Efficient Algorithms for Association Finding and Frequent Association Pattern Mining

Gong Cheng(✉), Daxin Liu, and Yuzhong Qu

National Key Laboratory for Novel Software Technology,
Nanjing University, Nanjing, China
{gcheng,yzqu}@nju.edu.cn, dxliu@smail.nju.edu.cn

Abstract. Finding associations between entities is a common information need in many areas. It has been facilitated by the increasing amount of graph-structured data on the Web describing relations between entities. In this paper, we define an association connecting multiple entities in a graph as a minimal connected subgraph containing all of them. We propose an efficient graph search algorithm for finding associations, which prunes the search space by exploiting distances between entities computed based on a distance oracle. Having found a possibly large group of associations, we propose to mine frequent association patterns as a conceptual abstract summarizing notable subgroups to be explored, and present an efficient mining algorithm based on canonical codes and partitions. Extensive experiments on large, real RDF datasets demonstrate the efficiency of the proposed algorithms.

Keywords: Association finding · Canonical code · Distance oracle · Frequent association pattern mining · Graph search

1 Introduction

Finding associations between entities has found applications in many areas. For instance, social networking services suggest friends based on known associations between people. Security agents are interested in associations between suspected terrorists. In recent years, the increasing amount of graph-structured data on the Web, like RDF data, has made association finding easier than extracting from Web text [14]. In such a graph describing relations between entities, associations between entities are reflected by paths or subgraphs connecting them. Finding such connections is also an essential component of some semantic search and question answering systems [18].

Existing research efforts mainly focus on finding, ranking, and filtering associations between two entities [2,3,5–8,10,15,20], which are usually defined as paths connecting them in a graph. Given multiple (i.e., two or more) entities, a more general notion of association naturally builds on the paths between all pairs of entities, but requires a more concise structure [4,12,17]. In this work, we define an association connecting multiple entities in a graph as a minimal connected subgraph that contains all of them. Then two challenges arise:

(a) how to efficiently find associations in a possibly very large graph, and (b) how to help users explore a possibly large set of associations that have been found. Both challenges are addressed in this paper. Our contribution is threefold.

- We propose an efficient algorithm for finding associations based on graph search and path merging. To prune the search space, distances between entities are exploited, and a distance oracle is used to achieve a trade-off between time for computing and space for materializing distances.
- To help users explore a large group of associations, complementary to the existing ranking approaches [4,12,17], we propose to identify its notable subgroup(s) that match a common conceptual structure called a frequent association pattern, which provides a high-level abstract of major results. Our efficient algorithm for mining frequent association patterns calculates frequency based on canonical codes of association patterns, and reduces calculations using partitions of associations.
- We carry out extensive experiments based on large, real RDF datasets. The results demonstrate the efficiency of the proposed algorithms.

In this paper, we focus on the efficiency of algorithms for finding associations and mining frequent association patterns. The effectiveness of using frequent association patterns for exploring associations between two entities has been demonstrated in [6]. The effectiveness in a multiple-entity setting will be empirically tested in future work.

The remainder of this paper is structured as follows. Section 2 provides preliminaries. Sections 3 and 4 introduce our algorithms for finding associations and mining frequent association patterns, respectively. Section 5 presents experiments. Section 6 discusses related work. Section 7 concludes the paper with future work.

2 Preliminaries

We deal with a directed unweighted *entity-relation graph* $G = \langle E, A, R, l \rangle$ characterizing binary relations over entities, where

- E is a set of entities as vertices,
- A is a set of arcs, each arc $a \in A$ directed from its tail vertex $t(a) \in E$ to its head vertex $h(a) \in E$,
- R is a set of binary relations on entities, and
- $l : A \mapsto R$ labels each arc $a \in A$ with a relation $l(a) \in R$.

Let C be the set of all classes. For each entity $e \in E$, let $T(e) \subseteq C$ be e's types, and we assume that each entity has at least one type, i.e., $T(e) \neq \emptyset$. Figure 1 shows an entity-relation graph to be used as a running example in this paper. An RDF graph (i.e., a set of RDF triples) can be regarded as an entity-relation graph if considering only the triples connecting two entities; T is given by the `rdf:type` property. In this paper, we will stick to the above graph notation rather than RDF because our approach is not specific to RDF but also applies to other kinds of graph-structured data.

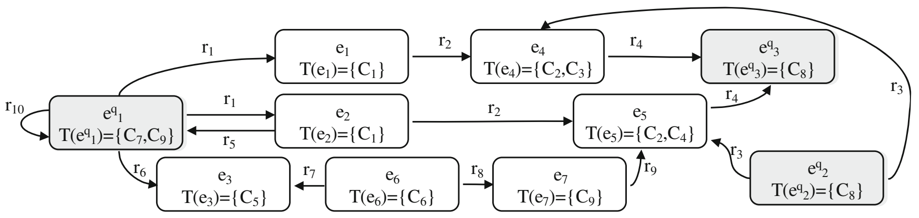


Fig. 1. An example entity-relation graph, with three query entities in grey.

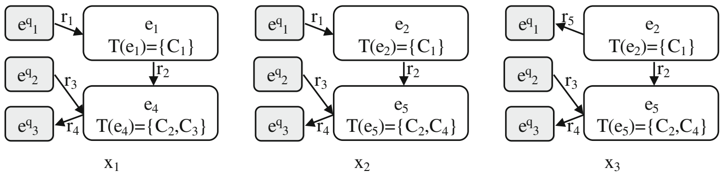


Fig. 2. Three associations connecting e_1^q, e_2^q, and e_3^q.

