Attribute-Based Subsequence Matching and Mining

Yu PENG *, Raymond Chi-Wing Wong *, Liangliang Ye *, Philip S. Yu #

*The Hong Kong University of Science and Technology
{gracepy,raywong,llye}@cse.ust.hk

#University of Illinois at Chicago
psyu@cs.uic.edu

***Abstract*—Sequence analysis is very important in our daily life. Typically, each sequence is associated with an ordered list of elements. For example, in a movie rental application, a customer's movie rental record containing an ordered list of movies is a sequence example. Most studies about sequence analysis focus on subsequence matching which finds all sequences stored in the database such that a given query sequence is a subsequence of each of these sequences. In many applications, elements are associated with properties or attributes. For example, each movie is associated with some attributes like "Director" and "Actors". Unfortunately, to the best of our knowledge, all existing studies about sequence analysis do not consider the attributes of elements.**

In this paper, we propose two problems. The first problem is: given a query sequence and a set of sequences, considering the attributes of elements, we want to find all sequences which are matched by this query sequence. This problem is called attribute-based subsequence matching (ASM). All existing applications for the traditional subsequence matching problem can also be applied to our new problem provided that we are given the attributes of elements. We propose an efficient algorithm for problem ASM. The key idea to the efficiency of this algorithm is to compress each whole sequence with potentially many associated attributes into just a triplet of numbers. By dealing with these very compressed representations, we greatly speed up the attribute-based subsequence matching. The second problem is to find all frequent attribute-based subsequence. We also adapt an existing efficient algorithm for this second problem to show we can use the algorithm developed for the first problem. Empirical studies show that our algorithms are scalable in large datasets. In particular, our algorithms run at least an order of magnitude faster than a straightforward method in most cases. This work can stimulate a number of existing data mining problems which are fundamentally based on subsequence matching such as sequence classification, frequent sequence mining, motif detection and sequence matching in bioinformatics.

I. Introduction

Sequences are one of the important data types in our daily life [1]. A *sequence* is an ordered list of *elements* where each element is drawn from a given *element domain set*. For example, in the movie rental application, each movie in the rental store corresponds to an element and a set of all movies corresponds to the element domain set. Each customer rents a list of movies. This (ordered) list corresponds to a sequence. Table I(a) shows a movie rental record table. Each element in the element domain set is associated with a set of *properties* or *attributes*. For instance, in the movie rental application, each movie is associated with some attributes like "Release Year", "Director" and "Actors". The properties of some movies are shown in Table I(b). This table is called a *property table*. In particular, the director of movie "Titanic" is "James Cameron" and one of the actors is "Leonardo DiCaprio".

Sequence analysis has received a lot of interest from not only database and data mining communities but also bioinformatics communities. Database researchers study subsequence matching [2-6] and similarity search [7] while data mining researchers study frequent subsequence mining [8] and sequence prediction [9]. On the other hand, bioinformatics researchers study DNA sequence alignment [10], motif discovery [11, 12] and sequence classification [13, 14].

All of the above sequence analysis applications depend on a fundamental operator called *subsequence matching*. Given two sequences s and s', sequence s is said to be a *subsequence* of s' if for any two elements e and e' in s where e occurs before e' in s, both elements e and e' occurs in s' and e occurs before e' in s'. We also say that s *matches* s' (or s' is matched by s). For example, if s is "Titanic, Inception" and s' is "Titanic, The Aviator, Inception", then s is a subsequence of s' and thus s matches s'. *Subsequence matching* is formulated as follows: given a set of sequences and a query sequence q, we want to find all sequences such that q is a *subsequence* of each of these sequences. In our running example, if q is "Titanic, Inception", Alice's sequence in Table I is one of the answers for subsequence matching.

Unfortunately, the traditional subsequence matching problem fails to answer a lot of interesting questions related to the properties of elements. Consider that we want to study how movie "Titanic" creates a star "Leonardo DiCaprio". In order to do this, we want to know how many customers are interested in watching movies acted by "Leonardo DiCaprio" after they watch movie "Titanic". Note that "Leonardo DiCaprio" is not a movie (or more formally not an element in the element domain set) but is a property value of a movie. The traditional subsequence matching problem cannot achieve this goal because the original problem is based on the elements but not the property values. One may adapt the traditional problem and generate all *possible* queries in form of "Titanic, x" where x is a movie acted by "Leonardo DiCaprio". If there are M movies acted by "Leonardo DiCaprio", then this adapted approach will issue M queries, which is quite inefficient.

Motivated by this, in this paper, we are studying a new problem called *attribute-based subsequence matching* (ASM). Informally speaking, the problem is described as follows: given a query sequence which contains some elements and some property values, we want to find all sequences which are *matched* by this sequence query. We will give a formal definition in Section III.

There are a lot of convincing applications for problem ASM. One application is finding a researcher with a certain back-

TABLE I
AN EXAMPLE SHOWING THE MOVIE RENTAL APPLICATION

Customer	List of movies
Alice	Titanic, The Aviator, Inception
Bob	Titanic, The Aviator
Clement	The Departed, The Dark Knight
...	...

(a) Movie rental record table

Movie Name	Release Year	Director	Actor 1	Actor 2	...
Titanic	1997	James Cameron	Leonardo DiCaprio	Kate Winslet	...
The Aviator	2004	Martin Scorsese	Leonardo DiCaprio	Cate Blanchett	...
The Departed	2006	Martin Scorsese	Leonardo DiCaprio	Matt Damon	...
The Dark Knight	2008	Christopher Nolan	Christian Bale	Heath Ledger	...
Avatar	2009	James Cameron	Sam Worthington	Zoe Saldana	...
Inception	2010	Christopher Nolan	Leonardo DiCaprio	Joseph Gordon-Levitt	...
...	...	...	...	...	...

