. This has wide applications, including entity linking, missing value imputation, and deduplication over knowledge graphs. Recent work has used graph embeddings to capture structure and semantics to match entities in knowledge graphs [37]. Probabilistic graph matching computes a mapping to induce similar subgraphs from a pair of graph instances [38]. By using GKeys, we can precisely define the topological and attribute value conditions to uniquely identify an entity over large-scale graphs.

## 3. Preliminaries

**Graphs.** A *directed graph* is defined as $G = (V, E, L, F)$ with labeled nodes and edges, and attributes on its nodes. The set $V$ is a finite set of vertices, and $E \in V \times V$ is a set of edges. Each node $v \in V$ has a label $L(v)$, and edge $e \in E$ has a label $L(e)$. For a node $v$, $F(v)$ is a tuple to specify the set of attributes as $(A_1 = a_1, \ldots, A_n = a_n)$ of $v$. More specifically, $A_i$ with a constant value $a_i$ determines the attribute $A_i$ of $v$ written as $v.A_i = a_i$. A graph $G' = (V', E', L', F')$ is a *subgraph* of $G$ denoted as $G' \subseteq G$ if $V' \subseteq V, E' \subseteq E$, and for each node $v \in V'$, $L'(v) = L(v)$, and for each edge $e \in E'$, $L'(e) = L(e)$, and $F'(v) = F(v)$ for each $v \in V'$. Attributes model the properties of an entity such as *name*, *age*, etc., as found in social networks and knowledge graphs. We note two attributes $v$.type and a numeric id, denoted by $v$.id, representing the entity type and its unique URI. In this work, we focus on attributed, property graphs, and key discovery over such graphs.

**Example 2.** Returning to Fig. 1, we have three colleges with unique ids $\{college_1, college_2, college_3\}$ and all of them have the attribute name = Trinity College. However, $college_1$ and $college_2$ have motto, while $college_3$ has mascot. Moreover, entities are connected to each other via edges, *e.g.*, $city_1$ with the name = Toronto has an edge *city_of* to $country_1$ named Canada.

**Graph pattern.** A *graph entity pattern* is defined as a connected, directed graph $P(u_o) = (V_P, E_P, L_P)$ where (1) $V_P$ is a finite set of pattern nodes; (2) $E_P$ is a finite set of pattern edges; (3) $L_P$ is a function that assigns a label $L_P(v)$ to each vertex $v \in V_P$, and a label to each edge $e \in E_P$. The pattern nodes $V_P$ may be one of three types: (1) a *center* node $u_o \in V_P$, representing the main entity to be identified; (2) a set of *variable nodes* $V_x \subseteq V_P$; or (3) a set of *constant nodes* $V_c = V_P \setminus V_x$. A variable node is mapped to an *entity*, and contains a type and id, while a constant node has no id. Since $V_x \subseteq V_P$, $u_0$ may be contained in $V_x$, and we assume that the center node contains an id to uniquely identify the main entity.

**Graph pattern matching.** We use a similar matching semantics as past work on keys for graphs that is the conventional semantics of matching using subgraph isomorphism [10]. A *match* of a graph entity pattern $P(u_o)$ in $G$ is a subgraph $G'$ that is isomorphic to $P(u_o)$. There exists a bijective function $h$ from $V_P$ to $V'$ such that: (i) for each node $v \in V_P$, $L_P(v) = L'(h(v))$; and (ii) for $e = (u, u') \in E_P$, if and only if there exists an edge $e' = (h(u), h(u')) \in G'$, and $L_P(e) = L'(e')$. We note that $L_P(v) = L'(h(v))$ always holds if $L_P(v) =$ '_', denoting that wildcards match any labels.

**Example 3.** Given pattern $P_1$ of Fig. 1(a), we find matches $h_1$ and $h_2$ in graph $G$ of Fig. 1(b), such that $h_1(college) = college_1$ and $h_2(college) = college_2$. $college_3$ is not a match of $P_1$ as there is no match for the node motto. However, there exist three matches $h_1$, $h_2$ and $h_3$ for pattern $P_2$ in $G$ for $college_1$, $college_2$ and $college_3$ (all with value Trinity College), respectively. Similarly, we have three cities $city_1$ (Toronto), $city_3$ (Dublin), $city_4$ (Dublin) matched with the pattern $P_3(city)$ in $G$ and all three countries $country_1$ (Canada), $country_2$ (Ireland) and $country_3$ (USA) match pattern $P_4(country)$.

**Graph keys** (GKeys). A key for a graph is defined using a pattern $P(u_o)$ for a designated entity $u_o$.

**Definition 1.** A graph key for an entity type $\tau$ is a graph pattern $P(u_0)$ where $u_0$ is of type $\tau$. [10]

Intuitively, for a graph $G$, for entities of type $\tau$, the specified conditions in $P(u_0)$ uniquely identify $u_0$. Since $P(u_0)$ is a graph pattern, it necessarily includes topological constraints. A topological constraint is specified using a graph pattern, defined as a connected, directed graph. This is similar to past definitions of graph keys that also specify a desirable graph structure, including PG-Keys [10,22]. Fig. 1(a) shows four topological constraints defined via graph patterns.

Given two matches $h_1$ and $h_2$ of $P(u_0)$ in graph $G$, $(h_1, h_2)$ satisfies $P(u_0)$ denoted as $(h_1, h_2) \vDash P(u_0)$, if (a)$\{\forall v \in V_x, h_1(v).\mathsf{id} = h_2(v).\mathsf{id}\}$; (b) $\{\forall v \in V_c, L(h_1(v)) = L(h_2(v))\}$; and (c)$\{\forall e \in E_P, L(h_1(e)) = L(h_2(e))\}$; then $h_1(u_o).\mathsf{id} = h_2(u_o).\mathsf{id}$. This means the two matches refer to the same entity in $G$. We say a graph $G$ satisfies a key $P(u_o)$, denoted as $G \vDash P(u_o)$, if for every pair of matches $(h_1, h_2) \in G$, we have $(h_1, h_2) \vDash P(u_0)$. Moreover, we say a key $P(u_o)$ is a *recursive* key if it contains at least one variable node $v \neq u_o$, otherwise, $P(u_o)$ is called a *value-based* key [10].

**Example 4.** Returning to Fig. 1, a GKey $P_1(college)$ uniquely identifies $college_1$ and $college_2$ as they have different motto, despite the same name. $P_2(college)$ is a recursive GKey that uniquely identifies all three colleges. It is recursively dependent on city in $P_3(city)$, while city is recursively defined via country of the GKey $P_4(country)$. Although $city_3$ and $city_4$ share the same name Dublin, they belong to different countries USA and Ireland, respectively. There exist two levels of recursion in $P_2(college)$ (via city and then country), to uniquely identify all three colleges in $G$.

