Mining Simple Path Traversal Patterns in Knowledge Graph

Feng Xiong and Hongzhi Wang*

Department of Computer Science, Harbin Institute of Technology, China
E-mail: fxiong@hit.edu.cn; wangzh@hit.edu.cn
*Corresponding Author

Received 19 July 2021; Accepted 16 August 2021; Publication 03 January 2022

Abstract

The data mining has remained a subject of unfailing charm for research. The knowledge graph is rising and showing infinite life force and strong developing potential in recent years, where it is observed that acyclic knowledge graph has capacity for enhancing usability. Though the development of knowledge graphs has provided an ample scope for appearing the abilities of data mining, related researches are still insufficient. In this paper, we introduce path traversal patterns mining to knowledge graph. We design a novel simple path traversal pattern mining framework for improving the representativeness of result. A divide-and-conquer approach of combining each path is proposed to discover the most frequent traversal patterns in knowledge graph. To support the algorithm, we design a linked list structure indexed by the length of sequences with handy operations. The correctness of algorithm is proven. Experiments show that our algorithm reaches a high coverage with low output amounts compared to existing frequent sequence mining algorithms.

Keywords: Knowledge graph, path traversal, data mining.
1 Introduction

The advancement of the Mobile Internet increases the likelihood of completely realizing the Internet of Everything. The amount of data created as a result of such interconnectivity has skyrocketed, and it might be used as useful raw materials for relation analysis. Aside from the prior emphasis on each individual in terms of intellectual analysis, the relations between each will definitely become significant as a result of the Mobile Internet age. Fortunately, Knowledge Graph might be useful as long as there is a need for relation analysis.

Knowledge graph is a repository that is constructed by semantic networks in essence, and it could be seen as a multi-relation graph. Introducing path traversal patterns mining algorithms to knowledge graph promotes efficiency. For instance, in a knowledge graph of investor relations researchers use frequent supply chain partners to analyze the cause of relationship-specific asset [19]. Path traversal patterns mining algorithms could help obtain possible passes from data compression graph [23]. The intention to identify suitable documents by extracting a knowledge graph of relations from a large scientific dataset requires frequent path traversal patterns relations [1].

Additionally, loops are always not welcomed in such graph. According to the survey, it is commonly observed that the 96\% of loops in Probase [36] have wrong isA relations [18]. Removing loops from the Wikipedia graph when constructing a primary taxonomic tree is imperative [9]. A loopless knowledge graph that is told facts about what is true in the world is often a sign of a bug [26]. Therefore, developing the research of simple path (path without loops) [12, 13, 40] traversal patterns mining in knowledge graph is imperative.

However, the appropriative research of simple path traversal patterns on knowledge graph is absent. Even though we may utilize several classic algorithms to handle this issue, such as GSP [32], Spade [41], PrefixSpan [24], Spam [2], they could not satisfy well. The vulnerability of such algorithms is that they could produce frequent subsequences from a graph which is constructed by a training set, but they could not produce frequent simple path traversal patterns from it. Furthermore, we could modify path traversal patterns mining algorithms [5, 21] for mining path traversal patterns from knowledge graph, the additional cost for loop computation could decrease the efficiency.

To determine the frequency of simple path, We introduce an evaluation parameters Coverage. Coverage is accumulated by each sequence of traversal
Figure 1 An example of a naive knowledge graph.

data, which is a subsequence of other sequences. The sufficiency of a set of frequent sequences could be determined by its Coverage because one covers and represents most sequences in the training set. We consider Coverage as a critical parameter to measure the effectiveness of an algorithm for producing frequent sequences on knowledge base. To illustrate the point, we give a naive example.

We have four entities $A, B, C, D$ in an knowledge graph $G$. There are three sequences in the training set, in which one sequence $< A, B, C >$ is used to build relations between $A, B, C$, where $A$ is directly related to $B$, $B$ is directly related to $C$. We also have such two sequences $< B, C, D >$ and $< A, C >$ in the training set. Together, a naive knowledge graph $G$ is build by the training set as shown in Figure 1 where the number on each edge tells the total frequency of relations between each pair of entities. We want to discover frequent sequences in $G$. Classic frequent sequential patterns mining algorithms would give the result that there are two sequences, $< B, C >$ and $< A, C >$, who have a support of 2 sequences. However, there is a longer sequence $< A, B, C, D >$ which is a substructure of $G$ despite it has a support of only 1 sequence. This sequence represents all of three sequences $< A, B, C >$, $< B, C, D >$ and $< A, C >$, namely it has a Coverage of 3 sequences. Hence one can see that Coverage is significative and valuable to determine if a sequence of traversal data in knowledge graph is frequent enough.

In this paper, we study simple path traversal mining problem on acyclic knowledge base. With the intention of gaining high Coverage, we develop a novel framework containing both data structure and algorithm, which is different from all classic sequential pattern mining algorithms. The structure indexing the length of each sequence to support the algorithm is concise and easy to operate. The algorithm aims to produce a set of frequent sequences of traversal data with high Coverage through combining each sequence of different length, which is efficient and effective. To ensure the efficiency, we design the combining process in a divide-and-conquer manner and the
pruning strategy to accelerate it. To ensure the effectiveness, we combine sequences in the priority of sequence length and support resulting in a small set of generated simple path traversal patterns with high Coverage.

The contributions of this paper are summarized as follows.

- We study the simple path traversal patterns mining problem on knowledge base with Coverage as the optimization goal. As we know, this is the first paper that studies this problem.
- To discover high-coverage paths, we design a novel algorithm that considers both the length and frequency of the discovered results. For the consideration of efficiency, we combine sequences of traversal data in a divide-and-conquer approach. We prove that our algorithm achieves the goal of problem by reasoning.
- In this algorithm, we add pruning strategy facing to not only the frequency but also the Coverage. We also design a linked list for the sequences indexed by their lengths to support the algorithm efficiently.
- Extensive experimental results on real data sets show that the proposed approach could discover simple path traversal patterns on knowledge graph with high coverage efficiently.

The rest of the paper is organized as follows. Section 2 introduces the preliminaries, problem definition, and the outline of algorithm. Section 3 illustrates the data structure with its maintenance operations. Section 4 introduces algorithms of sequences combination. In Section 5, we design the algorithm, and its pruning strategies. We prove the correctness of algorithm by reasoning. Section 6 shows the result of our experiments. Section 7 discusses the related work of the problem. Section 8 draws the conclusions and presents an expectation of our work.