(b) Movie property table

ground. Suppose that we are given a number of researchers' background where each sequence of a researcher's background is a series of affiliations. Note that affiliations correspond to elements in our context. Each affiliation can be associated with some properties like "Place" and "Private/Public". UCLA is an example of affiliations and it is a *public* university located in *LA*. Microsoft Research is another example and it is a *private* company located in *Redmond*. One interesting query is to "find all researchers who graduate in *LA* and find jobs in *Microsoft Research*". Note that LA is a property value and Microsoft Research is an affiliation.

Another application can be finding a purchase pattern in point-of-sale transactions. Each transaction is a sequence of items (e.g., laptop and mouse). Each item can be associated with some categories. For example, laptop is under category "Electronics" and mouse is under category "Computer Accessories". One interesting query is to find all customers who purchased "laptop" and then purchased some "Computer Accessories". Note that "laptop" is an item and "Computer Accessories" is a category value.

The third application can be finding a biological pattern in protein sequences. In protein sequences, 20 amino-acids form an element set. Similarly, each amino acid is associated with some properties like "side chain polarity", "side chain charge" and "hydropathy index". It is interesting to study the biological patterns related to these properties in the bioinformatics literature. One may want to find all protein sequences containing "glutamic acid", a polar amino acid, and then another polar amino acid to study some chemical reactions for generating medicines and drugs.

It is worth mentioning that problem ASM can be used in a lot of problems about sequence mining and problems about bioinformatics. Sequence classification is one example. We should classify sequences according to not only the elements in the sequence but also their property values. Motif detection over sequences is another example. Based on the property values, we can find some motifs that we have not discovered before. In conclusion, if we can solve problem ASM, all existing sequence mining problems can also be extended for discovering meaningful patterns.

In this paper, we will show that a straightforward implementation by simply re-using some existing algorithms runs *inefficiently*. Motivated by this, we propose an efficient algorithm for the problem based on *Chinese Remainder Theorem*. The key idea to the efficiency of this algorithm is using some mathematical techniques in Chinese Reminder Theorem to generate concise synopsis of all sequences. Specifically, each whole sequence with associated attribute values is compressed into a triplet of *numbers* using the theorem. All of the triplets form the concise synopsis. By using the synopsis, we can find all sequences which are matched by the query sequence efficiently. In other words, this compressed representation greatly speeds up the attribute-based subsequence matching.

Note that each number in the triplet for a sequence is a *large* number so that it can store the information about a sequence. One may think that operations over large numbers are costly. However, one advantage of using large numbers is that we can make use of efficient bitwise operations over only three large numbers. Compared with the approaches not using the large numbers which involves a lot of operations on non-large numbers, manipulating the large numbers is more efficient because it involves fewer operations on large numbers and these bitwise operations are implemented efficiently.

In order to illustrate how ASM can be used in other important data mining problems, we propose a problem which finds all frequent subsequences based on ASM. Besides, we adapt an existing efficient algorithm called *SPAM* [8].

Our contributions are summarized as follows. Firstly, to the best of our knowledge, we are the first to propose problem ASM which considers the property table. This problem has a lot of convincing applications. This work can stimulate a number of existing data mining problems which are fundamentally based on subsequence matching such as sequence classification, motif detection and sequence matching. Secondly, we propose a novel algorithm for problem ASM based on Chinese Remainder Theorem. Thirdly, we propose a data mining problem to find all frequent subsequences based on ASM to illustrate how ASM can be used in other data mining problems. Fourthly, we conducted some experiments to show the efficiency of our proposed algorithms.

The rest of the paper is organized as follows. Section II summarizes the related work in the literature. Section III gives the problem definition. Section IV introduces some fundamental concepts which are used in our proposed algorithm. Section V gives our proposed algorithm. Section VI formulates the problem of finding frequent subsequence based on ASM and gives our adapted algorithm for this problem. Section VII gives comprehensive experimental results. Section VIII concludes this paper.

II. RELATED WORK

Subsequence matching attracted a lot of attention in the database community, the data mining community and the

bioinformatic community [2-4, 15, 16, 10]. Subsequence matching can be classified into two types: *accurate matching* and *approximate matching*. Given a query sequence q, accurate (subsequence) matching [15, 16, 10] is to find all sequences such that q is a subsequence of each of these sequences. On the other hand, approximate (subsequence) matching [2-4] is to find all sequences such that each of these sequences contains a subsequence s', and the *distance* between s' and a query sequence q is at most a given *tolerance threshold*. In this paper, we focus on accurate matching.

Due to its usefulness in biological sequences, accurate matching have been studied extensively in the literature of bioinformatics. Algorithm *SW* [15] is known as the first algorithm solving subsequence matching problem in biological sequences. Besides, [16] and [10] proposed more efficient algorithms for accurate matching.

There are a lot of studies about approximate matching [2-4] requiring users to define a distance metric. [2] used the Euclidean distance as a metric and presented an algorithm for approximate matching by using some indexing techniques. [3] adopted the Dynamic Time Warping (DTW) distance as a metric and proposed an algorithm. [4] proposed to find k sequences with their smallest distances from a given sequence q where the distance metric adopted is the DTW distance.

To the best of our knowledge, no existing studies about sequence matching consider the property table which is studied in this paper.

Since subsequence matching is very useful, it has been commonly adopted in many data mining problems such as frequent subsequence mining [17, 8, 18, 6, 5], sequence classification [13, 9, 14], sequence clustering [16, 15], motif detecting [11, 12] and subsequence matching [2, 3, 15, 10].

III. Problem Definition

We are given a set $\mathcal{E}$ of *elements*. $\mathcal{E}$ is called an *element domain set*. Each element e is associated with a set $\mathcal{A}$ of m *properties* or *attributes*, namely $A_1, A_2, ..., A_m$. The value of attribute A_i of an element e is denoted by $e.A_i$ where $i = 1, 2, ..., m$. In our running example, a set of movies corresponds to the element domain set $\mathcal{E}$. Attribute "Year of Release" and attribute "Director" are two examples of the properties of a movie. For the sake of discussion, we assume that one of the attributes in $\mathcal{A}$ can uniquely identify an element e. This attribute is called an *identifying attribute*. In our example, attribute "Movie Name" is an identifying attribute. Attribute "Director" and attribute "Release Year" are non-identifying attributes.