## 4. Discovery of GKeys

We introduce the GKey discovery problem. Given an input graph $G$, find a set of GKeys for a given type $u_o$ satisfying a minimum support level.

### 4.1. Key properties

To avoid returning an overly large number of keys, we mine minimal and frequent GKeys as defined by a support threshold. *Minimality* avoids mining redundant GKeys, and reduces the discovery time. *Support* mines keys that satisfy a minimum number of instances in $G$. We define these notions, and then introduce the GKMiner algorithm.

GKey **Embedding**. We say a GKey $P(u_o) = (V_P, E_P, L_P)$ is *embeddable* in another GKey $P'(u_o) = (V'_P, E'_P, L'_P)$, if there exists a subgraph isomorphic mapping $f$ from $V_P$ to a subset of nodes in $V'_P$ that preserves node labels/values of $V_P$, and all the edges that are induced by $V_P$ with the corresponding edge labels.

**Minimality.** A GKey $P(u_o)$ is minimal if there exists no GKey $P'(u_o)$ such that $P'(u_o)$ is embeddable in $P(u_o)$. A set $\Sigma$ of GKeys with $G \vDash \Sigma$ is minimal, if: (1) for every $P(u_o) \in \Sigma$, $P(u_o)$ is minimal; and (2) there is no $P(u_o)$ that can be removed from $\Sigma$ such that the resulting set $\Sigma'$ is logically equivalent to $\Sigma$, *i.e.,* $\Sigma'$ uniquely identifies the same entities as $\Sigma$ in $G$.

**Support.** For a candidate GKey $P(u_o)$, we define support as the ratio of the number of entities in $G$ (with type equal to $u_o$) that are uniquely identified by $P(u_o)$ to the total number of entity instances of type $u_o$. Let $|P(u_o)|$ be the number of entities uniquely identified by $P(u_o)$. Let $N$ be the total number of instances of type $u_o$ in graph $G$. For a GKey $P(u_o)$, such that $G \vDash P(u_o)$, we define $\mathsf{sup}(P(u_o)) = \frac{|P(u_o)|}{N}$. Intuitively, support measures the ratio of satisfying occurrences to the total number of occurrences. For example, consider $u_o$ to be of type `Professor`, and there exist 100 entities of type `Professor` in $G$. If 75 of these entities are uniquely identified by $P(u_o)$, then support $\mathsf{sup}(P(u_o)) = 75\%$. We seek keys that uniquely identify a frequent number of entity occurrences. We define a support threshold $\delta$, where for each discovered key $\mathsf{sup}(P(u_o)) > \delta$.

***k-bounded*** GKeys. For a given user defined natural number $k$, a GKey $P(u_o)$ is *k-bounded* if $\mathsf{size}(P(u_o)) \leq k$, where size is defined as:

$$\mathsf{size}(P(u_o)) = |E_p| + \mathsf{size}(P(v_P)), \forall v_p \in V_P \tag{1}$$

$\mathsf{size}(P(u_o))$ counts the number of edges in the pattern $P(u_o)$, including the pattern(s) of all variable nodes (in recursive GKeys). To validate a recursive GKey, one must validate the matches of the recursive patterns [10]. A set $\Sigma$ of GKeys is *k-bounded*, if each $P(u_o) \in \Sigma$ is *k-bounded*.

**Problem statement.** Given a graph $G$, a node type $u_o$, a support threshold $\delta$, and a natural number $k$, mine all minimal *k-bounded* GKeys $\Sigma$ of the node type $u_o$, such that for each GKey $P(u_o) \in \Sigma$, $\mathsf{sup}(P(u_o)) > \delta$.

### 4.2. GKMiner: A level-wise graph key mining algorithm

GKMiner is a sequential GKey discovery algorithm that proceeds in a level-wise manner to generate patterns of size $i$ at level $i, 0 < i \leq k$. We create new patterns via vertical expansion of a candidate pattern $Q_i$ at level $i$ to $Q_{i+1}$ at level $i+1$. We compute matches of $Q_i$ in $G$ that localizes subgraph isomorphism operations.

For a given entity type $u_o$, a naive algorithm mines all frequent graph patterns centered on $u_o$, and explores all combinations of variable and constant nodes in each pattern to verify whether they form a GKey. Such an approach explores a large search space, which is infeasible in real world graphs. We introduce GKMiner, an efficient algorithm to mine all minimal GKeys in a graph. GKMiner discovers both value-based and recursive keys, where the former only involves constant-based nodes in the candidate pattern, and does not require recursive expansion of variables nodes to evaluate a dependent key. GKMiner takes as input a graph $G$, an entity type $u_o$, a natural number $k$ and a support threshold $\delta$ to discover GKeys. It proceeds in three steps: (a) Create a summary graph $S$ to explore the structure of $G$. This will help us to prune nodes that cannot form a GKey based on the given support threshold $\delta$. (b) Create a lattice $\mathcal{L}$ of candidate GKeys from $S$ that prunes further candidate GKeys. (c) Mine minimal *k-bounded* GKeys from $\mathcal{L}$ in a level-wise search. Using a lattice to model candidate GKeys, and with early pruning techniques performed on the lattice, GKMiner avoids the large search space from the naive approach. We describe each step of GKMiner next.

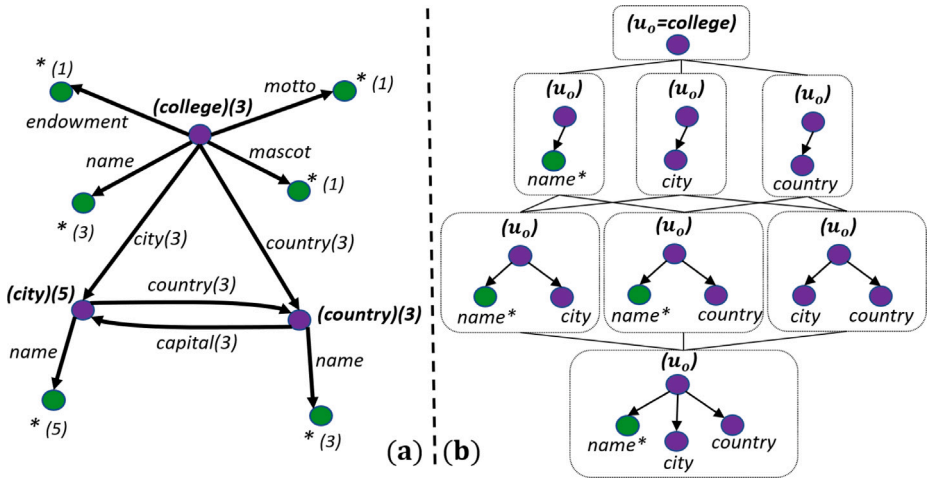



**Fig. 2.** (a) Summary graph $S$ of graph $G$ of Fig. 1(b) Lattice for the type *college* based on sup = 75%.

#### 4.2.1. Graph summarization

As the first step of mining GKeys, we traverse $G$ to create a summary graph $S$ that reflects the structure of $G$. $S$ provides an abstract graph of $G$, where: (1) nodes represent the entity types that exist in $G$, and (2) an edge between two nodes in $S$ shows that there exists at least one edge between two entities with the corresponding types in $G$. $S$ helps us to model the relationship between entity types via a smaller graph. $S$ will be used to define graph patterns for candidate GKeys. $S$ is an auxiliary data structure $S(V_S, E_S)$, where $V_S$ (resp. $E_S$) is a set of nodes (resp. edges), with the following properties:

1. For each node type $t$ in graph $G$, there exists a node $v_t$ in $V_S$.
2. For each node $v_t \in V_S$, $v_t$.count is the number of nodes in $G$ of type $t$.
3. For each edge, $e = (u_1, u_2)$ in $G$:
   (a) if $u_1$ is of type $t_1$ and $u_2$ does not carry a type, *i.e.,* a constant node, then create an attribute $A_e$ with the name $L(e)$, and without any value (*e.g.,* set value as $*$) and add to $v_{t_1}$ in $V_S$. Increase the $A_e$.count by one (initial value is 0).
   (b) if $u_1$ is of type $t_1$ and $u_2$ is of type $t_2$, then add an edge $e = (u_1, u_2)$ to $E_S$ and increase the $e$.count by one (initial value is 0).

**Example 5.** Fig. 2(a) shows the summary graph generated for the graph $G$ of Fig. 1 where we have three entities of type college, five entities of type city and three of type country. Out of three colleges, one of them has the attribute endowment, one has mascot, two have motto, and all three have attribute name.

After computing the summary graph, we prune the summary graph to find a set of attributes and variable nodes that meet the support threshold $\delta$, if they are added to a candidate GKey. For a given summary graph $S(V_S, E_S)$, and a given center node $v_{u_o}$, we first prune the attributes of $v_{u_o}$ based on $\delta$. Following the support definition of Section 4.1, we compute the support of an attribute $A$ of $v_{u_o}$ in $V_S$ as following:

$$\text{sup}(A) = \frac{A.\text{count}}{v_{u_o}.\text{count}} \tag{2}$$

This equation computes the support of an attribute $A$, if we add $A$ as a singleton attribute in a candidate GKey. If we have $\text{sup}(A) < \delta$, then adding $A$ to any candidate GKey $P(u_o)$, makes $\text{sup}(P(u_o)) < \delta$, hence $P(u_o)$ will not be a valid GKey. Therefor, we select a set of candidate attributes $\mathcal{A} = \{A_1, \ldots, A_n\}$ of $v_{u_o}$ in $V_S$ such that for each $A_i$, $\text{sup}(A_i) \geq \delta$. We utilize hashing techniques to identify and mark different node types for classification and counting. For a set of $n$ nodes, aggregating the nodes requires $O(n)$ in space and time complexity to compute the summary graph.

Similar to Eq. (2), we compute the support of the variable nodes, that are immediate neighbors of $v_{u_o}$. If $v_{u_o}$ is connected to a node $v$ with an edge $e$, then the support of $v$ is computed as:

$$\text{sup}(v) = \frac{e.\text{count}}{v_{u_o}.\text{count}} \tag{3}$$

Following the same reasoning of Eq. (2), adding a variable node $v$ with $\text{sup}(v) < \delta$ to a GKey $P(u_o)$, makes $\text{sup}(P(u_o)) < \delta$. Hence, we define a set of variable nodes $\mathcal{V} = \{v_1, \ldots, v_n\}$, where $v_{u_o}$ is connected to each $v_i$ and $\text{sup}(v_i) \geq \delta$.

#### 4.2.2. Lattice construction

For the entity type $u_o$, we create a lattice $\mathcal{L}(u_o)$ of candidate patterns based on the set $\mathcal{A}$ and $\mathcal{V}$ that are extracted from the summary graph $S(V_S, E_S)$. $\mathcal{L}(u_o)$ is rooted at node $u_o$ and expands level-wise based on the attributes in $\mathcal{A}$, and the immediate variable nodes connected to $v_{u_o}$ in $\mathcal{V}$. We create the lattice $\mathcal{L}(u_o)$ as follows:

1. Create a lattice $\mathcal{L}(u_o)$ rooted at node $x$ of type $u_o$ (level 0).
2. At the first level, we create a candidate GKey for the attributes and variable nodes in $\mathcal{A}$ and $\mathcal{V}$. For each attribute $A_i \in \mathcal{A}$, we create a candidate GKey by connecting $u_o$ to $A_i$ with an edge labeled by the name of $A_i$ and add the candidate to $\mathcal{L}(u_o)$. For each variable node $v_i \in \mathcal{V}$, we connect $u_o$ to $v_i$ with the corresponding edge label from $S$ and add as a candidate GKey to $\mathcal{L}(u_o)$.
3. At level $l$, we create a graph pattern for each $l$-combinations of the attributes and nodes in $\mathcal{A}$ and $\mathcal{V}$ respectively. Similarly, we connect $u_o$ to each of the nodes with a direct edge and add the pattern to $\mathcal{L}$. A candidate pattern $P(u_o)$ of level $l-1$ is connected to a pattern $P'(u_o)$ of level $l$ with a direct edge, if $P(u_o)$ is embedded in $P'(u_o)$.
4. Each pattern $P(u_o) \in \mathcal{L}(u_o)$ has a boolean flag $P(u_o)$.prune set by default to false. This flag helps us to mine minimal GKeys and prune the candidates in the lattice.

The lattice $\mathcal{L}(u_o)$ is created for the entity type $u_o$ to generate candidate GKeys that initially meet the support threshold $\delta$. However, since $\mathcal{L}(u_o)$ might contain other recursive entity types from the set $\mathcal{V}$, we need to create a lattice $\mathcal{L}(v_i)$ for each entity type $v_i \in \mathcal{V}$.

**Example 6.** Fig. 2(b) shows the sample lattice created for the type *college* based on the summary graph of Fig. 2(a) given the support threshold sup = 75%. If we calculate the sup for the attributes of the college, we have $\text{sup}(name) = \frac{3}{3}$, $\text{sup}(endowment) = \frac{1}{3}$, $\text{sup}(motto) = \frac{1}{3}$, and $\text{sup}(mascot) = \frac{1}{3}$. Based on the sup = 75%, we have $\mathcal{A} = \{name\}$. Similarly, if we compute the support of variable nodes connected to college, we have $\text{sup}(city) = \frac{3}{3}$, and $\text{sup}(country) = \frac{3}{3}$, leads us to have $\mathcal{V} = \{city, country\}$. Using $\mathcal{A}$ and $\mathcal{V}$, we created the lattice in Fig. 2(b), where we have three levels in the lattice with seven candidates GKeys.