2 Overview of Algorithm

In this section, we introduce the basic definitions related to the problem. After that we determine the definition and optimization goals of our problem. Finally, the outline of our algorithm is given to make our work easier to understand.

2.1 Basic Definitions

A simple path traversal patterns is modeled as a sequence, which is an ordered list of elements denoted by \( s = \langle e_1, \ldots, e_n \rangle \), where \( e_1 \neq e_n \) because the path should be simple. The length of a sequence \( s \) is the number of elements
in $s$, denoted by $|s|$. A sequence $s_j$ is a subsequence of another sequence $s_i$ iff the set of ordered element in $s_j$ is a subset of that in $s_i$, denoted by $s_j \sqsubseteq s_i$. Also, $s_j$ is a subsequence of $s_i$. Given two sequences $s_i = <a_1, \ldots, a_n>$ and $s_j = <b_1, \ldots, b_m>$, formally we have the following definitions.

**Definition 1.** $s_j \sqsubseteq s_i$ iff there exists integers $1 \leq x_1 < \cdots < x_m \leq n$ such that $b_1 = a_{x_1}, \ldots, b_m = a_{x_m}$.

**Definition 2.** $s_j \sqsubseteq s_i$ iff $s_j \sqsubseteq s_i$ and $s_j \neq s_i$.

For example, we describe the sequences of marital status as follows.

**Example 1.** Consider five elements $e_1, e_2, e_3, e_4, e_5$, and three sequences $s_1 = <e_1, e_2, e_3>$, $s_2 = <e_1, e_2, e_4>$, $s_3 = <e_1, e_2, e_4, e_5>$, we notice that $s_2 \sqsubseteq s_3$.

We denote $D$ as a training set containing a set of sequences such that $D = \{s_1, \ldots, s_n\}$. The acyclic knowledge base is a directed acyclic graph $G = (V, E)$ constructed by $D$, where $V$ is a set of vertices that $\forall v' \in V$, $v' \in s_i$ and $E$ is a set of edges that $\forall e' = <v', v''> \in E$, $<v', v''>$ is an adjacent pair in $s_i$.

**2.2 Problem Definition**

In this paper, we focus on simple path traversal patterns mining problem. In order to achieve an unsupervised process effectively and efficiently, we have three demands of the problem as follows. Note that these requirements are not incompatible and we prove that our approaches achieve these demands later.

**Maximize coverage.** Coverage is accumulated by each sequence of traversal data, which is a subsequence of one of resulted sequences in the knowledge graph. The resulted set of frequent sequences covering and representing most sequences in the training set always has high Coverage. Thus, the sufficiency of the resulted set of frequent sequences could be determined by its Coverage. This is the main optimization goal of the problem.

**Maximize length of rules.** Because a longer sequence could cover more paths in the knowledge base and our algorithm is on the basis of combining, it is necessary to reduce the size of set of sequences for efficiency and effectiveness. Besides, retrench the size of the set of frequent sequences could make one more representative and help save both of internal and external storage space.
Ensure rules are frequent. We should ensure the support of all sequences are over a predefined threshold. This will make the generated sequences frequent enough. Then, our result could benefit from it and obtain frequent sequences from the knowledge base. This improves the usability of the resulted set of frequent sequences.

Based on these requirements, we give a formal definition of our problem. We denote $D$ as the input and $T$ the result. $D$ is a set containing a number of sequences such that $D = \{s_1, \ldots, s_n\}$. Then, we denote one resulted sequence by a pattern $t_i$. For each pattern $t_i$, its compatible set in $D$ is denoted by $S^D_{t_i}$ such that $S^D_{t_i} = \{s_i|s_i \subseteq t_i, s_i \in D\}$. $T$ is a set of such patterns that $T = \{t_1, \ldots, t_1, \ldots, t_m\}$. Finally, the coverage of $T$ is $|\bigcup_{t_i \in T} S^D_{t_i}|$. To compare the Coverage among different results without loss of generality, we define the rate of Coverage of $T$ as

$$cov = \frac{|\bigcup_{t_i \in T} S^D_{t_i}|}{|D|}$$

Problem. For the input $D$, the goal of the problem is to obtain a proper pattern set $T$ from $D$, that satisfies the following three conditions.

- Maximize $cov(T)$, and
- Maximize $\sum |t_i|$, the average length of sequences in $T$, and
- For each $t_i \in T$, $|S^D_{t_i}| \geq \epsilon_f$, where $\epsilon_f$ is the threshold of the pattern frequency.

As above, three conditions meet the demands of the problem. In this paper, we design the data structure and algorithms around this problem. And we prove how our algorithms meet these three conditions in Section 5.2.

3 Data Structure

To support effective and efficient currency rule discovery, we develop a specific data structure of the sequence set $D$. Based on above discussions, the basic elements in our data structure named $FSGroup$ are sequences with their support. $FSGroup$ has two basic operations as follows.

- (Insert.) Inserting a sequence into the structure with its support.
- (Delete.) Deleting a sequence and remove its entry from the structure.

Additionally, the data structure should support the sequential access of sequences with the same length one by one. Note that the combination of two
sequences, which is an important operation in the algorithm, is implemented with these operations, and will be discussed in Section 4. In this section, we focus on the structure (in Section 3.1) and the implementation algorithms of these basic operations (in Section 3.2).

### 3.1 Storage Structures

Since a longer sequence contains more elements than the shorter, the coverage of the former is higher than the latter. Thus, the combine operation on $D$ is on the longest and second longest sequence. To support these operations, the sequences are grouped by their lengths. We use a linked list to store the sequences with the same length for the efficiency of insert and delete operation. To access the sequences with the specific length, we group the heads of all linked lists as an array and index the heads with the lengths of sequences in the linked lists. Formally, the structure is defined as follows.

**Definition 3.** A FSGroup contains following three components.

- Groups of linked lists, denoted by $B$. Each linked list represents the set of sequences with the same length. Each entry in a linked list is a triad $(seq, count, next)$ where $seq$ is a sequence, $count$ the support of $seq$, and $next$ the pointer to the next entry.
- A pair of length information $P = (plen, mlen)$, where $plen$ predefines the maximum length of the combined sequences, and $mlen$ is the maximum length of all sequences.
- An array for head nodes of all linked lists, denoted by $H$. Head node $h_i$, $i = [2 \ldots P.plen]$ indicates the head of the linked list of all sequences with length $i$.