We define the *domain* of attribute A_i, denoted by D_i, to be the set of all possible values in attribute A_i where $i = 1, 2, ..., m$. For instance, all directors like "James Cameron" and "Martin Scorsese" form the domain of attribute "Director". We define the *value domain set*, denoted by $\mathcal{V}$, to be the union of the domains of all attributes. That is, $\mathcal{V} = \cup_{i=1}^{m} D_i$. Note that $\mathcal{V}$ is a set of all possible *values*. The table (like Table I(b)) containing all attribute values of each element is called the *property table*.

A value v is said to be an *identifying value* if v is a value of an identifying attribute. Note that this value v can be used to uniquely identify an element e. For the sake of clarity, we simply say that v identifies e (or e is identified by v). For example, both "Titanic" and "The Aviator" are identifying values but "James Cameron" and "Leonardo DiCaprio" are not. We define $\mathcal{U}$ to be a set of all identifying values. Note that $\mathcal{U} \subseteq \mathcal{V}$. If value v is an identifying value, we define the *attribute value set* of v, denoted by $\alpha(v)$, to be the set of all possible attribute values of the element identified by v. For example, if v is "Titanic" (an identifying value), then $\alpha(v) = \{$"Titanic", 1997, "James Cameron", "Leonardio DiCaprio", "Kate Winslet", ...$\}$.

A *sequence* is an ordered list of values where each value is drawn from $\mathcal{V}$. Suppose that there are k values in the sequence, a sequence is represented in form of "$v_1, v_2, ..., v_k$" where $v_i \in \mathcal{V}$ for $i = 1, 2, ..., k$. In this representation, for any two values v_i and v_j where $i < j$, v_i appears before v_j. "Titanic, The Aviator, Inception" and "Titanic, Leonardo DiCaprio" are two examples of sequences. Consider that s is in form of "$u_1, u_2, ..., u_l$". Value u_i in s is defined to have the *temporal position* equal to i for $i \in [1, l]$. For example, if s = "Titanic, Leonardo DiCaprio", then "Titanic" has the temporal position equal to 1 and "Leonardo DiCaprio" has the temporal position equal to 2.

A sequence is said to be an *identifying sequence* if all values in the sequence are identifying values (i.e., all values in the sequence are drawn from $\mathcal{U}$). For example, "Titanic, The Aviator, Inception" is an identifying sequence but "Titanic, Leonardo DiCaprio" is not.

In a lot of applications, we are given a set S of *identifying* sequences. Let n be the total number of identifying sequences in S. In our running example, we are given a set of identifying sequences and each identifying sequence corresponds to the movie rental record of a customer. Table I(a) shows the set of identifying sequences. In the application of finding a researcher, an identifying sequence corresponds to the academic research background of a researcher while in the biology application, it corresponds to a protein sequence.

Let q be a query in form of "$v_1, v_2, ..., v_k$" where $v_i \in \mathcal{V}$ for $i \in [1, k]$. Given a value $v \in \mathcal{V}$ and a value $u \in \mathcal{U}$, v is said to *match* u if $v \in \alpha(u)$. For example, if u is "Titanic" and v is "Leonardo DiCaprio", then "Leonardo DiCaprio" matches "Titanic" since "Leonardo DiCaprio" $\in \alpha$("Titanic").

Definition 1 (Match): Consider a query q in form of "$v_1, v_2, ..., v_k$" where $v_i \in \mathcal{V}$ for $i \in [1, k]$. Consider an (identifying) sequence $s \in S$ in form of "$u_1, u_2, ..., u_l$" where $u_i \in \mathcal{U}$ for $i \in [1, l]$. Query q is said to *match* s (or s is matched by q) if there exist k integers, namely $j_1, j_2, ..., j_k$, such that (1) for each $i \in [1, k]$, v_i matches u_{j_i}, and (2) $1 \leq j_1 < j_2 < ... < j_k \leq l$ □

Consider that q is "Titanic, Leonardo DiCaprio" and s is "Titanic, The Aviator, Inception". Since "The Aviator" was acted by "Leonardo DiCaprio" and "Titanic" occurs before "The Aviator" in s, it is easy to verify that q matches s. In this example, $k = 2$ and $l = 3$. We also have $j_1 = 1$ and

Algorithm 1 Straightforward Algorithm for problem ASM

Input: a query q and a set S of identifying sequences
Output: a set of identifying sequences in S which are matched by q

```
1: O ← ∅
2: for each s ∈ S do
3:    isMatch ← checkMatch(q, s)
4:    if isMatch = true then
5:       O ← O ∪ {s}
6: return O
```

Algorithm 2 Algorithm **checkMatch**(q, s) (Naive Implementation)

Input: a query q and an identifying sequence s
Output: whether q matches s

```
1: let s be a sequence in form of "u_1, u_2, ..., u_l"
2: let q be a query in form of "v_1, v_2, ..., v_k"
3: j ← 1
4: for i = 1 to k do
5:    find the smallest integer r ∈ [j, l] such that v_i ∈ α(u_r)
6:    if there exists such a value r then
7:       j ← r + 1
8:    else
9:       return false
10: return true
```

$j_2 = 2$. Note that the first requirement in the above definition holds (i.e., "Titanic" matches itself and "Leonardo DiCaprio" matches "The Aviator"), and the second requirement also holds (i.e., $1 \leq j_1 < j_2 \leq l$).

In this paper, we are studying the following problem called *Attribute-based Subsequence Matching (ASM)*: Given a query sequence q, we want to find all sequences in S which are matched by q.

In our running example, if q is "Titanic, Leonardo DiCaprio", we want to find all sequences in S which are matched by q. In Table I(a), Alice's sequence is one of the sequences which are matched by q.