#### 4.2.3. Mining via level-wise search

GKMiner is a sequential GKey mining algorithm that traverses a lattice in a level-wise manner to mine all GKeys for a given type $u_o$. We first create the summary graph $S$ from the input graph $G$. Next, we create the main lattice $\mathcal{L}(u_o)$ and traverse the lattice level by level to discover GKeys and prune when an embeddable key is already mined. The lattice is constructed and traversed *level-by-level* starting with single node candidates at level 1. Since the entire lattice is not constructed completely, we only visit subsequent levels that contain promising (non-pruned) candidates, thereby saving search time and memory. For each candidate $P_i(u_o)$ at level $i$, we check if it forms a GKey via incremental matching algorithm IsoUnit which enables localized subgraph isomorphism [39]. For each candidate $P_i(u_o)$ that has the prune flag equal to false, we first check $\text{size}(P_i(u_o))$ to ensure it is $k$-bounded. If $\text{size}(P_i(u_o)) > k$, then we set prune = true for all the descendant nodes of $P_i(u_o)$ in $\mathcal{L}(u_o)$. Next, we calculate $\text{sup}(P_i(u_o))$ by computing the matches as described in Section 4.1. If $\text{sup}(P_i(u_o)) \geq \delta$, then we report $P_i(u_o)$ as a GKey and prune its descendant nodes in $\mathcal{L}(u_o)$ to ensure the minimality of GKeys. However, if $\text{sup}(P_i(u_o)) < \delta$, we ignore $P_i(u_o)$ and continue with the next candidate.

**Algorithms.** Algorithm 1 provides the pseudo code of the GKMiner. After initialization (line 1–4), we create the summary graph $S$ by iterating over the nodes and edges of $G$. We compute the number of nodes of type $u_o$ in $G$ and assign this value to $N$. For each node $v \in G.V$, we add a node of the corresponding type of $v$ to $S$ and maintain the count of the nodes (lines 5–9). Next, we iterate over the edges of $G$ and add/maintain the edges and their count in $S$ based on the type of the two end points of each edge in $G$ (lines 10–18). After creating the summary graph $S$, we call the Discovery algorithm and pass $S$ and $u_o$ along with other inputs to find GKeys.

The pseudo code of the Discovery algorithm is provided in Algorithm 2. It is a recursive algorithm to evaluate *k-bounded* GKeys for a given type. The algorithm takes as input the graph $G$, a center type $u_o$, summary graph $S$, (initially empty) dependency graph $D$, (initially empty) set of keys, and three integers $k$, $\delta$, and size. The value of size is initially set to 0 and it will be updated for the recursive calls to avoid mining recursive GKeys of size greater than $k$. We first create a lattice for the given type $u_o$ (line 1). The function takes $u_o$, the summary graph $S$ and $\delta$ as an input. It first computes the set of attributes $\mathcal{A}$ and variable nodes $\mathcal{V}$ from $S$ based on the support parameter $\delta$. It then creates the lattice based on the combination of $\mathcal{A}$ and $\mathcal{V}$. Next, we traverse the lattice in a level-wise manner and check whether each candidate pattern forms a GKey. Despite traversing the lattice level-wise, the candidate patterns have always height = 1. For each pattern that is not to be pruned (line 3), we first check if it is *k-bounded* (lines 4–5). If the pattern contains a recursive type $t$ without a GKey, then we need to call the Discovery algorithm for $t$. We first check if adding the edge $(u_o, t)$ creates a cycle in $D$. If so, we remove the edge from $D$ and remove $t$ from the pattern to avoid cycles in recursive calls (lines 6–9). Otherwise, we call the Discovery algorithm by passing $t$ and the current size of the GKey (line 11). If we were not able to find a GKey for $t$, then we prune the pattern and its descendants (lines 12–13). After these steps, we find the matches of the pattern and compute the number of entities that are uniquely identified by the candidate GKey (line 14). We classify matched instances into a set of classes $\Pi_{P(u_o)} = \{\pi_1, \ldots, \pi_k\}$, where each class $\pi$ has a set of matches $\{h_1, \ldots, h_n\}$ with the same id on center node $u_o$ *i.e.,* $h_1(u_o).\text{id} = h_2(u_o).\text{id}$, $h_2(u_o).\text{id} = h_3(u_o).\text{id}$, $\ldots$, $h_{n-1}(u_o).\text{id} = h_n(u_o).\text{id}$. A class $\pi$ is unique if it only contains matches for the same ID. A GKey $P(u_o)$ is *unique*, if all the classes in $\Pi_{P(u_o)}$ are unique. If $\text{sup}(P_i(u_o)) \geq \delta$ and $|\Pi_{P_i(u_o)}|$ only contains single classes (*i.e.,* uniquely identifies all entities matched by $P_i(u_o)$), then we report $P_i(u_o)$ as a GKey and prune its descendant nodes in the lattice

**Algorithm 1:** GKMiner ($G$, $u_o$, $k$, $\delta$)

keys := ∅; /* set of keys for each type*/
Initialize $\mathcal{D}(V, E) := \emptyset$ /*empty dependency graph*/
Initialize $\mathcal{S}(V, E) := \emptyset$ /*empty summary graph*/
$N$:= count nodes of type $u_o$ in $G$
**foreach** *node* $v \in G.V$ **do**
    $t = v$.type;
    **if** $t \neq$ null **then**
        **if** $u_t \notin \mathcal{S}.V$ **then** add $u_t$ to $\mathcal{S}.V$;
        $u_t$.count + +;
**foreach** *edge* $(v_1, v_2) \in G.E$ **do**
    $t = v_1$.type;
    **if** $v$.type == null **then**
        **if** $l \notin F(u_t)$ **then** add $l$ to $F(u_t)$; /*add $l$ as an attribute of $u_t$*/
        $u_t.l$.count + +;
    **else**
        $t' = v_2$.type;
        **if** $(u_t, u_{t'}) \notin \mathcal{S}.E$ **then** add $(u_t, u_{t'})$ to $\mathcal{S}.E$;
        $(u_t, u_{t'})$.count + +;
Discovery ($G$, $u_o$, $N$, $\mathcal{S}$, $\mathcal{D}$, keys, $k$, $\delta$, 0);
**return** keys;