In summary, we describe the data structure as a triad.

$$L = (B, P, H)$$

where $L$ is the structure of $D$, $B$ its set of sequence entries, $P$ its additional properties, $H$ its index table of linked lists. The additional space consumption is the index table, of which the length is considered to be the maximum length of combined sequences. Moreover, it is convenient for using this structure because sequences with high supports are also longer than others. Therefore, we could only visit the big end of the index table $H$ when we retrieve the final result of all combinations. The structure is demonstrated in Figure 2.
3.2 Operations

In this section we introduce basic operations in $L$ to support our algorithms. Since we can search a sequence directly by its length, we only need two operations including insert and delete. The time complexity of insert operation is $O(n)$ and delete $O(1)$, such that it is easy to use the data structure we devise.

**Insert.** The insert operation is accomplished by function $FSGInsert(L,b_m)$, which inserts an entry $b_m$ into $L$ in accordance with its length. We insert $b_m$ into the proper position to ensure that the support of nodes in the linked listed pointed by $h_{|b_m|}$ are in a monotone decreasing single linked list.

The pseudocode of this algorithm is shown in Algorithm 1. Line 2 obtains the head node of the linked list indexed by the length of input sequence. Lines 3–4 find the proper position where we insert the entry to keep monotone decreasing. Lines 5–6 insert $b_m$ into the linked list.

**Delete.** We delete a sequence by using the basic delete operation of linked list structure. We invoke function $FSGDelete(L,b_m)$ to delete an entry $b_m$ in

---

**Algorithm 1 FSGInsert algorithm**

1: function $FSGInsert(L,b_m)$
2:    $p \leftarrow L.h_{|b_m|}$
3:    while $p.next.count > b_m.count$ do
4:      $p \leftarrow p.next$
5:    $b_m.next \leftarrow p.next$
6:    $p.next \leftarrow b_m$
7:    end while
8: end function

---

Figure 2  Illustrative diagram of $L$. 
4 Combination of Sequences

In this section, we propose methods to combine sequences effectively and efficiently. It is crucial for the combine operation because it generates a long sequence of two sequences that meets the requirements. We implement this in a divide-and-conquer approach. For this task, we define two operations, \textit{merge} and \textit{combine}. The former fuses a sequence into its supersequence. The latter generates the supersequence of two sequences which overlap but do not contain each other. We will discuss these two operations in Sections 4 and 4.2, respectively.

4.1 Merge Operation

With regard to \( b_m \) and \( b_n \), two entries in \( L \) with \( b_m.seq \sqsupseteq b_n.seq \), the merge operation merges \( b_m \) into \( b_n \). Concretely, this operation delete \( b_m \) and add the support of \( b_m \) to \( b_n \). Since \( b_m.seq \sqsupseteq b_n.seq \), \( b_n.seq \) does not change in this operation.

\begin{algorithm}
\begin{algorithmic}[1]
\Function{FSMerge}{$L, b_n, b_m$}
\State $b_n.count \leftarrow b_n.count + b_m.count$
\State \Call{FSGDelete}{$L, b_m$}
\State \Return $b_n$
\EndFunction
\end{algorithmic}
\end{algorithm}

The pseudocode of merge operation is shown in Algorithm 2. Line 2 accumulates the support of \( b_m \) to \( b_n \). Then in Line 3, we delete \( b_m \). The time complexity of this operation is \( O(1) \).

4.2 Combine Operation

Given \( b_m \) and \( b_n \), two entries in \( L \), the combine operation combines them into one new entry \( b_f \), where \( b_f.seq \) contains all elements in both \( b_m.seq \) and \( b_n.seq \), and satisfies \( b_m.seq \sqsubset b_f.seq \) and \( b_n.seq \sqsubset b_f.seq \). We will discuss the implementation algorithm for this operation in this section. At first, we overview the implementation in Section 4.2.1 and then we describe the steps in detail in Sections 4.2.2, 4.2.3, and 4.2.4, respectively.
4.2.1 Algorithm overview

Combine operation attempts to get the “union” of $b_m.seq$ and $b_n.seq$. However, the union of two sequences is different to that of sets due to the order of elements in the sequence. Consider a brief example. $s_1 = <h, A, t>$, and $s_2 = <h, B, t>$. The longest common sequence (LCS for short) of $s_1$ and $s_2$ is $<h, t>$. Between $h$ and $t$, $s_1$ contains the subsequence $<A>$ while $s_2$ contains the subsequence $<B>$. The results of the combination should be a sequence. It is a problem how to combine such different subsequences between common sequences.

As simple path traversal patterns mining requires frequent sequences, the combination of two different subsequences should maximize the frequency of the result. Thus, we attempt to choose the order of the elements in different sequences with the highest frequency among all possible combinations. In the example above, if the frequency of subsequence $<A,B>$ is larger than that of $<B,A>$ in the whole sequence set $D$, we should choose $<A,B>$ in the combination result. As a result, the combination of $s_1$ and $s_2$ is $<h, A, B, t>$.

To state our approach formally, we first introduce some concepts. We split two sequences $s_1$ and $s_2$ into two kinds of parts, common parts and alien parts, as introduced in Definition 4.

**Definition 4.** For two sequences $s_1, s_2$, if there exists a continues sequence $t$, $t \subseteq s_1, t \subseteq s_2$ and does not exists any subsequence $t^*, t \sqsubset t^*, t^* \subseteq s_1, t^* \subseteq s_2$, we call $t$ a common part. In $s_1$ or $s_2$, if there exists a sequence $t'$ between two common parts, then $t'$ is called an alien part.

We use an example to illustrate this concept.

**Example 2.** For two sequences $s_1$ and $s_2$ that $s_1 = <h, A, B, F, G, K, I, t>$, $s_2 = <h, C, D, E, F, G, I, J, t>$, their common and alien parts are shown in Figure 3.

Intuitively, $s_1$ and $s_2$ share same common parts and their common parts form their largest common sequence. Thus, common parts of two sequences could be generated by the largest common sequence generation algorithm [6]. Then, two sequences are split into several parts according to

$$s_1 = \langle h|A.B|F.G|K.I|t \rangle$$

$$s_2 = \langle h|C.D.E|F.G|I.J|t \rangle$$

**Figure 3** Common and alien parts in $s_1$ and $s_2$. 
the common parts. Between two consecutive common parts, each continuous part is an alien part. After this step, each sequence is split into a series of common parts and alien parts and could be represented as the form \( <h, t_1, \ldots, t_i, \ldots, t_n, t>\), \( t_i \sqsubseteq s, i \in [1 \ldots n] \), where \( t_i \) is either a common part or an alien part. \( h \) and \( t \) identify the head and tail of sequence, respectively.