Note that problem ASM is more general than the traditional problem without considering the property table. This is because problem ASM becomes the traditional problem if there is only one attribute in the property table and this attribute is an identifying attribute.

A possible approach for problem ASM is shown in Algorithm 1. In this algorithm, method **checkMatch**(q, s) is to return a boolean value indicating whether a query q matches an identifying sequence s. The efficiency of this algorithm depends on how to implement method **checkMatch**. One naive implementation is shown in Algorithm 2 which takes $O(lm)$ time where l is the maximum length of a sequence in S and m is the total number of attributes. However, the time complexity of Algorithm 1 is $O(nlm)$ where n is the total number of sequences in S. In Section V, we will present an efficient algorithm based on Chinese Remainder Theorem.

IV. Preliminaries

The *remainder* for a division of a positive integer N by a positive integer d is denoted by "$N \bmod d$". Notation "$\bmod$" is called a *modular* operator. Suppose that N is a large number and can be represented by $\mathcal{N}$ 4-byte integers, and d is a small number and can be represented by a 4-byte integer. The time complexity of the modular operation is $O(\mathcal{N})$ [19].

Given a positive integer N and a positive integer d, d is said to be a *divisor* of N if $(N \bmod d) = 0$. Given three positive integers N, M and d, N and M are said to have a *common divisor* d if d is a divisor of both N and M. Two integers N and M are said to be *relatively prime* if their greatest common divisor is equal to 1.

The theorems to be introduced are based on the equation using this modular operator in form of $(N \bmod d) = r$ where N, d and r are three positive integers. This equation is called a *congruence equation*.

We introduce Unique Factorization Theorem. Unique Factorization Theorem is a fundamental theorem in number theory and is widely used in cryptography and security field.

Theorem 1: (Unique Factorization Theorem [19]) Any positive integer $N \geq 1$ can be uniquely expressed as a product of one or more prime numbers called *factors* of N. □

Property 1 (Factorization Property [19]): Given a positive integer N and a positive integer f, f is a factor of N if and only if $(N \bmod f)$ is equal to 0. □

Next, we introduce *Chinese Remainder Theorem*.

Theorem 2: (Chinese Remainder Theorem [19]) Let m be the number of congruence equations. Let $r_1, r_2, ..., r_m$ be m positive integers. Suppose that there are m pairwise relatively prime numbers: $n_1, n_2, \cdots, n_m$. Let $N = n_1 n_2 \cdots n_m$. There exists a unique integer $x \in [0, N-1]$ solving the system of m congruence equations each of which is in form of $(x \bmod n_i) = r_i$ for $i \in [1, m]$. □

In the literature, we can compute x by *Extended Euclidean algorithm* [19] in $O(m(\log n_{max})^2)$ time where $n_{max} = \max_{i \in [1,m]} n_i$. Details can be found in [19].

V. Modular Algorithm

We want to design an algorithm called *Modular Algorithm* with two major requirements which are simple to understand and are used to ease the understanding on how we use Chinese Remainder Theorem for problem ASM. They are *Requirement Value Matching* and *Requirement Sequential Order*.

Definition 2: Given a value $v \in \mathcal{V}$ and a sequence $s \in S$, $p(v, s)$ is defined to be a set of all temporal positions such that each of the values at these positions in s is matched by v. □

Consider Alice's sequence s. If v is equal to "Leonardo DiCaprio", then $p(v, s) = \{1, 2, 3\}$. If v is equal to "Martin Scorsese", then $p(v, s) = \{2\}$.

Consider query q in form of "$v_1, v_2, ..., v_k$".

Definition 3 (Requirement Value Matching): Let s be a sequence and q be a query in form of "$v_1, v_2, ..., v_k$". If for each

TABLE II
LABELS

Value	Label	Value	Label
Titanic	2	The Aviator	13
1997	3	2004	17
James Cameron	5	Martin Scorsese	19
Leonardo DiCaprio	7	Cate Blanchett	23
Kate Winslet	11	...	...
...	...	...	...

$i \in [1,k]$, $p(v_i, s) \neq \emptyset$, then s is said to satisfy *Requirement Value Matching*. □

Intuitively, Requirement Value Matching requires that each value in q matches some of the values in a sequence s (without considering the temporal ordering of values). If this requirement is satisfied, we proceed to check the second requirement, Requirement Sequential Order, considering the temporal ordering of values.

Definition 4 (Requirement Sequential Order): Let s be a sequence. If there exist k integers, namely $j_1, j_2, ..., j_k$, such that $j_i \in p(v_i, s)$ for $i \in [1,k]$ and $j_1 < j_2 < ... < j_k$, then s is said to satisfy *Requirement Sequential Order*. □

Intuitively, Requirement Sequential Order requires that these "matched" values in s have the same temporal ordering as the correspondence values in q.

This algorithm involves two major phases. The first phase is called *Phase Preprocessing* and the second phase is called *Phase Query*. In Phase Preprocessing, given a set S of identifying sequences and the property table, we generate not only some synopsis of sequences but also some data structures which will be used in Phase Query. In Phase Query, given a query q, we find all sequences in S which are matched by q using the information generated in Phase Preprocessing.

Consider a *particular* sequence s in S and an *arbitrary* query q. We want to create a synopsis for s such that we can determine whether q matches s efficiently using the synopsis. In order to achieve this goal, we should create this synopsis which can help to determine whether the two requirements are satisfied *efficiently*. In particular, the synopsis contains two separate components. The first component, denoted by V_s, is used for Requirement Value Matching and corresponds to the first number in this triplet (Section V-A). The other, denoted by X_s, is used for Requirement Sequential Order and corresponds to the last two numbers in this triplet (Section V-B).

Before we create a synopsis, we first assign each *value* in $\mathcal{V}$ with a unique prime *number* called the *label* of v. The label of v is denoted by $P(v)$. Table II shows the labels of some values in $\mathcal{V}$ in our running example.

A. Requirement Value Matching

Given a sequence $s \in S$, V_s is the *value matching number* of s which is defined as follows.