Given a set of n *query entities* $e_1^q, \ldots, e_n^q \in E$, an *association* x connecting $e_1^q, \ldots, e_n^q$ is a minimal subgraph of G that contains all of them and is connected; no proper subgraph of it also has these properties. Therefore, the underlying graph of x is a tree (i.e., having no parallel edges, loops, or cycles), and the leaves only come from query entities; otherwise x would not be minimal. For consistency, e_1^q is always designated as the root of x. Figure 2 illustrates three associations connecting the three query entities in the running example.

Note that in this paper, the arcs in a path or in a rooted tree are not required to all go the same direction, since an arc a directed from $t(a)$ to $h(a)$ labeled with a relation $l(a) = r$ can be equivalently treated as an arc directed from $h(a)$ to $t(a)$ labeled with a relation $\widehat{r}$ that represents the inverse of r. For the same reason, later in our algorithms, every arc can be traversed in both directions in graph search.

The *diameter* of an association x, denoted by $diam(x)$, is the greatest distance between any pair of entities in x. Given a *diameter constraint* λ, a *valid association* has a diameter of λ or less. For instance, given $\lambda = 3$, Fig. 2 shows all the valid associations connecting the three query entities in the running example; all of them have a diameter of 3. An *invalid association* has a diameter larger than λ. We will focus on valid associations because such shorter-distance associations usually represent stronger connections between entities and thus are more attractive to users.

An *association pattern* matched by an association x is a directed graph obtained by replacing each non-query entity in x with one of its types. For instance, x_1 and x_2 in Fig. 2 match z_1 in Fig. 3; x_1 also matches z_2. Since an association is tree-structured and the leaves only come from query entities, an association pattern also has these properties, and e_1^q is designated as its root for consistency.

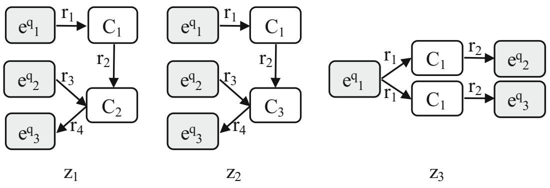


Fig. 3. Three association patterns.

3 Association Finding

Given an entity-relation graph G and a diameter constraint λ, we aim to find all the valid associations connecting a set of n query entities $e^q_1, \ldots, e^q_n$. Firstly, we present a basic algorithm for finding valid associations based on graph search and path merging. Then, we prune the search space by exploiting distances between vertices. Finally, to achieve a trade-off between time for computing and space for materializing distances, we discuss the use of distance oracles.

3.1 A Basic Algorithm

Our basic algorithm is inspired by the following theorem.

Theorem 1. *An association x connecting a set of query entities can be decomposed into a set of (possibly overlapping) paths of length $\left\lfloor \frac{diam(x)+1}{2} \right\rfloor$ or less that have query entities as their start vertices and have a common end vertex.*

For instance, x_1 in Fig. 2, with $diam(x_1) = 3$, can be decomposed into three paths of length $\left\lfloor \frac{3+1}{2} \right\rfloor = 2$ or less: $e^q_1 r_1 e_1$, $e^q_2 r_3 e_4 \widehat{r_2} e_1$, and $e^q_3 \widehat{r_4} e_4 \widehat{r_2} e_1$; all of them start from query entities and have e_1 as a common end vertex.

Proof. Let p be a longest path in x, having a length of $diam(x)$. Let e' be an entity in the middle of p, i.e., the two paths p_1 and p_2 connecting the start and end vertex of p to e' have a length of $\left\lfloor \frac{diam(x)+1}{2} \right\rfloor$ or $\left\lfloor \frac{diam(x)+1}{2} \right\rfloor - 1$. Then for every leaf e of x, the path connecting e to e' must have a length of $\left\lfloor \frac{diam(x)+1}{2} \right\rfloor$ or less; otherwise we can merge such a path with p_1 or p_2 to obtain a path longer than $diam(x)$, which contradicts that the diameter of x is $diam(x)$. Therefore, x can be decomposed into a set of paths of length $\left\lfloor \frac{diam(x)+1}{2} \right\rfloor$ or less, each connecting a leaf of x (which is a query entity) to e'.

Following this theorem, we develop Algorithm 1 for finding all the valid associations by searching for and merging paths. Specifically, all the paths of length $\left\lfloor \frac{\lambda+1}{2} \right\rfloor$ or less starting from each query entity are found by searching G in a breadth-first manner (line 3–4). For instance, when $\lambda = 3$, starting from e^q_1 in Fig. 1, four paths of length 1 and four paths of length 2 are found:

$$\begin{aligned} P_1 = \{ & e^q_1 r_1 e_1,\ e^q_1 r_1 e_2,\ e^q_1 \widehat{r_5} e_2,\ e^q_1 r_6 e_3, \\ & e^q_1 r_1 e_1 r_2 e_4,\ e^q_1 r_1 e_2 r_2 e_5,\ e^q_1 \widehat{r_5} e_2 r_2 e_5,\ e^q_1 r_6 e_3 \widehat{r_7} e_6 \}\,. \end{aligned} \tag{1}$$