With this definition, we may discuss the specifics of the combine operation’s implementation methods. This method integrates related common and alien components into a single algorithm. Since the more complex step is the combination of alien parts, we discuss AlienCombine algorithm in Section 4.2.2 at first. Then based on the AlienCombine algorithm, we propose PartCombine algorithm in Section 4.2.3. PartCombine algorithm combines both common parts and alien parts. In order words, it includes three conditions, one of which is processed by AlienCombine algorithm. At last, with PartCombine algorithm as the core step, the whole algorithm, FSCCombine algorithm is introduced in Section 4.2.4. The relationship of these algorithms is shown in Figure 4.

4.2.2 AlienCombine algorithm

In this section, we discuss the implementation of AlienCombine step. As discussed above, the combination of two alien parts selects their element order with frequency.

To make the selection, we attempt to combine the elements in two alien parts in sort-merge style. That is, we use two cursors, \( c_1 \) and \( c_2 \), pointing to the processing elements in two alien parts \( t_1 \) and \( t_2 \). If the frequency of \( <c_1, c_2>\) is larger than that of \( <c_2, c_1>\), \( c_1 \) is selected as an element of the combined sequence and the cursor in \( t_1 \) moves to the next. The process continues until the cursor points to the tail of one sequence and the remaining elements of the other are concatenated to the tail of the generating sequence.

To compare the elements according to the sequence frequency, we define the weight of element pair in Definition 5 and give a brief example in Example 3.
Definition 5. Let \( \Gamma(e_i, e_j) \) denote the total support of the subsequence \( s_{\text{sub}} =< e_i, e_j > \). That is, \( \Gamma(e_i, e_j) = | \{ s \in D \mid s_{\text{sub}} \subseteq s \land e_i, e_j \text{ are adjacent in } s \} | \).

Example 3. Let \( D = \{ s_1, s_2, s_3 \} \), where \( s_1 =< A, B >, s_2 =< A, B, C >, s_3 =< A, B, C, D > \). Thus, \( \Gamma(A, B) = 3, \Gamma(B, C) = 2, \Gamma(C, D) = 1, \Gamma(A, C) = 0 \) and \( \Gamma(B, A) = 0 \).

Algorithm 3 AlienCombine algorithm

```plaintext
function AlienCombine(t_1, t_2)
    Out ← ∅
    c_1 ← t_1[0], c_2 ← t_2[0]
    while c_1 ≠ ∅ or c_2 ≠ ∅ do
        if \( \Gamma(c_1, c_2) > \Gamma(c_2, c_1) \) then
            Out ← Out + c_1
            c_1 ← c_1.next
        else
            Out ← Out + c_2
            c_2 ← c_2.next
        end if
    end while
    while c_i ≠ ∅, i = 1, 2 do
        Out ← Out + c_i
        c_i ← c_i.next
    end while
    return Out
end function
```

Based on the weight function, the pseudocode of this algorithm is shown in Algorithm 3, where we use + to represent the concatenation of sequences. For instance, if \( s =< a, b > \) and \( t =< c, d, e > \), \( s + t =< a, b, c, d, e > \).

Line 2 initiates the output. Line 3 points the cursors to the head elements of two sequences, respectively. Lines 4–10 judge which cursor should be concatenated to the result \( Out \), then move the cursor to the next, until one is pointed to \( ∅ \), the tail. Lines 11–13 output rest of elements to \( Out \) orderly. As a result, Line 14 returns the combination of two alien parts.

Example 4. We use the example shown in Figure 5 to demonstrate the proposed algorithm. Firstly, the cursor \( c_1 \) and \( c_2 \) are pointed to head of \( t_1 =< A, B > \) and \( t_2 =< C, D, E > \), respectively. We suppose \( \Gamma(A, C) > \Gamma(C, A) \), so we choose the element \( A \) and move \( c_1 \) to the next. Then we suppose \( \Gamma(B, C) > \Gamma(C, B) \), so we select \( B \) and move \( c_1 \) to the next.
Noticing that $c_1$ points to the tail of $t_1$, we orderly output rest of elements in $t_2$. Finally, we get the result in $Out$ as shown.

Since this operation is implemented in a divide-and-conquer approach, the time complexity is reduced from $O(mn)$ to $O(m + n)$ where $m$ and $n$ denote the length of input sequences, respectively. To simplify representation of time complexity, we let $m$ denote the max length of two sequences. The time complexity is $O(m)$. Property 1 demonstrates the correctness of this strategy. $A \rightarrow B$ means $A$ and $B$ are adjacent in the sequence combined.

Property 1. For two alien parts $s = < e_1, \ldots, e_i, \ldots, e_n >$, $t = < f_1, \ldots, f_j, \ldots, f_m >$, $\forall e_i \in s$, if $\exists f_j \in t$, $e_i \rightarrow f_j$, we have $e_i \rightarrow f_{j'}$, $j' \in [(j + 1) \ldots m]$, vice versa.

Proof. We use contradiction. Suppose $f_{j+1} \rightarrow e_i$. Because of $f_j \rightarrow f_{j+1}$ and $e_i \rightarrow f_j$, these three paths form a cycle, which is contradict to our statement of breaking the cycle in Section 2.1. Hence we have $e_i \rightarrow f_j \Rightarrow e_i \rightarrow f_{j+1} \Rightarrow \cdots \Rightarrow e_i \rightarrow f_m$.

4.2.3 PartCombine algorithm

In this section, we propose the algorithm to combine both common parts and alien parts. Generally, the combination of two sequences $s_1$ and $s_2$ may involve following three conditions.

- A common part $s'$ of $s_1$ and $s_2$, $s'$ is included in the result of combination, and
- An alien part $s_i$ in one sequence without a corresponding alien part in the other sequence. That is, for two common parts $s_{i-1}$ and $s_{i+1}$, $s_1 = < h, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, t >$ and $s_2 = < h, \ldots, s_{i-1}, s_{i+1}, \ldots, t >$. In this case, the combination result is $< h, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, t >$, and
• Two alien parts \( s_i \) and \( s_j \) share the same adjacent common parts. That is, for two common parts \( s_{i-1} \) and \( s_{i+1} \), \( s_1 = \langle h, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, t \rangle \) and \( s_2 = \langle h, \ldots, s_{i-1}, s_j, s_{i+1}, \ldots, t \rangle \). For this case, we invoke Algorithm 3 to combine \( s_i \) and \( s_j \).