Definition 5 (Value Matching Number): Given a sequence $s \in S$ where s is in form of "$v_1, v_2, ..., v_l$", the *value matching number* of s is defined to be the product of the labels of all the values in s and their property values. Formally, $V_s = \prod_{i=1}^{l} \prod_{v \in \alpha(v_i)} P(v)$. □

Algorithm 3 Algorithm **valueMatchCheck**(q, V_s) for Requirement Value Matching

Input: a query q, V_s
Output: whether the temporal order of the matching values in s is consistent with the temporal order of all values in q

```
1: let query q be in the form of "v_1, v_2, ..., v_k"
2: for i = 1 to k do
3:     compute (V_s mod P(v_i)) and store the answer as a_i
4: if a_i = 0 for each i ∈ [1, k] then
5:     return true
6: else
7:     return false
```

Phase Preprocessing: In Phase Preprocessing, for each sequence $s \in S$, we compute V_s.

It is easy to verify that the running time of this phase is $O(nlm)$ where l is the greatest length of a sequence.

Phase Query: Recall that Requirement Value Matching is that given a query q and a sequence s, each value in q matches one of the values in s. By using V_s, we can perform k modular operations to check whether s satisfies this requirement or not where k is the length of the query sequence. Algorithm 3 shows the algorithm.

With the following lemma, we know that the algorithm is correct.

Lemma 1: Let s be a sequence and q be a query in form of "$v_1, v_2, ..., v_k$". If $(V_s \bmod P(v_i)) = 0$ for each $i \in [1,k]$, then s satisfies Requirement Value Matching. □

We know that we can use V_s to check for Requirement Value Matching. Now, we want to introduce a stronger requirement called *Requirement Duplicate Value Matching*. This requirement states that given a sequence s and a query sequence q, for each value v in q, if v appears γ times in q and there exists γ values in s such that v matches each of these γ values, then s is said to satisfy *Requirement Duplicate Value Matching*. For example, consider Alice's sequence s. If q = "Titanic, Titanic", then s does not satisfy Requirement Duplicate Value Matching. If q = "Leonardo DiCaprio, Leonardo DiCaprio", then s satisfies this requirement.

By using V_s, we can also check for Requirement Duplicate Value Matching efficiently. By doing this, we modify Algorithm 3 to Algorithm 4.

Now, we analyze the storage size of V_s. Consider a particular sequence s. According to Definition 5, we need to multiply ml prime numbers. In all of our experiments, each prime number can be represented by a 4-byte integer. The storage size of V_s is *at most* $4ml$ bytes. In case that a prime number needs more bits for storage, we can use some libraries [20] over large numbers which contain a lot of efficient bitwise operations.

Let the storage size of V_s be $\mathcal{N}$. In the above analysis, $4ml$ is a loose upper bound on $\mathcal{N}$. In other words, in most cases, $\mathcal{N} << 4ml$. In method **valueMatchCheck**, it invovles k modular operations where the divisor of each operation is a prime number. Note that in all our experiments, each prime number can be represented by a 4-byte integer. It is easy to

Algorithm 4 Algorithm **valueMatchCheck**(q, V_s) for Requirement (Duplicate) Value Matching

Input: a query q, V_s
Output: whether the temporal order of the matching values in s is consistent with the temporal order of all values in q

```
1: let query q be in the form of "v_1, v_2, ..., v_k"
2: V ← V_s
3: for i = 1 to k do
4:    compute (V mod P(v_i)) and store the answer as a_i
5:    if a_i = 0 then
6:       V ← V/P(v_i)
7: if a_i = 0 for each i ∈ [1, k] then
8:    return true
9: else
10:   return false
```

verify that the running time of **valueMatchCheck** is $O(k\mathcal{N})$.

B. Requirement Sequential Order

In Section V-B1, we first describe a *bulky* version of component X_s. Then, in Section V-B2, we describe a *compressed* version of component X_s by just a pair of two numbers based on Chinese Remainder Theorem.

1) Bulky Version of X_s: **Phase Preprocessing:** Before we describe this component, we give some definitions first.

An *interval* is defined to be in form of (l, u) where l and u are two positive integers and $l \leq u$.

Definition 6 (Appear After/Before): Let $\triangle_i$ and $\triangle_j$ be two intervals (l_i, u_i) and (l_j, u_j), respectively. $\triangle_i$ *appears before* $\triangle_j$ (or $\triangle_j$ *appears after* $\triangle_i$) if $u_i < l_j$. □

Definition 7 (Lifespan): Given a value $v \in \mathcal{V}$ and a sequence $s \in S$ where $p(v, s) \neq \emptyset$, the *lifespan* of v in s, denoted by $LS_{v,s}$, is defined to be an interval in form of (l, u) where (1) $l = \min p(v, s)$ and (2) $u = \max p(v, s)$. We define $LS_{v,s}.l$ to be l and $LS_{v,s}.u$ to be u. □

Consider Bob's sequence s. If v is "Titanic", then $p(v, s)$ = $\{1\}$ and thus the lifespan of v in s is (1, 1). If v is "Leonardo DiCaprio", then $p(v, s)$ = $\{1, 2\}$ and thus the lifespan of v in s is (1, 2).

We are ready to describe the bulky version of X_s.

Consider a sequence s in S. Let Y be the set of all values in s and their property values. Let h be the total number of possible values in Y. For each value $v \in Y$, we create an entry in form of $(v, LS_{v,s})$. The *bulky version* of X_s is equal to the table storing all these entries.