Algorithm 1. A Basic Algorithm for Association Finding

```
Data: An entity-relation graph $G$, a set of $n$ query entities $e^q_1, \ldots, e^q_n$, and a
      diameter constraint $\lambda$.
Result: A set of valid associations connecting $e^q_1, \ldots, e^q_n$.
1  $X = \emptyset$ ;                                    /* a set of associations */
2  $codes = \emptyset$ ;              /* a set of canonical codes for associations */
3  for $i = 1$ to $n$ do
4      $P_i$ = the set of all paths of length $\lfloor \frac{\lambda+1}{2} \rfloor$ or less starting from $e^q_i$ found by
       searching $G$ in a breadth-first manner;
5  foreach $\langle p_1, \ldots, p_n \rangle \in (P_1 \times \cdots \times P_n)$ do
6      if $p_1, \ldots, p_n$ have a common end vertex then
7          Merge $p_1, \ldots, p_n$ to form a connected subgraph $x$ of $G$;
8          if $x$ is minimal then
               /* $x$ is minimal if its underlying graph is a tree, and the
                  leaves only come from $e^q_1, \ldots, e^q_n$.                  */
9              if $diam(x) \leq \lambda$ then
10                 $code(x)$ = the canonical code of $x$;
11                 if $code(x) \notin codes$ then
12                     Add $code(x)$ to $codes$;
13                     Add $x$ to $X$;
14 return $X$
```

Then, all possible combinations of such paths are examined (line 5–13); each combination consists of one path starting from each query entity, i.e., one from P_1, one from P_2, ..., one from P_n. If all the paths in a combination have a common end vertex (e.g., $e^q_1 r_1 e_1$, $e^q_2 r_3 e_4 \widehat{r_2} e_1$, and $e^q_3 \widehat{r_4} e_4 \widehat{r_2} e_1$ in Fig. 1), they will be merged into a subgraph x of G (e.g., x_1 in Fig. 2) that is potentially a valid association to be found (line 6–7). However, before adding x to the results X (line 13), it has to satisfy three requirements.

Firstly, x should be minimal; that is, its underlying graph is a tree, and the leaves only come from query entities (line 8). These tests can be carried out within a single depth-first search of x.

Secondly, x should be valid, i.e., $diam(x) \leq \lambda$ (line 9). This test is needed because when λ is odd, it is possible that x is formed by merging paths of length $\lfloor \frac{\lambda+1}{2} \rfloor = \frac{\lambda+1}{2}$ so that $diam(x) = \lambda + 1 > \lambda$.

Thirdly, the same association should not be added to X multiple times. For instance, x_1 in Fig. 2 can be formed twice by merging the paths in two different combinations: one with e_1 as a common end vertex and the other with e_4. To avoid such duplicates, we generate a *canonical code* for x (line 10), denoted by $code(x)$, so that two associations will have the same canonical code if and only if they are *isomorphic* to each other, i.e., they have the same set of entities as vertices and there is a bijection between their arcs that preserves adjacency and arc labels. If it is the first time $code(x)$ is seen, x will be added to X (line 11–13).

There have been various ways of defining and generating canonical codes for trees [11], assuming a total order ($\preceq$) on each set of sibling vertices. We adopt the following recursive definition, and implement $\preceq$ by the alphabetical order of entity identifiers (e.g., URIs).

– For a tree T with a single vertex e, we define

$$code(T) = e\$\,, \tag{2}$$

where \$ is a special symbol not in the alphabet for naming entities and relations.
– For a tree T with more than one vertex, assuming its root is e and the arcs connecting e to its children $e_1, \ldots, e_k$ (subject to $e_1 \preceq \cdots \preceq e_k$) are labeled with relations $r_1, \ldots, r_k$, respectively, we define

$$code(T) = er_1 code(T_1) \cdots r_k code(T_k)\$\,, \tag{3}$$

where $T_1, \ldots, T_k$ are the subtrees rooted at $e_1, \ldots, e_k$, respectively.

Such a code can be generated for x via a depth-first search of x. For instance, for x_1 in Fig. 2 with e_1^q always designated as its root, assuming $e_2^q \preceq e_3^q$, we have

$$code(x_1) = e_1^q r_1 e_1 r_2 e_4 \widehat{r_3} e_2^q \$ r_4 e_3^q \$\$\$\$. \tag{4}$$

Let $\varDelta$ be the maximum of the degrees of vertices in G. In the algorithm, the number of paths that can be found from a query entity is bounded by $O(\varDelta^{\lfloor \frac{\lambda+1}{2} \rfloor})$. Given n query entities, there are $O(\varDelta^{\lfloor \frac{\lambda+1}{2} \rfloor n})$ combinations of paths to examine; in practice we can index paths by their end vertices to significantly improve the performance. The time for checking one combination of paths for the three requirements of a valid association is linear with its size, which is bounded by $O(n\lambda)$. Overall, the algorithm takes $O(\varDelta^{\lfloor \frac{\lambda+1}{2} \rfloor n} n\lambda)$ time, but n and λ are both very small in practice.

3.2 Distance-Based Search Space Pruning

To improve the performance of Algorithm 1, we notice that some paths found in graph search will not be merged into any valid association. For instance, when $\lambda = 3$, among the eight paths in P_1 as shown in Eq. (1), $e_1^q r_6 e_3$ and $e_1^q r_6 e_3 \widehat{r_7} e_6$ eventually do not take part in any valid association in Fig. 2. If we can prune the search space to exclude such paths, graph search will end earlier (line 4) and there will be much fewer combinations of paths to be examined (line 5–13), so that the performance of the algorithm can be improved.