**Algorithm 2:** Discovery ($G$, $u_o$, $N$, $\mathcal{S}$, $\mathcal{D}$, keys, $k$, $\delta$, size)

$\mathcal{L}(u_o)$ := createLattice($\mathcal{S}, u_o, \delta$); /*create lattice for the given type $u_o$*/
**foreach** *pattern* $P(u_o) \in \mathcal{L}(u_o)$ **do**
    **if** $P(u_o)$.prune == false **then**
        **if** $|E_{P(u_o)}|$ + size > $k$ **then**
            $P(u_o)$.prune = true; **continue**;
        **if** $P(u_o)$ *contains type* $t$ *and* keys[$t$] == null **then**
            add $(u_o, t)$ to $\mathcal{D}$;
            **if** $\mathcal{D}$ *has cycle* **then**
                remove $(u_o, t)$ from $\mathcal{D}$; remove $t$ from $P(u_o)$ and $\mathcal{L}(u_o)$;
            **else**
                Discovery ($G$, $t$, $\mathcal{S}$, $\mathcal{D}$, keys, $k$, $\delta$, $|E_{P(u_o)}|$ + size);
                **if** keys[$t$] == null **then**
                    $P(u_o)$.prune = true; **continue**;
        $\mathcal{M}$ := IsoUnit($G, P(u_o)$);
        $\Pi_{P(u_o)}$ := computeClasses($\mathcal{M}, P(u_o)$);
        **if** sup($P(u_o)$) ≥ $\delta$ **then**
            keys[$u_o$].$add(P(u_o))$; /*$P(u_o)$ is a valid GKey for $u_o$*/
**return** keys;

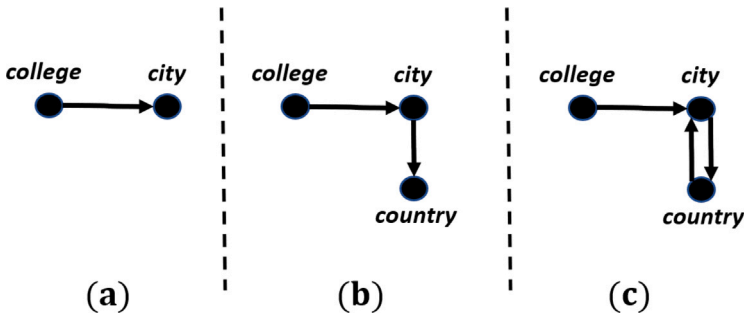



**Fig. 3.** Dependency graph $D$.

to ensure the minimality of GKeys. We add the pattern as a GKey for $u_o$ if it meets the support threshold $\delta$ (lines 16–17). At the end, we return the set of keys that are found.

**Handling recursive** GKeys. In the process of mining GKeys for the type $u_o$, if the candidate $P_i(u_o)$ contains a variable node of type $t$ (*i.e.,* $P_i(u_o)$ is a recursive key), we first need to evaluate and find the GKeys for the dependent type $t$. To this end, we create the lattice $\mathcal{L}(t)$ and recursively call the GKMiner for the type $t$. We maintain a data structure called *dependency graph* $D(V_D, E_D)$ to detect and avoid cycles in recursive calls. Cycles lead us to fall into an infinite loop of recursive calls similar to deadlocks in process management [40]. Cycles happen when there exists a set of types that the GKey of each type is dependent to the GKey of another type in the cycle. Using the dependency graph, we follow a cycle prevention strategy and avoid cycles in recursive calls. To avoid such cycles, whenever we call GKMiner for the type $t$ while mining GKeys for type $u_o$, we add $u_o$ and $t$ to $V_D$ of $D$, and then add a direct edge $(u_o, t)$ to $E_D$. In general, if adding an edge $(t_i, t_j)$ leads us to have a cycle in $D$, we break the cycle by removing the dependency $(t_i, t_j)$. To this end, we remove $t_j$ from the nodes in $\mathcal{L}(t_i)$. In this case, the GKeys of $t_i$ won't be dependent to the GKeys of $t_j$.

**Example 7.** Going back to Fig. 2(b) and assuming sup = 60%, when we want to check a GKey for the type *college* that contains the type *city*, we find that there exists no GKey for *city* yet. Hence, we need to call GKMiner for *city* and we add an edge $(college, city)$ to the dependency graph $D$ as shown in Fig. 3(a). While mining GKey for *city*, we need to call GKMiner for the type *country* and we add an edge $(city, country)$ to $D$ in Fig. 3(b). However, while mining GKeys for the *country* and as there is no GKey for the type *city* yet, we cannot call GKMiner for *city*. As shown in Fig. 3(c), the edge $country, city$ makes a cycle in $D$. Hence, we need to remove *city* from all the candidate for the type *country* and continue the mining to avoid cycle in $D$.

**Complexity**. Constructing the summary graph takes time $O(V+E)$ to traverse $G$ and determine all unique entity instances, and edges. Traversing the lattice, $\mathcal{L}(u_o)$, takes time exponential in the number of attributes of $u_o$. For each dependent entity, $u_j, j \in \{1 \dots M\}$, we must construct and traverse such a lattice, thereby increasing the complexity by a factor of $M$. In practice, most keys are relatively small, similar to graph dependencies for small $k$ [32]. In addition, pruning strategies eliminate edges in the lattice that connect to supersets of discovered keys minimize the discovery time, and ensure minimality of discovered keys. While the attribute domain plays an important role, i.e., attributes with high cardinality are more likely to lead to GKeys due to their unique value combinations. We exploit this intuition, and describe this as an optimization next.

### 4.3. Optimizations

In this section, we propose an optimization for the GKMiner algorithm. While creating the summary graph $S$, and as we check the existence of the attributes for each node in $G$, we maintain a hash-map of the values in the attribute domain. This helps us to find which values are unique for each specific attribute. For an attribute $A$, we hash the values $\{a_1, \dots, a_n\}$, where $a_i$ is the value of the attribute $A$ for the node $v_i$ in $G$. The result of the hash is a set of classes $\{\pi_1, \dots, \pi_m\}$, where each $\pi_j$ has one or more equal attribute values, assuming the collision is handled in the hashing process. If a value $a_i$ uniquely exists in a class $\pi_j$, then $a_i$ is a unique value for the attribute $A$ among all the nodes that carry $A$. For each node $v_i \in G$, we may maintain a bit vector flag called unique. We set $v_i.\text{unique}(A) = \text{true}$, if the corresponding value $a_i$ is unique among all nodes that share the same type as $v_i$ and carry attribute $A$. We can use the unique bit vector when computing the set of matches for $P(u_o)$. Assume $P(u_o)$ contains a set of constant nodes $\{v_{c1}, \dots, v_{cn}\}$. For a match $h \in \mathcal{M}$, if we have $h(u_o).\text{unique}(v_{ci}) = \text{true}$ for any attribute $v_{ci}$, then $h$ is uniquely identified by $P(u_o)$ without further exploration. This is true as if an attribute $v_{ci}$ is unique for a node $h(u_o)$ in $G$, then any combination of the attributes that contains $v_{ci}$ is unique for $h(u_o)$. Note that hashing can be performed in constant time. Hence, we maintain the unique bit vector for all the attributes in $G$ while creating the summary graph $S$ with the same time complexity $O(V + E)$.