Example 1: Consider Bob's sequence s, "Titanic, The Aviator". Note that "Titanic" has the temporal position equal to 1 and "The Aviator" has the temporal position equal to 2. The attribute values of "Titanic" are

- "Titanic" (with label = 2),
- "1997" (with label = 3),
- "James Cameron" (with label = 5),
- "Leonardo DiCaprio" (with label = 7) and
- "Kate Winslet" (with label = 11)

The attribute values of "The Aviator" are

- "The Aviator" (with label = 13),
- "2004" (with label = 17),
- "Martin Scorsese" (with label = 19),
- "Leonardo DiCaprio" (with label = 7) and
- "Cate Blanchett" (with label = 23)

TABLE III
A TABLE SHOWING THE BULKY VERSION OF X_s

Value v	Lifespan of v in s
"Titanic"	(1, 1)
"1997"	(1, 1)
"James Cameron"	(1, 1)
"Leonardo DiCaprio"	(1, 2)
"Kate Winslet"	(1, 1)
"The Aviator"	(2, 2)
"2004"	(2, 2)
"Martin Scorsese"	(2, 2)
"Cate Blanchett"	(2, 2)

Thus, we have the set Y equal to {"Titanic", "1997", "James Cameron", "Leonardo DiCaprio", "Kate Winslet", "The Aviator", "2004", "Martin Scorsese", "Cate Blanchett" }. Then, we calculate the lifespan of each value $v \in Y$ in this sequence as shown in Table III. For example, if v = "Titanic", then the lifespan of v in s is $(1, 1)$. Similarly, if v = "Leonardo DiCaprio", then the lifespan of v in s is $(1, 2)$. Table III corresponds to the bulky version of X_s. □

Let l be the greatest length of a sequence in S. There are $O(lm)$ possible values in Y. Thus, the size of the bulky version of X_s is $O(lm)$. In Section V-B2, we will present a compressed version of X_s which contains only two positive numbers. Similarly, we can easily derive that the complexity of this phase is $O(lm)$.

Phase Query: Suppose that we are given a query sequence q and a sequence s in S. We want to check whether q matches s using the bulky version of X_s.

Definition 8 (Query-Aware Lifespan): Let s be a sequence in S and X_s be the bulky version of X_s for s. Given a query sequence q in form of $(v_1, v_2, ..., v_k)$, the *query-aware lifespan* of s with respect to q, denoted by $QA\text{-}LS(s, q)$, is defined to be $(\triangle_1, \triangle_2, ..., \triangle_k)$ where $\triangle_i$ is the lifespan of v_i in s for $i \in [1, k]$. □

Our strategy is to create the query-aware lifespan of s with respect to q according to the bulky version of X_s. This can be done in $O(k)$ time if the bulky version of X_s is indexed with a hash data structure. Then, according to the query-aware lifespan, we can determine whether q matches s efficiently, which will be described next.

Definition 9 (Non-Overlapping): Consider a sequence s and a query sequence q. Let the query-aware lifespan of s with respect to q be $(\triangle_1, \triangle_2, ..., \triangle_k)$. The query-aware lifespan is said to be *non-overlapping* if and only if for each $i, j \in [1, k]$ where $i < j$, $\triangle_i$ appears before $\triangle_j$. □

Definition 10 (Invalid): Consider a sequence s and a query sequence q. Let the query-aware lifespan of s with respect to q be $(\triangle_1, \triangle_2, ..., \triangle_k)$. The query-aware lifespan is said to be *invalid* if and only if there exist any two integers $i, j \in [1, k]$ such that $i < j$ and $\triangle_i$ appears after $\triangle_j$. □

With the above definitions, we have the following lemma about how to determine whether q matches s or not.

Algorithm 5 Algorithm **timespanCheck**(q, s, $QA\text{-}LS(s,q)$)

Input: a query q, s and $QA\text{-}LS(s,q)$
Output: whether s satisfies Requirement Sequential Order
1: **if** $QA\text{-}LS(s,q)$ is non-overlapping **then**
2: **return** true
3: **else**
4: **if** $QA\text{-}LS(s,q)$ is invalid **then**
5: **return** false
6: **else**
7: isMatch ← **checkMatch**(q, s)
8: **return** isMatch

Lemma 2: Consider a sequence s and a query sequence q. If the query-aware lifespan of s with respect to q is non-overlapping, then q matches s. If the query-aware lifespan of s with respect to q is invalid, then q does not match s. □

Suppose that we are given the query-aware lifespan of s with respect to q. Algorithm 5 shows the steps of checking whether q matches s according to the query-aware lifespan only. It is easy to verify that checking the conditions on whether the query-aware lifespan is non-overlapping (or invalid) takes $O(k)$ time. If these conditions are not satisfied, we need to execute the statements in lines 7-8 involving **checkMatch** which takes $O(lm)$ time. In our experiments, on average, there are about 90% cases that the query-aware lifespan is either non-overlapping or invalid. Thus, in most cases, the running time of **sequentialOrderCheck** is $O(k)$.

Let us analyze the storage complexity of the bulky version of X_s. Consider a particular sequence s. Let $|Y|$ be the average size of Y (i.e., the average number of possible values in a sequence s and their property values). For each value $v \in Y$, we need to store entry $(v, LS_{v,s})$. In our implementation, v is stored in form of a prime number and $LS_{v,s}$ is stored in form of two temporal positions. Since in all our experiments, the greastest possible values of each prime number and each temporal position can be represented by a 4-byte integer, each prime number and each temporal position are stored in a 4-byte integer. Thus, each entry occupies $4 \times 3 = 12$ bytes. Since there are $|Y|$ entries, the storage size of the bulky version of X_s for a particular sequence s is equal to $12|Y|$ bytes.

2) Compressed Version of X_s Based on a Pair of Numbers: Consider a sequence s in S. The bulky version contains h entries and each entry contains a value and its lifespan in s. This bulky version occupies a lot of space. Interestingly, the *compressed* version of X_s to be described contains only two positive numbers, which is quite space-efficient.

Specifically, given a sequence $s \in S$, the *compressed* version of X_s is defined to be equal to a pair of two numbers. The first number is called the *lower-bound sequential order number* of s, denoted by L_s, and the second number is called the *upper-bound sequential order number* of s, denoted by U_s. In Phase Preprocessing, these two numbers are to be found.