We prune the search space by exploiting distances between entities in the entity-relation graph G. Let *dist* return the distance between two entities in G. For instance, in Fig. 1, we have $dist(e_1^q, e_3) = 1$ and $dist(e_2^q, e_3) = 4$. When searching G for the set of paths P_i starting from a query entity e_i^q and arriving at an entity e via a path $p_{e_i^q e}$ from e_i^q to e, the search space may then be pruned depending on the distances between e and other query entities, i.e., $dist(e_j^q, e)$ for $j \neq i$.

Specifically, if $dist(e_j^q, e) > \left\lfloor \frac{\lambda+1}{2} \right\rfloor$ for any other query entity e_j^q $(j \neq i)$, $p_{e_i^q e}$ can be excluded from P_i *safely* (i.e., not affecting the final results X) because it will not take part in any valid association since P_j is not likely to contain a path from e_j^q to e of length $\left\lfloor \frac{\lambda+1}{2} \right\rfloor$ or less. For instance, given $\lambda = 3$, when searching the graph in Fig. 1 starting from e_1^q and arriving at e_3 via the path $e_1^q r_6 e_3$, this path will be excluded from P_1 because $dist(e_2^q, e_3) = 4 > 2 = \left\lfloor \frac{3+1}{2} \right\rfloor$.

Further, let $ln(p)$ be the length of a path p. If $ln(p_{e_i^q e}) + dist(e_j^q, e) > 2\left\lfloor \frac{\lambda+1}{2} \right\rfloor$ for any other query entity e_j^q $(j \neq i)$, which implies $dist(e_j^q, e) > \left\lfloor \frac{\lambda+1}{2} \right\rfloor$ since $ln(p_{e_i^q e}) \leq \left\lfloor \frac{\lambda+1}{2} \right\rfloor$, we can safely exclude from P_i not only $p_{e_i^q e}$ but also all the paths that extend $p_{e_i^q e}$ (i.e., having $p_{e_i^q e}$ as a prefix); in other words, the entire branch of search stemming from $p_{e_i^q e}$ can be pruned. For instance, given $\lambda = 3$, when searching the graph in Fig. 1 starting from e_1^q and arriving at e_3 via the path $e_1^q r_6 e_3$, we will not only exclude this path from P_1 but also prune the branch of search stemming from it because $ln(e_1^q r_6 e_3) + dist(e_2^q, e_3) = 1 + 4 = 5 > 2\left\lfloor \frac{3+1}{2} \right\rfloor$; as a result, the path $e_1^q r_6 e_3 \widehat{r_7} e_6$ will be implicitly excluded from P_1. We prove the safeness by showing that any path $p_{e_i^q e'}$ from e_i^q to an entity e' that extends $p_{e_i^q e}$ will not take part in any valid association. Specifically, $p_{e_i^q e'}$ is composed of $p_{e_i^q e}$ from e_i^q to e and $p_{ee'}$ from e to e'. If it can be merged with some path $p_{e_j^q e'} \in P_j$ $(j \neq i)$ from e_j^q to e' into a valid association, we will have $ln(p_{e_j^q e'}) \leq \left\lfloor \frac{\lambda+1}{2} \right\rfloor$ and

$$\begin{aligned} 2\left\lfloor \frac{\lambda+1}{2} \right\rfloor &= \left\lfloor \frac{\lambda+1}{2} \right\rfloor + \left\lfloor \frac{\lambda+1}{2} \right\rfloor \\ &\geq ln(p_{e_i^q e'}) + ln(p_{e_j^q e'}) \\ &= ln(p_{e_i^q e}) + ln(p_{ee'}) + ln(p_{e_j^q e'}) \\ &\geq ln(p_{e_i^q e}) + dist(e_j^q, e)\,, \end{aligned} \tag{5}$$

which contradicts $ln(p_{e_i^q e}) + dist(e_j^q, e) > 2\left\lfloor \frac{\lambda+1}{2} \right\rfloor$.

3.3 Distance Computation

The above pruning strategy requires knowing distances between entities. When the entity-relation graph is large, e.g., consisting of millions of vertices and billions of arcs, obtaining distances will be nontrivial. On the one hand, online computing distances would be time-consuming and lead to unacceptable latency. On the other hand, materializing offline computed distances between all pairs of entities would be a challenge. To achieve a trade-off between time for computing and space for materializing distances, we turn to distance oracles [16].

A *distance oracle* is a data structure that, after preprocessing a graph, allows for fast distance computation. Specifically, the graph is offline processed to compute certain information (e.g., distances between each vertex and some landmark vertices) to be materialized in a distance oracle; its size is usually much smaller than the size of materializing distances between all pairs of vertices. By using a distance oracle, computing the distance between two vertices can be reasonably fast, though not as fast as looking up a materialized distance.

There are two types of distance oracles: exact and approximate. Given two vertices between which the distance is d, an *exact distance oracle* will return d, whereas an *approximate distance oracle* will return a value that is in the range of $[d, \alpha d + \beta]$ where $\alpha \geq 1$ and $\beta \geq 0$, which is said to have stretch (α, β). Different approximate distance oracles have different trade-offs between stretch, size, and time. Practical approximate distance oracles usually have stretch $\alpha = 2$ or $\alpha = 3$. However, such a distance oracle is not particularly useful for small-world graphs in which distances between vertices are typically very small [16]. As we will see in Sect. 5.1, some widely used entity-relation graphs are exactly small-world graphs. Therefore, we choose to implement a state-of-the-art exact distance oracle [1], to be used in distance-based pruning.