## 5. Experiments

We first evaluate GKMiner's efficiency, including its comparative runtime performance against SAKey, and the benefit of its optimizations. Secondly, we evaluate the effectiveness of GKMiner in a qualitative, comparative study against SAKey, with respect to a data linking task. Our experiments show that our algorithm runs up to 6x faster than SAKey, despite mining topological constraints of GKeys compared to the value constraint based keys mined by SAKey.

### 5.1. Experimental setup

We implement all our algorithms in Java v17, and ran our experiments on a Linux machine with AMD 2.7 GHz CPU with 128 GB of memory. Our source code and test cases are available online.[1]

**Datasets.** We used three real graphs for our experiments.

1. *DBpedia* [13]: The graph contains in total 5.04M entities with 421 distinct entity types, and 13.3M edges with 584 distinct labels. DBpedia is extracted from the Wikipedia pages.

[1] https://github.com/mac-dsl/GraphKeyMiner.git.

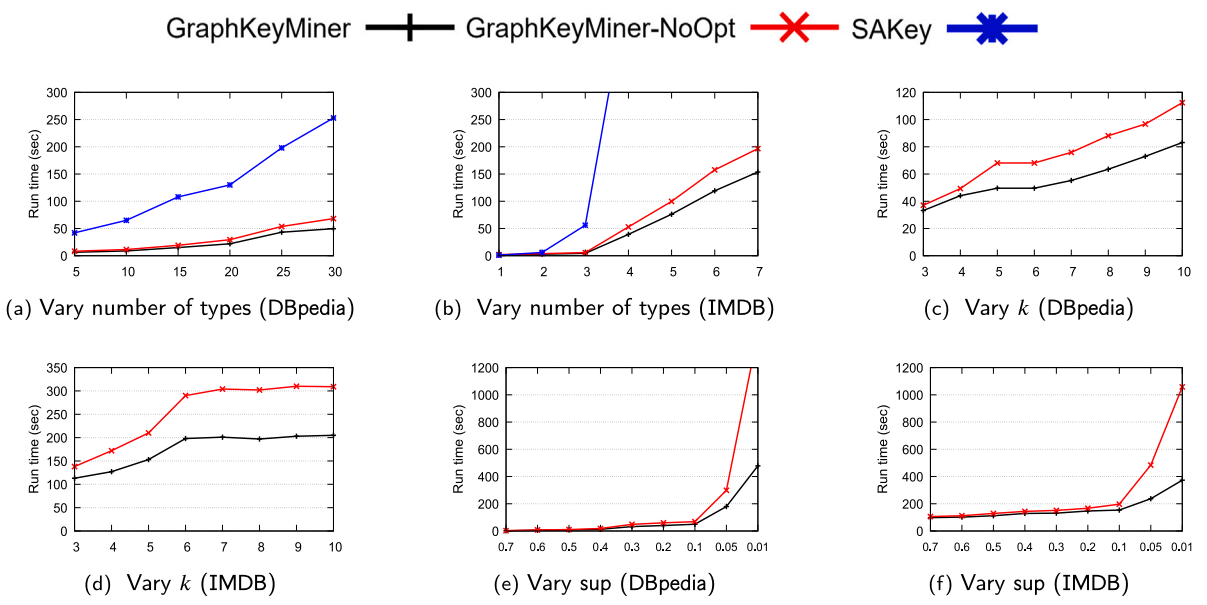



**Fig. 4.** GKMiner efficiency.

2. *IMDB* [41]: The data graph contains 6.1M entities with 7 types and 21.3M edges. This dataset describes movies extracted from the IMDB website, with a total of 44.2M facts.
3. *DBpediaYago* [31]: This dataset links entities from DBpedia [13] and Yago [42]. The ground truth entity links are available on the Yago Web page [43]. This dataset uses the ground truth to link the entities across the two knowledge bases. For each entity, we rewrite the attributes of the entity in the Yago using its DBpedia counterparts.

**Baselines.** We implemented the following algorithms.

1. *GKMiner*: Our graph key mining algorithm with optimizations (Section 4).
2. *GKMiner-NoOpt*: The GKMiner algorithm without optimizations, i.e., no usage of the unique vector.
3. *SAKey* [30]: Rule-based mining algorithm that discovers maximal non-keys first, and then derives keys from this set. SAKey does not consider topological constraints to mine keys for graphs.

**Parameters.** We tune the support and $k$ parameters to determine settings that provided a sufficient number of keys with varying topologies that can be manually validated by domain experts. Similarly, we selected the number of types based on obtaining a sufficient number of keys that could be manually verified to serve as the ground truth. We use a default support sup = 10%, unless otherwise noted.

### 5.2. Experimental results

We evaluate the efficiency of GKMiner against GKMiner-NoOpt and SAKey. Next, we compare the quality of the mined keys in a data linking task using the DBpediaYago dataset [11].

**Exp-1: Vary the number of types.** All three algorithms take a desirable entity type as input. To compare the scalability of the algorithms, we vary the number of types and evaluate the runtime. Using the DBpedia and IMDB datasets, we fixed $k = 5$, and vary the number of types from 5 to 30 for the DBpedia dataset, and from 1 to 7 for the IMDB dataset. After multiple executions, we tuned the support and $k$ parameters to determine settings that provided a sufficient number of keys with varying topologies that can be manually validated by domain experts. Similarly, we selected the number of types based on obtaining a sufficient number of keys that could be manually verified to serve as the ground truth. For SAKey, we set $n = 1$ to find exact keys as we do in GKMiner. Figs. 4(a) and 4(b) show the runtime of the three algorithms over the DBpedia and IMDB datasets, respectively. GKMiner is on average 30% faster than GKMiner-NoOpt and 6 times faster than SAKey. This demonstrates the efficiency of our optimizations. We stopped executions of SAKey over IMDB after 120 min. SAKey was only able to finish mining keys for the types *distributor*, *genre*, and *country* which in total contain only 6.77% of the facts in the IMDB dataset. However, both GKMiner and GKMiner-NoOpt were able to mine GKeys for all types in less than 200 s.

**Exp-2: Vary the pattern size** $k$. We vary the size of the pattern $k$ from 3 to 10 over DBpedia, and IMDB datasets, using 30 and 7 types respectively. We excluded SAKey since there exists no parameter to tune pattern size. Fig. 4(c) shows the runtime over the

**Table 1**
Comparative accuracy of GKMiner against SAKey.

| Entity type (# triples) | GKMiner P/R/F | SAKey P/R/F |
|---|---|---|
| Book(258.4K) | 0.99/**0.07**/**0.13** | **1**/0.03/0.06 |
| Actor(57.2K) | **1**/**0.36**/**0.52** | 0.99/0.27/0.43 |
| Museum(12.9K) | **1**/**0.21**/**0.34** | **1**/0.12/0.21 |
| Scientist(258.5K) | **0.99**/**0.09**/**0.16** | 0.98/0.05/0.11 |
| University(85.8K) | **0.99**/**0.12**/**0.21** | **0.99**/0.09/0.16 |
| Movie(832.1K) | **0.99**/**0.12**/**0.21** | **0.99**/0.04/0.08 |

DBpedia dataset. We observe the following: (1) By increasing the value of $k$, the runtime increases as we have larger patterns to match in $G$. (2) On average, GKMiner runs 33% faster than GKMiner-NoOpt due to our optimization that reduces the search space of entities, and more efficiently finds unique values for candidate GKeys. The same trend exists in the IMDB dataset (Fig. 4(d)), except that the runtime stablizes for $k > 6$. We can explain this behavior by noting that our dataset only contains 7 types, and the recursion depth (*i.e.,* the maximum diameter of the dependency graph) is limited, compared to the DBpedia dataset with over 400 distinct types.