Phase Preprocessing: In Phase Preprocessing, for each sequence $s \in S$, we compute L_s and U_s as follows.

Let Y be the set of all values in s and their property values. Let h be the total number of possible values in Y. Recall that in the *bulky* version, for each value $v \in Y$, we create an entry in form of $(v, LS_{v,s})$ where $LS_{v,s}$ is the lifespan of v in s. Note that $LS_{v,s}$ is in form of $(LS_{v,s}.l, LS_{v,s}.u)$. However, in the *compressed* version, for each value $v \in Y$, we *conceptually* create a pair of congruence equations as follows. The following equations are in the format of Chinese Remainder Theorem. Note that $P(v)$ is the label of value v.

$$(L_s \bmod P(v)) = LS_{v,s}.l \quad (1)$$
$$(U_s \bmod P(v)) = LS_{v,s}.u \quad (2)$$

Since we have h values in Y, we conceptually generate h congruence equations in form of (1) and h congruence equations in form of (2).

We first consider the h equations for L_s in form of (1) and describe how to determine L_s, one of the two numbers stored in the compressed form of X_s. Specifically, since the labels of all values in Y are prime numbers, they are pairwise relatively prime. Note that $P(v)$ and $LS_{v,s}.l$ are given in Equation (1) where $v \in Y$. This is the equation format of Chinese Remainder Theorem. By using the Extended Euclidean algorithm, we can find a unique integer $L_s \in [0, N-1]$ where N is the product of the labels of all values in Y.

We can use a similar technique to find U_s by considering the h equations for U_s in form of (2).

Thus, we obtain that the final compressed version of X_s are two numbers, namely L_s and U_s.

Example 2: Consider Bob's sequence again. According to Example 1, we can obtain the bulky version of X_s (Table III).

In the compressed version, since we have 9 values in Y, we can conceptually formulate 9 congruence equations for L_s and 9 congruence equations for U_s. Take v = "Titanic" for illustration. Since its label is equal to 2 and its $LS_{v,s}.l$ is equal to 1, according to Equation (1), we create

$$(L_s \bmod 2) = 1$$

The other 8 congruence equations for L_s are:

$$(L_s \bmod 3) = 1 \quad (L_s \bmod 11) = 1 \quad (L_s \bmod 19) = 2$$
$$(L_s \bmod 5) = 1 \quad (L_s \bmod 13) = 2 \quad (L_s \bmod 23) = 2$$
$$(L_s \bmod 7) = 1 \quad (L_s \bmod 17) = 2$$

Similarly, we can construct the 9 congruence equations for U_s. By the Extended Euclidean Algorithm, we obtain $L_s = 134,918,071$ and $U_s = 7,436,431$. □

In Section IV, we know that the time complexity of finding a solution for L_s (and U_s) with m congruence equations is $O(m(\log n_p)^2)$ time where n_p is the largest prime numbers we use. Since each sequence s is associated with L_s (and U_s), the running time of this phase considering all sequences is equal to $O(nm(\log n_p)^2)$.

Phase Query: Suppose that sequence s satisfies Requirement Value Matching. Recall that Requirement Sequential Order is that the "matched" values in s have the same temporal ordering as the correspondence values in q. Algorithm 6 shows how we check whether s satisfies Requirement Sequential Order using

Algorithm 6 Algorithm **sequentialOrderCheck**(q, L_s, U_s) for Requirement Sequential Order

Input: a query q, L_s and U_s
Output: whether s satisfies Requirement Sequential Order
1: let query q be in the form of "$v_1, v_2, ..., v_k$"
2: **for** $i = 1$ to k **do**
3: $l_i \leftarrow (L_s \bmod P(v_i))$
4: $u_i \leftarrow (U_s \bmod P(v_i))$
5: $\triangle_i \leftarrow (l_i, u_i)$
6: call **timespanCheck**$(q, s, (\triangle_1, \triangle_2, ..., \triangle_k))$

Algorithm 7 Modular Algorithm for problem ASM

Input: a query q and a set S of identifying sequences
Output: a set of identifying sequences in S which are matched by q
1: $\mathcal{O} \leftarrow \emptyset$
2: **for** each $s \in C(q)$ **do**
3: isMatch $\leftarrow$ **checkMatch-Synopsis**(q, s)
4: **if** isMatch $=$ true **then**
5: $\mathcal{O} \leftarrow \mathcal{O} \cup \{s\}$
6: **return** $\mathcal{O}$

L_s and U_s. It is easy to see Algorithm 6 returns a correct solution with the following lemma.

Lemma 3: Consider a sequence s. Let L_s and U_s be the lower-bound sequential order number and the upper-bound sequential order, resepectively. Let Y be the set of all values in s and their property values. Given a value $v \in Y$, the lifespan of v in s is equal to (l, u) where $l = (L_s \bmod P(v))$ and $u = (U_s \bmod P(v))$. □

Let us analyze the storage complexity of the compressed version of X_s. Consider a particular sequence s. For this sequence s, we need to store two numbers, namely L_s and U_s. Consider number L_s which is computed based on $|Y|$ congruence equations. Note that L_s is at most the multiplication of the divisors of all congruence equations (i.e., $\prod_{v \in Y} P(v)$). In all of our experiments, each divisor (or each prime number) can be represented by a 4-byte integer. Thus, L_s can be represented by $|Y|$ 4-byte integers and thus the size of L_s is *at most* $4|Y|$ bytes. Note that $4|Y|$ is an *upper bound* of the size of L_s. In most cases, the exact size is smaller than $4|Y|$. Similarly, we can derive that the size of U_s is *at most* $4|Y|$ bytes. The storage size of the compressed version of X_s given a particular sequence is at most $4|Y| + 4|Y| = 8|Y|$ bytes. Since $|Y| \leq ml$, the storage size is at most $8ml$ bytes.