4 Frequent Association Pattern Mining

Having found a possibly large group of associations, we aim to identify its notable subgroup(s) that match a common conceptual structure, i.e., a frequent association pattern, to provide a high-level abstract of major results. Specifically, given a group of associations X, the *frequency* of an association pattern z, denoted by $f_X(z)$, is the number of associations in X that match z. Given a threshold $\tau \in [0, 1]$, we aim to find all the *frequent association patterns* z for which $\frac{f_X(z)}{|X|} \geq \tau$. Note that existing solutions to frequent tree pattern mining [11] do not apply here because their resulting subtrees may not contain all the query entities. In the following, we firstly present a basic algorithm. Then we improve its performance by partitioning X.

4.1 A Basic Algorithm

The idea is to firstly, for each association in X, enumerate all the association patterns it matches; for instance, x_1 in Fig. 2 matches z_1 and z_2 in Fig. 3. Then we calculate the frequency of each association pattern and identify frequent ones; to this end, the main problem is to judge whether two association patterns enumerated for different associations are isomorphic to each other. Since an association pattern is tree-structured, we intend to generate a canonical code for each enumerated association pattern by reusing the way of defining and generating canonical codes presented in Sect. 3.1, and then count the occurrence of each canonical code as the frequency of the corresponding association pattern.

Recall that in Sect. 3.1, the definition of canonical code relies on a predefined total order ($\preceq$) on each set of sibling vertices; there, we implement $\preceq$ by the alphabetical order of entity identifiers, considering that sibling vertices in an association are always different entities with different identifiers. However, if sibling entities in an association have a common type, the corresponding sibling vertices in an association pattern will represent the same class; for instance, in Fig. 3, the two children of e_1^q in z_3 both represent C_1. Hence, the alphabetical order of entity and class identifiers fails to give a total order on such a set of sibling vertices. If we still use this order and break ties arbitrarily, different canonical codes may be generated for isomorphic association patterns, leading to incorrect calculation of frequency. For instance, the canonical code for z_3 in Fig. 3 could be

$$\begin{aligned} & e_1^q r_1 C_1 r_2 e_2^q \$\$ r_1 C_1 r_2 e_3^q \$\$\$ \\ \text{or } & e_1^q r_1 C_1 r_2 e_3^q \$\$ r_1 C_1 r_2 e_2^q \$\$\$ \,, \end{aligned} \tag{6}$$

depending on how to order the two children of e_1^q.

To obtain a unique canonical code, a less efficient solution is to generate codes in all possible orders and choose the lexicographically smallest one [11]. Differently, we propose a more efficient solution that directly generates a unique code by implementing $\preceq$ in a different way that exploits query entities. Specifically, instead of directly ordering sibling vertices by their identifiers (which may represent the same class), for each sibling vertex v that is not a query entity, we choose a query entity as its *proxy* to be ordered by entity identifiers, which is the one with the alphabetically smallest entity identifier in the subtree rooted at v. Since subtrees rooted at sibling vertices contain different sets of query entities, the proxies chosen are different. This successfully gives a total order on each set of sibling vertices, and thus ensures a unique canonical code for isomorphic association patterns. For instance, assuming e_2^q alphabetically precedes e_3^q, the unique canonical code for z_3 in Fig. 3 will be

$$\begin{aligned} & e_1^q r_1 C_1 r_2 e_2^q \$\$ r_1 C_1 r_2 e_3^q \$\$\$ \\ \text{but not } & e_1^q r_1 C_1 r_2 e_3^q \$\$ r_1 C_1 r_2 e_2^q \$\$\$ \,, \end{aligned} \tag{7}$$

because the proxy for the upper child of e_1^q in Fig. 3 is e_2^q, which alphabetically precedes e_3^q, the proxy for the lower child of e_1^q. Proxies for all the vertices in an association pattern z can be found within a single depth-first search of z.

The size of an association is bounded by $O(n\lambda)$. Let γ be the maximum number of types that an entity can have. An association can match $O(\gamma^{n\lambda})$ association patterns, and thus $O(|X|\gamma^{n\lambda})$ canonical codes will be generated. Generating one canonical code takes $O(n\lambda)$ time, plus $O(n\lambda)$ time for finding proxies. Overall, the algorithm takes $O(|X|\gamma^{n\lambda}n\lambda)$ time to generate all the canonical codes to be counted, but γ, n, λ are all very small in practice.

4.2 Partitioning-Based Performance Improvement

Enumerating association patterns and generating canonical codes for them can be time-consuming. To improve the performance, we aim to divide X into mutually disjoint partitions, and ensure that only the associations in the same partition can match a common association pattern. Then, when mining frequent association patterns, we can ignore partitions containing fewer than $\tau|X|$ associations, without spending time processing association patterns they match.

We observe that two associations can match a common association pattern only if they: (a) consist of the same number of vertices, and (b) have the same set of arc labels (i.e., relations). We divide X based on a combination of these two metrics. For instance, x_1 and x_2 in Fig. 2 will be put in the same partition because both of them consist of five vertices and their arc labels are both $\{r_1, r_2, r_3, r_4\}$, whereas x_3 is in a different partition because its arc labels are $\{r_2, r_3, r_4, r_5\}$.

5 Experiments

We tested the performance of the proposed algorithms on an E3-1226 v3 with 24GB memory for JVM. Entity-relation graphs and entities' types were stored in memory. Distance oracles were stored in a MySQL database on disk.

5.1 Datasets and Test Queries

Datasets. Experiments were conducted on two widely used RDF datasets.