**Example 5.** We use an example to demonstrate these three conditions in Figure 6, where the number below each block shows the case of the part.

Considering all these three cases, we design PartCombine algorithm as depicted in Algorithm 4. In this algorithm, we also use + to represent the concatenation of sequences as in Algorithm 3.

---

**Algorithm 4** PartCombine algorithm

1. function PartCombine \((s_1, s_2)\)
2. \(LCS(s_1, s_2)\)
3. \(c_1 \leftarrow s_1.h, c_2 \leftarrow s_2.h, \ Out \leftarrow \emptyset\)
4. while \(c_1 \neq t \) and \(c_2 \neq t\) do
5. if \(c_1\) is common then
6. \(Out \leftarrow Out + c_2\)
7. \(c_2 \leftarrow c_2.next\)
8. if \(c_2\) is common then
9. \(c_1 \leftarrow c_1.next\)
10. else
11. \(c_1 \leftarrow c_1.next\)
12. if \(c_2\) is common then
13. \(Out \leftarrow Out + c_1\)
14. else
15. \(Out \leftarrow Out + AlienCombine(c_1, c_2)\)
16. \(c_2 \leftarrow c_2.next\)
17. end if
18. end if
19. end if
20. end while
21. return \(Out\)
22. end function
Initially, these two sequences are partitioned into common parts and alien parts with the longest common sequence algorithm (in Line 2). Then the parts are combined in a merge style. We use two cursors, \( c_1 \) and \( c_2 \), pointing to the processing parts of \( s_1 \) and \( s_2 \), respectively. Line 3 initiates the two cursors and the output. Lines 4–16 iteratively process parts pointed by cursors until one of which points to the tail of \( s_1 \) or \( s_2 \). We obtain the output of this function, namely, the result of combining two sequences in Line 18. In detail,

- When the part in \( c_1 \) is a common part but that in \( c_2 \) is an alien one (Line 5), we concatenate the part pointed by cursor \( c_2 \), then move it to the next (Lines 6, 7).
- When the both of parts in \( c_1 \) and \( c_2 \) are common parts (Lines 5, 8), we concatenate the any part (because they are the same as stated to the result, then move both cursors \( c_1 \) and \( c_2 \) to the next (Lines 6, 7, 8).
- When the both of parts in \( c_1 \) and \( c_2 \) are alien parts (Lines 9, 10), we invoke AlienCombine Algorithm to combine two alien parts, the output of which is concatenated to the result. Then we move both cursors \( c_1 \) and \( c_2 \) to the next (Lines 11, 12, 13).

The time complexity of this operation is still \( O(m) \) because the time complexity of all three conditions are \( O(m) \).

### 4.2.4 FSCombine algorithm

To combine two entries \( b_m \) and \( b_n \) in \( L \), besides the combination of sequences \( b_m.seq \) and \( b_n.seq \) with PartCombine Algorithm discussed in Section 4.2.3, we also need to delete entries \( b_m \) and \( b_n \), add theirs supports to the combined entry \( b_f \), and insert \( b_f \) to corresponding list.

**Algorithm 5** FSCombine algorithm

1: \( \text{function } \text{FSMerge}(L, b_n, b_m) \)  
2: \( \quad \text{b}_f.seq \leftarrow \text{PartCombine}(b_n.seq, b_m.seq) \)  
3: \( \quad \text{b}_f.count \leftarrow b_n.count + b_m.count \)  
4: \( \quad \text{FSGInsert}(L, b_f) \)  
5: \( \quad \text{FSGDelete}(L, b_m) \)  
6: \( \quad \text{FSGDelete}(L, b_n) \)  
7: \( \quad \text{return } b_f \)  
8: \( \text{end function} \)

The pseudocode is shown in Algorithm 5. Line 2 gets the result of combination from function PartCombine as mentioned before. Line 3 accumulates the support from two subsequences to the new entry. Line 4 inserts the new
entry $b_f$ to $L$. Lines 5–6 delete two old entries $b_m$ and $b_n$, respectively. Line 7 returns $b_f$.

The upper bond of time complexity of this algorithm is $\max\{O(m), O(l)\}$, where the time complexity of PartCombine is $O(m)$, with the maximum length of input sequences denoted by $m$, and that of FSGInsert is $O(l)$ with the maximum length of sequence blocks in linked lists of $L$ denoted by $l$.

5 The CFSM-AKB Algorithm

In this section, we propose the CFSM-AKB (Coverable Frequent Sequence Mining on Acyclic Knowledge Base), with the data structure and algorithms introduced above (in Section 5.1). Then we demonstrate the correctness of CFSM-AKB (in Section 5.2).

5.1 CFSM-AKB Algorithm

To achieve the goal of problem, in this section, we propose the CFSM-AKB algorithm. In this algorithm, all sequences are stored in a FSGroup $L$ introduced in Section 3.1. The algorithm iteratively combines long sequences by invoking the FSCombine algorithm introduced in Section 4.2.4.

Initially, all input sequences with same length are grouped respectively. Then $L$ is constructed according to these groups by invoking FSGInsert. In the algorithm, groups in $L$ are identified as $g_1, \ldots, g_l$, $g_i$ is the linked list for the sequences with the $i$th shortest lengths. Since the longer sequences may be generated and some groups may be withdrawn, $l$ and the subscript of some groups will change accordingly during the algorithm.

During the algorithm, we use a cursor $c$ to point to the processing entry. In the beginning, $c$ pointing to the head of the active group with the second longest length $g_{l-1}$.

In each round, the precondition is that the entry $b_i$ pointed by $c$ is in the group $g_j$. It is decided whether $b_i$ should be merged (but not combined) into entries in $g_{j+1}, \ldots, g_l$. After $b_i$ is processed, the increasing of the length of the longest sequence means that the current group with the second longest length is not $g_{l-1}$, and we denote current one by $g_t$. Then $c$ moves to the head of $g_t$. Otherwise, $c$ moves to the next entry in $g_j$. If all entries in $g_j$ are processed, $c$ moves to the head of $g_{j-1}$. The algorithm halts when the longest length of sequences does not change and the tail of $g_1$ has been processed.