**Exp-3: Vary the support of a** GKey. We vary the support, sup value, from 0.01 to 0.7 (*i.e.,* 1% to 70%) over the DBpedia and IMDB datasets with 30 and 7 types respectively, using a fixed pattern size $k = 5$. We exclude SAKey as there was no option to control the support of a key. Figs. 4(e) and 4(f) show the runtime results for the DBpedia and IMDB datasets, respectively. As expected, for increasing sup values, the runtime decreases over both datasets. GKMiner prunes more aggressively at higher support levels. Fewer candidates satisfy the stringent threshold, and less are evaluated at each level of the lattice. On average, GKMiner runs 66% and 42% faster than GKMiner-NoOpt on DBpedia and IMDB respectively.

**Exp-4: Effectiveness of** GKMiner **vs.** SAKey. We study the quality of the discovered GKeys vs. those from SAKey, for the task of entity linking. GKeys are commonly used to link entities across disparate knowledge bases, i.e., if two entities from different knowledge bases are identified by a key, sharing the same key attributes, then we validate whether they are in fact the same entity, by checking against the ground truth (from the DBpediaYago dataset) [31].

Table 1 shows the comparative precision (P), recall (R) and $F_1$-score(F) measures. We observe that the precision is consistently over 98%, with relatively equal performance between both algorithms. This demonstrates the high quality of keys returned from both methods. The recall scores are relatively low, with the highest scores occurring in the *Actor* category at 36% and 27%, respectively, for GKMiner and SAKey. recall scores for the other categories can partially be explained by the varying context that may be required to obtain more fine-grained matches, which strict equality matching, which both methods use, may be insufficient. Furthermore, data incompleteness in each of the Yago and DBpedia datasets contribute to missed matches as the two datasets are linked during the key validation phase, and fewer matches are found for a candidate key, leading to lower recall. However, we highlight that the value of adopting recursive keys can be seen in the improved recall scores of GKMiner over SAKey. For example, for *Movie* class, the recall increases from 4% to 12% when recursive keys are used. On average, we observe a +7% increase in recall, and +9% in $F_1$-score when using GKMiner over SAKey. This demonstrates the effectiveness of the (recursive) GKeys mined by GKMiner.

## 6. Conclusion and future work

We propose GKMiner, an efficient algorithm to mine recursive graph keys (GKeys) over real world graphs. We introduce new key properties, minimality and support for GKeys, to facilitate the discovery of non-redundant and frequent graph keys. We adopt these notions in GKMiner to implement early termination and pruning of candidate keys. To make GKMiner realizable in practice, we use auxillary structures to summarize input graphs and dependencies among entities to more effectively and efficiently traverse the space of candidate keys. Our results show that GKMiner scales well for increasing key sizes, increasing types, and support levels. Our comparative evaluation shows the effectiveness of using recursive keys in the entity linking task. As next steps, we plan to study the impact of varying graph sizes using synthetic graph data generators. We then plan to extend GKMiner to mine conditional GKeys. Similar to the notion of conditional functional dependencies, conditional constraints provide greater expressiveness for users to specify context when keys should be enforced. This has applications to data linking over knowledge bases to specify partial keys that hold under specific value conditions. Secondly, another avenue of study is to study parallel discovery of GKeys in distributed graphs. This will improve runtime performance, particularly since subgraph isomorphism is costly, and the space of GKeys is large. This will include addressing challenges of workload balancing among worker nodes, communication between workers to handle edges that span across workers, and how a coordinator manages candidate key generation and key validation among workers.

## CRediT authorship contribution statement

**Morteza Alipourlangouri:** Conceptualization, Data curation, Software. **Fei Chiang:** Conceptualization, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgment

This work has been funded by NSERC grant #05711.