– LinkedMDB[1] provided RDF data about movies and related entities like actors and directors. After filtering out RDF triples involving literals or `rdf:type`, an entity-relation graph was obtained, consisting of 1,327,069 entities as vertices and 2,132,796 arcs. Entities' types were derived from RDF triples involving `rdf:type`.
– DBpedia[2] provided encyclopedic RDF data extracted from Wikipedia. After filtering out RDF triples involving literals, an entity-relation graph was obtained from the Mapping-based Properties dataset, consisting of 4,337,485 entities as vertices and 15,007,564 arcs. Entities' types were derived from the Mapping-based Types dataset.

For entities having no type information, `owl:Thing` was added to be their type.

To characterize the two entity-relation graphs, we randomly selected 10,000 pairs of entities from each graph, and tested whether they were connected by paths and if so, calculated the distance between them. As shown in Table 1, in LinkedMDB, most pairs of entities (77.20 %) were connected, and their average and median distances were 6.61 and 7, respectively, showing the small-world effect, which was even more pronounced on DBpedia. The results revealed two findings.

Table 1. Distance between entities

	% of connected entities	Distance between connected entities	
		Average	Median
LinkedMDB	77.20 %	6.61	7
DBpedia	96.41 %	5.06	5

[1] http://www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-latest-dump.zip.
[2] http://wiki.dbpedia.org/Downloads2015-04.

– An exact (not approximate) distance oracle was needed for effective distance-based pruning on such small-world graphs as discussed in Sect. 3.3.
– The diameter constraint had to be set to a small value (≤ 4), because larger values would require searching almost the entire entity-relation graph, and could find too many paths and associations to fit in memory.

Test Queries. Test queries were constructed under different settings of diameter constraint (λ) and number of query entities (n). For each combination of $\lambda \in \{2, 4\}$ and $n \in \{2, 3, 4, 5\}$, we randomly selected 1,000 sets of n query entities from each of the two entity-relation graphs as test queries.

5.2 Association Finding

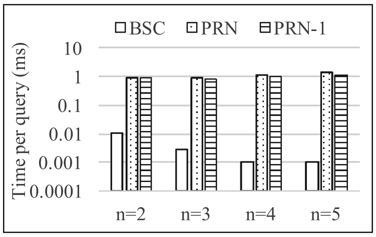
Algorithms. Three algorithms for association finding were tested:

– BSC: the basic algorithm described in Sect. 3.1, which can be regarded as an extension of the existing bi-directional BFS algorithm for finding paths between two entities [10],
– PRN: the improved algorithm using distance-based search space pruning described in Sects. 3.2 and 3.3, and
– PRN-1: a variant of PRN that would not try to prune the search space at the last level of search, and thus might exclude fewer paths than PRN but could reduce the number of distance computations, achieving a different trade-off.

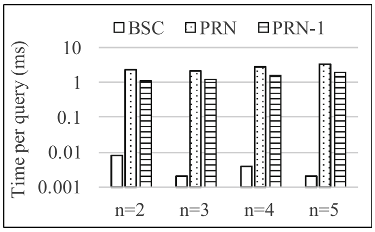
In PRN and PRN-1, the distance between two vertices would be cached in memory after being computed for the first time. However, to avoid distorting the results of performance tests, the cache would be cleared after every single run of an algorithm on a test query.

Results. We ran each algorithm five times on each test query, and took the median running time. Then we calculated the average running time per query used by each algorithm on all the test queries under each setting of λ and n.

As shown in Fig. 4 on a logarithmic scale, when $\lambda = 2$, all the three algorithms were very fast on both datasets, using not more than 4ms per query. PRN and

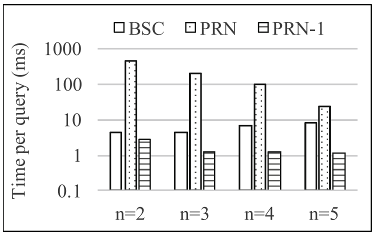


(a) LinkedMDB

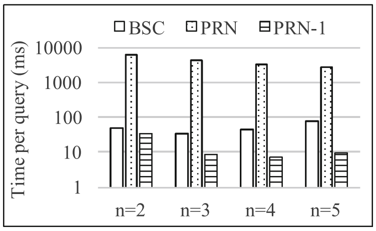


(b) DBpedia

Fig. 4. Running time of association finding under $\lambda = 2$.



(a) LinkedMDB



(b) DBpedia

Fig. 5. Running time of association finding under $\lambda = 4$.

Table 2. Number of distance computations

Dataset	Algorithm	$\lambda = 2$				$\lambda = 4$			
		$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
LinkedMDB	PRN	2.0	3.0	4.0	5.0	3,055.3	1,525.6	836.0	144.5
	PRN-1	2.0	3.0	4.0	5.0	2.9	4.0	4.8	5.7
DBpedia	PRN	2.2	3.0	4.0	5.0	32,530.2	24,061.5	19,057.1	15,346.5
	PRN-1	2.0	3.0	4.0	5.0	5.7	8.9	9.1	13.0

PRN-1 were relatively slow because the search space was very small when $\lambda = 2$, so that distance computation for pruning took more time than it saved.

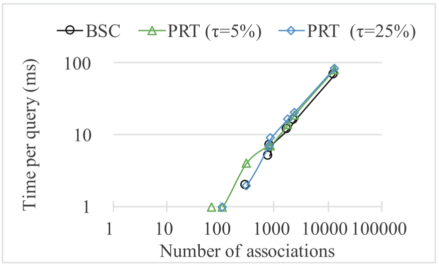
Distance-based pruning proved to be effective when the search space became large. As shown in Fig. 5 on a logarithmic scale, when $\lambda = 4$, PRN-1 used not more than 34ms per query, being 55 %–548 % faster than BSC on LinkedMDB, and 40 %–712 % faster on DBpedia. The difference rose when increasing n because given a larger number of query entities (i.e., n), more distances could be exploited in graph search and the search space would be more likely to be pruned.