The merge or combination operation is applied on the entries according to the relationship of their sequences. For an example, for $b_i$ pointed by $c$,
if \(\exists b_k, b_i, seq\) is a subsequence of \(b_k, seq\), then \(b_i\) is merged into \(b_k\). Let \(g_j+\) denote the set of all the entries in groups \(g_{j+1}, \ldots, g_l\) such that \(g_{j+} = \{b|b \in g_i, i \in [(j+1) \ldots l]\}\). If one entry \(b_i\) in \(g_j\) could be merged with multiple entries \(b_{k_1}, \ldots, b_{k_u}\) in \(g_{j+}\), we first merge \(b_i\) into any entry \(b_{k_i}\), then combine these multiple entries two by two. In other words, \(FSCombine(b_{k_1}, b_{k_{i+1}})\) is invoked iteratively with \(i\) from 1 to \(u-1\).

The pseudocode of the whole algorithm is shown in Algorithm 6. Line 2 initiates the cursor \(c\) to the head of the current second longest sequences group \(g_{l-1}\). Line 4 starts the scanning of groups in \(L\). The algorithm stops when the tail of \(g_j\) has been processed.

For each entry, the merge or combine operation is performed. If there exists any sequence as a supersequence of \(c, seq\) (in Line 5), it invokes \(FSMerge(\)\(L, b_{k_1}, c)\) to merge \(c\) into \(b_{k_1}\) (in Line 6). Moreover, if there exists multiple
entries as supersequences of \( c_{seq} \) (in Line 7), FSCombine is invoked iteratively to combine each pair of supersequences in Lines 8–11. Note that in Line 10, all combine operations satisfies an alternative discriminant \( \Delta_E \). If the length of the longest sequence increases (in Line 12), the cursor \( c \) moves to the head of \( g_t \) (in Line 13), new second longest sequences group, and we change \( L.mlen \) to the new length of the longest sequence (in Line 14). Then we restart the algorithm from Line 4. If \( c \) points to the tail of \( g_j \) (in Line 16), it will move to \( g_{j-1} \) (in Line 17), otherwise to the next group (in Line 19). Finally, in Line 20, we return entries with supports larger than the predetermined threshold \( \epsilon_f \).

**Example 6.** Figure 7 shows a brief example of the algorithm. Five sequences with their supports are shown in Figure 7(a). We point cursor \( c \) to \( < A, C > \) at first. Then we find that two sequences are supersequences of \( c \), as stated above, \( b_{k_1} = < A, B, C, D > \) and \( b_{k_2} = < A, C, D, E > \). We merge \( c \) to \( b_{k_1} \), by invoking FSMerge\( (L, b_{k_1}, c) \). Then we invoke FSCombine\( (L, b_{k_2}, b_{k_1}) \) to combine \( b_{k_1} \) and \( b_{k_2} \) into a new entry containing \( < A, B, C, D, E > \), as shown in Figure 7(b). Since the maximum length is increased from 4 to 5, in Figure 7(c), we move cursor \( c \) to the head of \( g_t = < B, C, D, E > \). Since there is one sequence \( < A, B, C, D, E > \) as a supersequence of \( < B, C, D, E > \), we only invoke FSMerge in this time. In Figure 7(d), since \( c \) points to the tail of \( g_j \) with Lines 15–16, we move \( c \) to the head of \( g_{j-1} \) to \( < C, E > \). Then we also invoke FSMerge. Finally in Figure 7(e), the algorithm ends when \( c \) points to the tail of \( g_1 \). We get the generated sequence \( < A, B, C, D, E > \) with support 6.

### 5.2 Properties of CFSM-AKB

In this section, we study the properties of our algorithm including the time complexity and the correctness.

**Time Complexity.** We consider the upper bound of our algorithms. The worst situation is that the longest length of sequences is always changed.
after the combination of two entries. Such operation will generate one new entry and delete two in each round. Let \( n \) denote the number of all input sequences. We choose a group with at most \( l \) entries in each round and the function FSCombine is invoked \( n \times l \) times. According to the analysis in Section 4.2.4, the time complexity of function FSCombine is \( \max\{O(l), O(m)\} \). Consequently, the CFSM-AKB algorithm is polynomial time and requires \( \max\{O(nl^2), O(nlm)\} \) arithmetic operations.

**Correctness.** Since the optimal solution of the problem satisfies three conditions, we will prove the correctness of CFSM-AKB algorithm by showing that its results satisfy these conditions in Lemma 3 and 4.

**Lemma 1.** In CFSM-AKB algorithm, \( \exists b \in g_{j+\delta} \) and \( \forall d \in g_{j+\epsilon} \), respectively, where \( 0 \leq \epsilon < \delta \leq l - j \), \( d \) is deleted iff \( d.seq \sqsupseteq b.seq \).

**Proof.** Since \( d.seq \sqsubseteq b.seq \), \( |d.seq| < |b.seq| \). Because we access the head table \( H \) from long end to short end by the cursor, the algorithm invokes FSMerge\((L,b,d)\) to delete \( d \). In turn, if \( d \) is deleted, FSMerge\((L,b,d)\) is surely invoked. It means that \( |d.seq| < |b.seq| \) and \( d.seq \sqsubseteq b.seq \) accordingly. This lemma holds.

**Lemma 2.** If \( d \) is deleted, \( d \) is not to be rebuilt.

**Proof.** We use induction to prove this lemma.

- When \( |d.seq| = 1 \), since \( d.seq \) contains only one element, if it is deleted, there are no subsequences to be combined, but \( d \) cannot be rebuild apparently.
- When \( \forall e_i, |e_i.seq| = l, l < n \), we suppose that if \( e_i \) is deleted, \( e_i \) is not to be rebuilt.
- When \( |d.seq| = n \), we attempt to prove that if \( d \) is deleted, \( d \) is not to be rebuilt.

First, since \( d \) is deleted, \( \exists b \) such that \( d.seq \sqsubseteq b.seq \). (Lemma 1) Then, we use contradiction. Suppose that \( d \) is rebuilt. In such case, FSCombine\((L,e_1,e_2)\) must be invoked, where \( e_1 \) and \( e_2 \) satisfy following conditions.

- \( e_1 \in g_{j+\mu_1}, \mu_0 < \mu_1 < \epsilon, \) and
- \( e_2 \in g_{j+\mu_2}, \mu_0 < \mu_2 < \epsilon, \) and
- \( \exists e_0 \) such that \( e_0 \in g_{j+\mu_0}, 0 < \mu_0 < \epsilon, \) and
- \( e_0.seq \sqsubseteq e_1.seq \) and \( e_0.seq \sqsubseteq e_2.seq \).