## References

[1] X.L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, W. Zhang, From data fusion to knowledge fusion, VLDB 7 (10) (2014) 881–892.
[2] W. Akhtar, A. Cortés-Calabuig, J. Paredaens, Constraints in RDF, in: International Workshop on Semantics in Data and Knowledge Bases, Springer, 2010, pp. 23–39.
[3] M. Atencia, M. Chein, M. Croitoru, J. David, M. Leclère, N. Pernelle, F. Saïs, F. Scharffe, D. Symeonidou, Defining key semantics for the RDF datasets: experiments and evaluations, in: International Conference on Conceptual Structures, 2014, pp. 65–78.
[4] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: An efficient algorithm for discovering functional and approximate dependencies, Comput. J. 42 (2) (1999) 100–111.
[5] P. Buneman, S. Davidson, W. Fan, C. Hara, W.-C. Tan, Keys for XML, Comput. Netw. 39 (5) (2002) 473–487.
[6] W. Fan, P. Lu, Dependencies for graphs, ACM Trans. Database Syst. 44 (2) (2019) 1–40.
[7] M.A. Langouri, A. Mansfield, F. Chiang, Y. Wu, Inconsistency detection with temporal graph functional dependencies, in: 39th IEEE International Conference on Data Engineering, ICDE 2023, IEEE, 2023, pp. 464–476.
[8] W. Fan, X. Wang, Y. Wu, J. Xu, Association rules with graph patterns, Proc. VLDB Endow. 8 (12) (2015) 1502–1513.
[9] W. Fan, W. Fu, R. Jin, P. Lu, C. Tian, Discovering association rules from big graphs, Proc. VLDB Endow. 15 (7) (2022) 1479–1492.
[10] W. Fan, Z. Fan, C. Tian, X.L. Dong, Keys for graphs, Proc. VLDB Endow. 8 (12) (2015) 1590–1601.
[11] H. Ma, M. Alipourlangouri, Y. Wu, F. Chiang, J. Pi, Ontology-based entity matching in attributed graphs, Proc. VLDB Endow. 12 (10) (2019) 1195–1207.
[12] J. Hellings, M. Gyssens, J. Paredaens, Y. Wu, Implication and axiomatization of functional constraints on patterns with an application to the RDF data model, in: Foundations of Information and Knowledge Systems, Springer, 2014, pp. 250–269.
[13] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al., Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia, Semant. Web (2015).
[14] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Vol. 8, Addison-Wesley Reading, 1995.
[15] J. Birnick, T. Bläsius, T. Friedrich, F. Naumann, T. Papenbrock, M. Schirneck, Hitting set enumeration with partial information for unique column combination discovery, Proc. VLDB Endow. 13 (12) (2020) 2270–2283.
[16] Z. Wei, U. Leck, S. Link, Discovery and ranking of embedded uniqueness constraints, Proc. VLDB Endow. 12 (13) (2019) 2339–2352.
[17] M. Atencia, J. David, F. Scharffe, Keys and pseudo-keys detection for web datasets cleansing and interlinking, in: International Conference on Knowledge Engineering and Knowledge Management, Springer, 2012, pp. 144–153.
[18] N. Pernelle, F. Saïs, D. Symeonidou, An automatic key discovery approach for data linking, J. Web Semant. 23 (2013) 16–30.
[19] T. Soru, E. Marx, A.-C. Ngonga Ngomo, ROCKER: A refinement operator for key discovery, in: Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 1025–1033.
[20] W. Fan, Y. Wu, J. Xu, Functional dependencies for graphs, in: SIGMOD International Conference on Management of Data, 2016, pp. 1843–1857.
[21] T. Deng, L. Hou, Z. Han, Keys as features for graph entity matching, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 1974–1977.
[22] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K.W. Hare, J. Hidders, V.E. Lee, B. Li, L. Libkin, W. Martens, et al., Pg-keys: Keys for property graphs, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 2423–2436.
[23] S. Link, Neo4j keys, in: International Conference on Conceptual Modeling, Springer, 2020, pp. 19–33.
[24] P. Skavantzos, K. Zhao, S. Link, Uniqueness constraints on property graphs, in: International Conference on Advanced Information Systems Engineering, Springer, 2021, pp. 280–295.
[25] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, F. Naumann, Scalable discovery of unique column combinations, Proc. VLDB Endow. 7 (4) (2013) 301–312.
[26] Y. Sismanis, P. Brown, P.J. Haas, B. Reinwald, Gordian: efficient and scalable discovery of composite keys, in: Proceedings of the 32nd International Conference on Very Large Data Bases, 2006, pp. 691–702.
[27] S. Baskaran, A. Keller, F. Chiang, L. Golab, J. Szlichta, Efficient discovery of ontology functional dependencies, in: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, in: CIKM '17, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450349185, 2017, pp. 1847–1856, http://dx.doi.org/10.1145/3132847.3132879, https://doi.org/10.1145/3132847.3132879.
[28] Z. Zheng, L. Zheng, M. Alipourlangouri, F. Chiang, L. Golab, J. Szlichta, S. Baskaran, Contextual data cleaning with ontology functional dependencies, J. Data and Information Quality (ISSN: 1936-1955) 14 (3) (2022) http://dx.doi.org/10.1145/3524303, https://doi.org/10.1145/3524303.
[29] L. Noronha, F. Chiang, Discovery of temporal graph functional dependencies, in: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, in: CIKM '21, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450384469, 2021, pp. 3348–3352, http://dx.doi.org/10.1145/3459637.3482087, https://doi.org/10.1145/3459637.3482087.
[30] D. Symeonidou, V. Armant, N. Pernelle, F. Saïs, Sakey: Scalable almost key discovery in RDF data, in: International Semantic Web Conference, Springer, 2014, pp. 33–49.
[31] D. Symeonidou, L. Galárraga, N. Pernelle, F. Saïs, F. Suchanek, VICKEY: mining conditional keys on knowledge bases, in: International Semantic Web Conference, Springer, 2017, pp. 661–677.
[32] W. Fan, C. Hu, X. Liu, P. Lu, Discovering graph functional dependencies, ACM Trans. Database Syst. 45 (3) (2020) 1–42.
[33] M. Alipourlangouri, F. Chiang, KeyMiner: Discovering keys for graphs, in: VLDB Workshop, 2018.
[34] M.H. Namaki, Y. Wu, Q. Song, P. Lin, T. Ge, Discovering graph temporal association rules, in: CIKM, ACM, 2017, pp. 1697–1706.
[35] M. Berlingerio, F. Bonchi, B. Bringmann, A. Gionis, Mining graph evolution rules, in: Machine Learning and Knowledge Discovery in Databases, 2009, pp. 115–130.

[36] S. Castano, A. Ferrara, S. Montanelli, G. Varese, Ontology and instance matching, in: G. Paliouras, C.D. Spyropoulos, G. Tsatsaronis (Eds.), Knowledge-Driven Multimedia Information Extraction and Ontology Evolution: Bridging the Semantic Gap, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-20795-2, 2011, pp. 167–195, http://dx.doi.org/10.1007/978-3-642-20795-2_7.
[37] M. Azmy, P. Shi, J. Lin, I.F. Ilyas, Matching entities across different knowledge graphs with graph embeddings, 2019, CoRR, abs/1903.06607. URL http://arxiv.org/abs/1903.06607.
[38] R. Zass, A. Shashua, Probabilistic graph and hypergraph matching, in: 2008 IEEE Conference on Computer Vision and Pattern Recognition, 2008, pp. 1–8, http://dx.doi.org/10.1109/CVPR.2008.4587500.
[39] W. Fan, X. Wang, Y. Wu, Incremental graph pattern matching, ACM Trans. Database Syst. 38 (3) (2013) 1–47.
[40] J.L. Peterson, A. Silberschatz, Operating System Concepts, Addison-Wesley Longman Publishing Co., Inc., 1985.
[41] IMDB dataset, 2021, URL https://www.imdb.com/interfaces/.
[42] F. Mahdisoltani, J. Biega, F. Suchanek, YAGO3: A knowledge base from multilingual wikipedias, in: CIDR, 2014.
[43] Yago knowledge base, 2022, https://yago-knowledge.org/downloads/yago-3.

**Morteza Alipour Langouri** is a Senior Data Scientist at Citi. His research interests are in data cleaning, temporal graph data cleaning, knowledge base integration, entity identification, and fact checking over knowledge graphs with machine learning. He completed his PhD at McMaster University in 2022.

**Fei Chiang** is an Associate Professor in the Department of Computing and Software (Faculty of Engineering) at McMaster University. She is the Director of the Data Science Research Lab, and a Faculty Fellow at the IBM Centre for Advanced Studies. Her research interest is in data management, spanning data quality, data profiling, temporal graphs, and database systems. She holds four patents for her work in self-managing database systems. She served as an inaugural Associate Director of the MacData Institute, and is an Associate Editor for the ACM Journal of Data and Information Quality. She received her M. Math from the University of Waterloo, and B.Sc and PhD degrees from the University of Toronto, all in Computer Science. She is a recipient of an Ontario Early Researcher Award (2018).