PRN was slower than BSC and PRN-1 because, compared with PRN-1, it also tried to prune the search space at the last level of search, which required computing distances between much more pairs of entities, as shown in Table 2. However, each of those computations could exclude at most one path, as opposed to a possibly large branch of search stemming from a path when pruning at earlier levels of search, thereby being cost-ineffective.

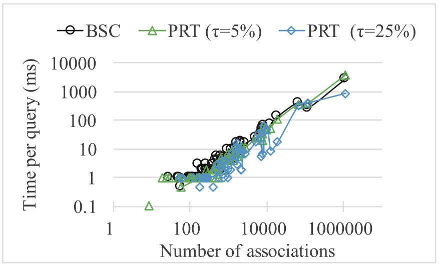
5.3 Frequent Association Pattern Mining

Approaches. Two algorithms for frequent association pattern mining were tested:

– BSC: the basic algorithm described in Sect. 4.1, whose running time was independent of the relative frequency threshold (τ), and
– PRT: the improved algorithm using partitions described in Sect. 4.2, with $\tau = 5\,\%$ or $\tau = 25\,\%$.



(a) LinkedMDB



(b) DBpedia

Fig. 6. Running time of frequent association pattern mining under $\lambda = 4$.

Results. We ran each algorithm five times on each test query that resulted in at least two associations when $\lambda = 4$, and took the median running time. Then we calculated the average running time per query used by each algorithm on all the test queries resulting in the same number of associations.

As shown in Fig. 6 on a log-log scale, all the algorithms were reasonably fast on both datasets for 10,000 or fewer associations, using not more than 21ms and 65ms per query on LinkedMDB and DBpedia, respectively. For larger sets of associations on DBpedia, hundreds or thousands of milliseconds was used. Actually, the reported running time had the potential to be reduced by easily parallelizing the algorithms, e.g., enumerating association patterns for different associations in parallel, and generating canonical codes for different association patterns in parallel.

When the number of associations was small, the difference between BSC and PRT was not significant. On most queries resulting in 5,000 or more associations on DBpedia, PRT was 13 %–722 % faster than BSC when $\tau = 25\,\%$, showing the effectiveness of using partitions. However, PRT was slower than BSC on some queries particularly when $\tau = 5\,\%$ because only very small partitions could be occasionally ignored so that computing partitions took more time than it saved.

5.4 Discussion

In the experiments, we found two limitations of our approach.

Firstly, to find associations, although PRN-1 was very fast when $\lambda \in \{2, 4\}$, using not more than 34ms per query on two fairly large datasets, it frequently used the memory up when we tried to increase λ to 6. That was due to the small-world effect; there were indeed quite many associations to find when $\lambda = 6$. If some of such long-distance associations were believed to be useful according to a certain ranking criterion, graph search could leverage the criterion to prune the search space and return not all but top-ranked associations. However, that would be a different research problem having its own applications [13].

Secondly, to mine frequent association patterns, associations were partitioned so that it was possible to avoid enumerating association patterns for some associations and generating canonical codes for them. However, to put an association into the right partition according to the number of its vertices and the set of its arc labels, the running time was linear with its size, being asymptotically equivalent to the time for generating its canonical code. Therefore, partitioning could not fundamentally improve the performance of the mining algorithm, and did not appear to be consistently superior to the basic algorithm in the experiment. One possibly essential improvement would be to integrate frequent association pattern mining into association finding. For instance, it would be interesting to combine our approach with the techniques in [19].

6 Related Work

Numerous research efforts have been made to find associations between *two* entities, and they define association in different ways [3,7,15]. In a seminal work [3], four types of associations are discussed. Among others, an association between two entities can be a *path* in an entity-relation graph that connects the two entities. Although recent attempts propose to merge certain paths to better explain relatedness between two entities [7,15], the path-based straightforward definition is adopted by most of the subsequent researches, which mainly focus on two problems: how to efficiently find all the paths of a limited length between two entities [10], and how to help users explore such a possibly very large set of paths [2,5,6,8,20]. Concerning the latter problem, one line of work studies the ranking of paths to show users more important paths earlier [2,5]. Complementary to that, other solutions allow users to filter paths by specifying keywords appearing on the paths [20], relations and classes of entities contained in the paths [8], or frequent patterns of the paths [6].

Different from the above efforts, in this work we aim to find associations between *multiple* (i.e., two or more) entities in an entity-relation graph. It goes beyond simply finding paths between all pairs of entities [9], but requires consolidating those paths into concise structures. For instance, in [4,12,17], their goal is to find an optimal association between multiple entities that is a subgraph connecting those entities via a limited number of other entities and maximizing a "goodness" function. In [13], the goal is to find top-k minimum-cost Steiner trees connecting those entities. Differently, we deal with unweighted graphs because we aim to find not top-ranked associations but all the associations having a limited diameter, and then identify their frequent patterns to provide a conceptual abstract of them. This extends our previous work on mining frequent patterns of paths connecting two entities [6], and complements the existing approaches to ranking associations between multiple entities [4,12,13,17].