In other words, \( |e_1.seq| < n \) and \( |e_2.seq| < n \). Nevertheless, since \( e_1.seq \sqsubseteq d.seq, e_2.seq \sqsubseteq d.seq, \) and \( d.seq \sqsubseteq b.seq \), we conclude \( e_1.seq \sqsubseteq
b.seq, $e_2$.seq $\sqsubseteq$ b.seq. Hence, FSMerge($L, b, e_1$) and FSMerge($L, b, e_2$) to delete $e_1$ and $e_2$, $e_1$ and $e_2$ will not be rebuilt, in accordance with the assumption that $\forall e_i, |e_i$.seq$| = l, l < n$, if $e_i$ is deleted, $e_i$ are not to be rebuilt. Therefore, $d$ does not exist, either. This is contradict to the assumption that $d$ is rebuilt. Thus, for any $d$, $|d$.seq$| = n$, if it is deleted, it is not to be rebuilt.

As a result, for any $d$, if it is deleted, it will not be rebuilt. This lemma holds.

**Lemma 3.** For the result set $T$ of CFSM-AKB algorithm, both $\text{cov}(T)$ and $\frac{\sum |t_i|}{|T|}$ are maximal.

**Proof.** Firstly, we prove that two claims in this lemma are equivalent.

- Formally, let $S^D_T$ denote the union of compatible set of all generated rules that $S^D_T = \bigcup_{t_i \in T} S^D_{t_i}$. According to the function that $\text{cov}(T) = \frac{|S^D_T|}{|D|}$, if we want to maximize $\text{cov}(T)$, for the same data set $D$, we should maximize $S^D_T$. In other words all sequences in $D$ should be covered by sequences in $T$. It is impossible that a sequence $s'$ in $D$ but not in $S^D_T$ is a subsequence of any sequences in $T$.

- Similarly, according to the function that $\frac{\sum |t_i|}{|T|}$, if we want to maximize it, we should maximize $\sum |t_i|$. Because the combination of two sequences increases the average length, we should ensure that there does not exist two sequences in $T$ satisfy the condition that one is a subsequence of the other. Formally, $\forall t_u \in T$, there does not exist $t_u \subseteq t_v$.

Therefore, all we need is to prove that during the algorithm such $s'$ and $t_v$ do not exist. We set $t_i$ and $t_v$ correspond to $b$.seq, and $s'$ and $t_u$ correspond to $d$.seq. Thus, two claims are equivalent.

According to Lemma 1 and $d$.seq $\sqsubseteq$ b.seq, $d$ is deleted. According to Lemma 2, $d$ is not to be built. As a result, in $T$, there does not exists $d$ such that $d$.seq $\sqsubseteq$ b.seq. This lemma holds.

**Lemma 4.** The frequency of each sequence generated by CFSM-AKB algorithm is over a threshold. Formally, for the result set $T$ of CFSM-AKB algorithm, $\forall t_i \in T, |S^D_{t_i}| \geq \epsilon_f$.

**Proof.** In Algorithm 6 with Line 20, only sequences whose supports are over the predetermined threshold $\epsilon_f$ are output. Thus, this lemma holds.

According to these lemmas, the following theorem shows the correctness of CFSM-AKB algorithm.
Theorem 1. The CFSM-AKB algorithm is correct because of the output of that, $T$, satisfies three following claims.

- $\text{cov}(T)$ is maximal, and
- $\sum_{t_i \in T} |S_{t_i}^D|$ is maximal, and
- $\forall t_i \in T, |S_{t_i}^D| \geq \epsilon_f$.

Proof. According to Lemma 3 and 4, $\forall T$, we have $\forall t_i \in T, |S_{t_i}^D| \geq \epsilon_f$, $\text{cov}(T)$ and $\sum_{t_i \in T} |S_{t_i}^D|$ reach their maximum value, respectively. Therefore, CFSM-AKB algorithm is correct for achieving the goal of problem.

6 Experiments and Analyses

To verify the performance of the proposed approach, we conduct extensive experiments on both real and synthetic data. We perform the experiments on a PC machine with 3.1GHz Intel(R) Core(TM) CPU and 8GB main memory, running Microsoft Windows 7. All algorithms are implemented in C language in Microsoft Visual Studio 2013.

To evaluate the performance comprehensively, we test three aspects of our algorithms. During the experiments, default parameters are $\alpha = 0.2$ and $p = 0.005$.

6.1 Effectiveness

We use the BMSWebView2 dataset [31] used in KDD-CUP competition to test the effectiveness of the propose approach, since such data set contains sequences of click-stream data and such sequence could construct a graph for simple path traversal patterns. This data set contains 77,512 records. We use 65,000 records as the training set, and others are used as the test set. Note that we use $e_{\text{cov}} = \frac{\bigcup_{t_i \in T} S_{t_i}^D}{|D_{\text{test}}|}$ as measure of the effectiveness of the approach.

Comparisons. We conduct three experiments for comparisons. The first one is to compare the effectiveness of the proposed algorithm with two frequent sequence discovery algorithms, PrefixSpan [24] and RuleGen [41]. Even though some other approaches have been proposed such as SPADE [41], SPAM [2], and LAPIN [39], they share the same result with PrefixSpan. Meantime, PrefixSpan have a relative better performance on large data set. RuleGen is a classic algorithm on frequent sequences mining problems, whose purpose is similar to CFSM-AKB. To ensure the fairness of the
comparison, we tune the parameters of PrefixSpan and RuleGen, and pick two groups of parameters for each algorithm. These parameters make the algorithm achieve high coverage and could accomplish the computation in a reasonable time (within 2 hours).

The experimental results are shown in Table 1, where $m_s$ and $m_c$ are minimum support and minimum confidence, respectively. To show the benefits of our approach, we also report the number of discovered sequences shown in the Amount column. From the experimental results, we find that our algorithm achieves the highest coverage with relative few sequences. It shows that our algorithm outperforms existing algorithms in effectiveness significantly.

**The Effect of Parameters.** In this section we test the impact of the parameters $p$ and $\alpha$ on the effectiveness of the proposed algorithm. The experimental results are shown in Figure 8. From the results, it is observed that when $p$ goes higher, which implies more precise prediction, the coverage also increases. Another observation is that if $\alpha$ goes higher, the coverage increases due to the decline of error tolerance. These observations show that our design goal is achieved with these two parameters.