Compared with a recent work on mining frequent patterns of associations connecting multiple entities in an entity-relation graph [19], our work has made two technical advances. Firstly, in [19], associations are efficiently found by merging paths of a limited length that are materialized in an index. However, it has two limitations: (a) the size of that index increases exponentially with the length of path, and may not be affordable for large datasets and long paths, and (b) when a larger diameter constraint is given, the index may have to be rebuilt to include longer paths. By comparison, to achieve a trade-off between time for computing and space for materializing, we materialize not paths but only a distance oracle which has a fixed, affordable size; using that, paths not taking part in any valid association can be efficiently pruned. Besides, once a distance oracle is built, it can work with arbitrarily large diameter constraints. Secondly, in [19], an association pattern (which is tree-structured) is formed by merging path patterns. That may result in structurally isomorphic association patterns that trivially differ in the designation of root. We eliminate such duplicates by defining and generating a canonical code for each pattern.

7 Conclusion

We have presented efficient algorithms for finding associations connecting a set of query entities in graph-structured data, and mining their frequent association patterns to summarize major results for exploration. Experiment results show that our algorithms are reasonably fast on large, real datasets. They can find applications in many areas where finding associations is a common information need. The novel idea of using a distance oracle to compute distances for pruning the search space may also benefit the study of other research problems such as semantic search and query processing over graph-structured data.

As discussed at the end of the experiments, to further improve the performance of our algorithms, one promising direction is to incorporate ranking criteria (if any) into graph search, and to embed frequent association pattern mining in association finding. This will be our future work. Besides, we have found that sometimes a large number of frequent association patterns can be found, some of which have overlapping meanings and some are not so meaningful to users. It inspires us to consider selecting appropriate ones from all the frequent association patterns, to help users effectively explore associations.

Acknowledgments. This work was supported by the 863 Program under Grant 2015AA015406, the NSFC under Grant 61572247 and 61223003, and the Fundamental Research Funds for the Central Universities.

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: 2013 ACM SIGMOD International Conference on Management of Data, pp. 349–360 (2013)
2. Anyanwu, K., Maduko, A., Sheth, A.: SemRank: ranking complex relationship search results on the semantic web. In: 14th International Conference on World Wide Web, pp. 117–127 (2005)
3. Anyanwu, K., Sheth, A.: ρ-queries: enabling querying for semantic associations on the semantic web. In: 12th International Conference on World Wide Web, pp. 690–699 (2003)
4. Chen, C., Wang, G., Liu, H., Xin, J., Yuan, Y.: SISP: a new framework for searching the informative subgraph based on PSO. In: 20th ACM International Conference on Information and Knowledge Management, pp. 453–462 (2011)

5. Chen, N., Prasanna, V.K.: Learning to rank complex semantic relationships. Int. J. Semant. Web Inf. Syst. **8**(4), 1–19 (2012)
6. Cheng, G., Zhang, Y., Qu, Y.: Explass: exploring associations between entities via top-K ontological patterns and facets. In: Mika, P., et al. (eds.) ISWC 2014, Part II. LNCS, vol. 8797, pp. 422–437. Springer, Heidelberg (2014)
7. Fang, L., Das Sarma, A., Yu, C., Bohannon, P.: REX: explaining relationships between entity pairs. Proc. VLDB Endow. **5**(3), 241–252 (2011)
8. Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T.: RelFinder: revealing relationships in RDF knowledge bases. In: Chua, T.-S., Kompatsiaris, Y., Mérialdo, B., Haas, W., Thallinger, G., Bailer, W. (eds.) SAMT 2009. LNCS, vol. 5887, pp. 182–187. Springer, Heidelberg (2009)
9. Heim, P., Lohmann, S., Stegemann, T.: Interactive relationship discovery via the semantic web. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010, Part I. LNCS, vol. 6088, pp. 303–317. Springer, Heidelberg (2010)
10. Janik, M., Kochut, K.J.: BRAHMS: a workbench RDF store and high performance memory system for semantic association discovery. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 431–445. Springer, Heidelberg (2005)
11. Jiménez, A., Berzal, F., Cubero, J.-C.: Frequent tree pattern mining: a survey. Intell. Data Anal. **14**(6), 603–622 (2010)
12. Kasneci, G., Elbassuoni, S., Weikum, G.: MING: mining informative entity relationship subgraphs. In: 18th ACM Conference on Information and Knowledge Management, pp. 1653–1656 (2009)
13. Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F.M., Weikum, G.: STAR: steiner-tree approximation in relationship graphs. In: IEEE 25th International Conference on Data Engineering, pp. 868–879 (2009)
14. Luo, G., Tang, C., Tian, Y.-L.: Answering relationship queries on the web. In: 16th International Conference on World Wide Web, pp. 561–570 (2007)
15. Pirró, G.: Explaining and suggesting relatedness in knowledge graphs. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 622–639. Springer, Berlin (2015)
16. Sommer, C.: Shortest-path queries in static networks. ACM Comput. Surv. **46**(4), 45 (2014)
17. Tong, H., Faloutsos, C.: Center-piece subgraphs: problem definition and fast solutions. In: 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 404–413 (2006)
18. Tran, T., Cimiano, P., Rudolph, S., Studer, R.: Ontology-Based interpretation of keywords for semantic search. In: Aberer, K., et al. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 523–536. Springer, Heidelberg (2007)
19. Yang, M., Ding, B., Chaudhuri, S., Chakrabarti, K.: Finding patterns in a knowledge base using keywords to compose table answers. Proc. VLDB Endow. **7**(14), 1809–1820 (2014)
20. Zhou, M., Pan, Y., Wu, Y.: Conkar: constraint keyword-based association discovery. In: 20th ACM International Conference on Information and Knowledge Management, pp. 2553–2556 (2011)