**The Effect of Training Set.** In this part, we generate synthetic datasets according to the approach introduced in [3]. Intuitively, the properties of the training dataset can also influence the results. Thus, we test the effect of the

---

**Table 1** Compared with PrefixSpan and RuleGen

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
<th>Coverage</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFSM-AKB</td>
<td>$\alpha = 0.2$, $p = 0.005$, $m_s = 10$</td>
<td>0.8320</td>
<td>1008</td>
</tr>
<tr>
<td>PrefixSpan</td>
<td>$m_s = 0.001$</td>
<td>0.0713</td>
<td>30586</td>
</tr>
<tr>
<td>PrefixSpan</td>
<td>$m_s = 0.0001$</td>
<td>0.2622</td>
<td>8597444</td>
</tr>
<tr>
<td>RuleGen</td>
<td>$m_s = 100$, $m_c = 0.6$</td>
<td>0.0231</td>
<td>98744</td>
</tr>
<tr>
<td>RuleGen</td>
<td>$m_s = 50$, $m_c = 0.6$</td>
<td>0.0424</td>
<td>3343536</td>
</tr>
</tbody>
</table>

![Figure 8](image.png)  
(a) $p$ VS. Coverage  
(b) $\alpha$ VS. Coverage

---


properties of training sets on the effectiveness. We consider three parameters of the training sets, total entries (E), max rule sequence length (L) and number of elements (I). We generate six datasets to examine the effect of different datasets. For each data set, 80 percent of data are regarded as the training data, other 20 percent as the test data.

The experimental results are shown in Figure 9(a), where E100L20I100 means there are total 100k entries, max rule sequence length is 20, containing 100 elements, and so on. The y-axis is the coverage.

From the experimental results, we have three observations. First, with fixed L and I, a larger E leads to a better coverage. It is because that more rules could be discovered from a larger training sets. Second, with a fixed E, when L and I increase, the coverage decreases. The reason is that more elements lead to more complex rules and make the rules difficult to discover. Third, even though the coverage decreases with the increasing L and I, we could still increase the coverage by enlarging the size of training data set.

### 6.2 Efficiency

We test the efficiency of CFSM-AKB Algorithm on synthetic data set generated in the same way as in Section 6.1.3. We vary E from 5k to 100k to test the efficiency on training set with various data sizes. In the experiments, we set L = 20 and I = 100. The experimental results are shown in Figure 9(b), where the x-axis is the size of total entries of a dataset and y-axis is the rule discovery time on the data set. From the experimental results, the running time is around linear with the number of entries, which agrees with the time complexity in Section 5.2.
6.3 Effectiveness of Pruning Strategy

Since in FSCombine algorithm, the combination of two long sequences results in error in high probability, we test the probability that two sequences with different lengths can be combined according to our pruning strategy. We test the results on different $p$ and $\alpha$. Dataset and the parameters are the same as those in Section 6.1.

Figure 10 shows the restraint effect of combination of two sequences, where x-axis indicates the length of long sequence, and y-axis that of short one. Each line in each figure shows the maximum length of short sequence that can be combined with long sequences with various lengths. The experimental results are shown in Figures 10(a) and 10(b) with different $p$ and $\alpha$, respectively. From the experimental results, the probability that two sequences with different lengths can be combined increases with the decline of $p$ but with the increasing of $\alpha$. From this observation, we conclude that $p$ and $\alpha$ could control the lengths of sequences involved in the combination successfully. It demonstrates the effectiveness of the proposed pruning strategy.

6.4 Summary

The following conclusions are drawn from the experimental findings. From an acyclic knowledge base, the suggested method may extract valuable frequent sequences. The suggested method considerably surpasses current alternatives in terms of effectiveness. The suggested method is fast and can handle a high number of training sets. The pruning method provided is
successful. The length of the sequences involved in the combination might be adjusted using the two parameters.

7 Related Work

The original work of sequential pattern mining is to acquire all sequential patterns with a predefined support from a database of sequences [32], where classic works are GSP [32], SPADE [41], PrefixSpan [24], Spam [2], Lapin-Spam [39]. All of them input the same training set and parameters and return the same set of frequent sequential patterns. There are also tons of sequential patterns mining algorithms designed under diverse background recently, such as Skopus [25], uWSequence [28], WoMine [4], UNB [35], etc. The problem definition of our work is different from these works because we aim at mining simple path traversal patterns from a knowledge graph. As a result, they are ineffective to directly apply them to our problem.

There is also a body of work on sub-graph mining. They are concerned with acquiring frequent sub-graphs in a graph, such as gSpan [38], Spin [14], Leap [37], ATW-gSpan [16], AW-gSpan [16], UBW-gSpan [16], DistGraph [34], MuGraM [15]. Additionally, there are data mining for path traversal patterns under different circumstances, especially on World Wide Web [5, 21]. However, their goals are different from ours. They lack the consideration of Coverage.

Another related literature is on the topic of knowledge graph mining. One of popular researches is utilizing data mining for developing and maintaining a knowledge graph [11, 17, 30], which are not of our concern. Fortunately, several researches are involving data mining on the knowledge graph in recent years, such as rule mining [8, 10, 27], mining outliers [22], mining cardinality [20], mining conditional keys [33], mining substructures [7], mining user behaviour information [29]. Diverse data mining tasks are springing up on account of emerging researches of knowledge graph. To our best knowledge, our work is the first study of mining simple path traversal patterns on it.

8 Conclusions

In this paper, we have solved the problem of simple path traversal patterns mining in knowledge base. Our model is a special frequent sequence mining problem with consideration of coverage. To tackle this problem, we propose an algorithm based on sequence combination. To increase the efficiency and effectiveness, we develop a suitable data structure and algorithms for
sequence combination as well as pruning strategies. We prove algorithm proposed is correct. Extensive experiments demonstrate that the proposed approach could find useful frequent sequences efficiently and effectively. The further works include mining different paths in knowledge graph, mining frequent patterns in knowledge graph, and other data mining problems in knowledge graph. We believe that studying data mining in the context of knowledge graphs will lead to exciting new research.

Acknowledgements

This paper was supported by NSFC grant U1866602. CCF-Huawei Database System Innovation Research Plan CCF-HuaweiDBIR2020007B.

References


Biographies

**Feng Xiong** received the BEng degree in computer science and technology at Harbin Institute of Technology, China, in 2015. He is now working toward the PhD degree in computer science at Harbin Institute of Technology, China. His research interests include knowledge base, data quality management, data mining, and big data.

**Hongzhi Wang** is a Professor and doctoral supervisor at Harbin Institute of Technology, ACM member. His research area is data management, including data quality and graph management. He is a recipient of the outstanding dissertation award of CCF and Microsoft Fellow.