Let $\mathcal{N}'$ be the storage size of L_s (or U_s). Similarly, in the above analysis, $4|Y|$ is a loose upper bound on $\mathcal{N}'$. In other words, $\mathcal{N}' << 4|Y| (< 4ml)$. It is easy to verify that the running time of **sequentialOrderCheck** is $O(k\mathcal{N}')$ time if the running time of **timespanCheck** is $O(k)$ in most cases.

3) Comparison: Let us compare the storage of the compressed version with the storage of the bulky version. Consider X_s of both versions. The compressed version of X_s occupies at most $8|Y|$ bytes and the bulky version of X_s occupies $12|Y|$ bytes. Thus, the storage size of the compressed version of X_s is at most 2/3 of the bulky version of X_s. Now, we consider the storage size of the compressed/bulky *synopsis* containing not only X_s but also V_s. Note that V_s is the common component used by the compressed version and the bulky version. Note that since the storage size of X_s is equal to $4ml$, the compressed synopsis occupies at most $4ml + 8|Y| \leq 12ml$ bytes and the bulky synopsis occupies $4ml + 12|Y| \leq 16ml$. Thus, the storage size of the compressed synopsis is *at most* 3/4 $(= 12/16)$ of the storage size of the bulky synopsis. In the experimental results (Section VII), the real compression effect is more significant. On average, the storage size of the compressed synopsis is about 1/4 of the storage size of the bulky synopsis. The above theoretical analysis is based on the *upper bound* of the storage size of the compressed synopsis (instead of the *exact* storage size) and thus the bound of 3/4 is not quite tight.

C. Putting Two Requirements Together

In this section, we present algorithms to combine the two requirements together in addition to introducing an indexing technique called *inverted list*.

1) Phase Preprocessing: In addition to the steps we discussed previously, we describe an indexing technique called *inverted list*.

Suppose that each sequence $s \in S$ is given a unique sequence ID. Given a value $v \in \mathcal{V}$, the *inverted list* of v, denoted by $I(v)$, is defined to be a set of sequence IDs such that one of the values in each of these sequences has its property values equal to v. Given a query q, we define $C(q)$ to be a set of sequence IDs where each of the sequences with these IDs satisfies Requirement Value Matching. Thus, $C(q)$ is equal to $\cap_{i=1}^{k} I(v_i)$.

So, there are two major steps in Phase Preprocessing. The first step is to generate the inverted list of v for each value $v \in \mathcal{V}$. The second step is to generate the synopsis of each sequence s (where the synopsis is in form of a triplet (V_s, L_s, U_s)).

Note that both component V_s described in Section V-A and inverted lists are used for Requirement Value Matching. However, there are some differences. Firstly, inverted lists are used to locate sequences satisfying Requirement Value Matching by sequence IDs. Secondly, V_s can be used for Requirement Duplicate Value Matching but inverted lists cannot.

Note that the complexity of synopsis generation is $O(nlm)$ where l is the greatest length of a sequence. Generating inverted lists also takes $O(nlm)$ time. The overall complexity of this phase is equal to $O(nlm)$.

2) Phase Query: With the inverted list, we can modify Algorithm 1 to Algorithm 7. The differences come from the statements in Line 2 and Line 3. Firstly, in Line 2 of Algorithm 7, instead of processing all sequences in S, we process the sequences in $C(q)$ using the inverted list. Secondly, in Line 3 of Algorithm 7, instead of calling the original method **checkMatch** without using any synopsis, we call the new method **checkMatch-Synopsis** using the synopsis.

Algorithm 8 shows the algorithm for **checkMatch-Synopsis**. With Lemmas 1, 2 and 3, it is easy to verify the following theorem.

Algorithm 8 Algorithm **checkMatch-Synopsis**(q, s)

Input: a query q and an identifying sequence s
Output: whether q matches s

```
1: isValueMatch ← valueMatchCheck(q, V_s)
2: if isValueMatch = true then
3:     isSeqOrder ← sequentialOrderCheck(q, L_s, U_s)
4:     if isSeqOrder = true then
5:         return true
6: return false
```

Theorem 3: Algorithm 7 returns all sequences which are matched by q. □

Consider Algorithm 7. There are $O(n)$ sequences in $C(q)$. Consider a sequence in $C(q)$. We need to execute **checkMatch-Synopsis** (Algorithm 8). In this algorithm, we know that **valueMatchCheck** takes $O(k\mathcal{N})$ time and **sequentialOrderCheck**(q, L_s, U_s) takes $O(k\mathcal{N}')$ time in most cases. Thus, **checkMatch-Synopsis** takes $O(k(\mathcal{N}+\mathcal{N}'))$ time. Since $\mathcal{N} \geq \mathcal{N}'$, the time complexity of **checkMatch-Synopsis** becomes $O(k\mathcal{N})$. In conclusion, the overall time complexity of Algorithm 7 is $O(nk\mathcal{N})$.

VI. Frequent Subsequence Mining

In this section, we introduce a data mining problem, *frequent attribute-based subsequence mining*, which frequently makes use of the efficient operator for ASM (i.e., checking whether a query sequence matches a sequence in the dataset). Traditional frequent subsequence mining has been studied extensively in the literature [17, 8, 18, 6, 5]. It is useful to find frequent patterns in order to study customers' behaviors and temporal patterns. In this section, we propose frequent *attribute-based* subsequence mining (FASM). It is the same as the traditional mining except that we consider the property table.

Given a sequence p in form of "$v_1, v_2, ..., v_x$" where $v_i \in \mathcal{V}$ for $i \in [1, x]$, the *frequency* of p is defined to be the total number of sequences in S which are matched by p. Given a parameter θ which is a positive integer and a user parameter, a sequence p is said to be *frequent* if the frequency of p is at least θ. The problem of finding frequent attribute-based subsequence mining (FASM) is: given a parameter θ, we want to find all possible frequent sequences in S.

There are at least two categories of finding frequent subsequence mining in the literature. The first category is *singleton-based mining* while the second category is *set-based mining*.

In singleton-based mining, a sequence is represented in form of "$u_1, u_2, ..., u_l$" where $u_i \in \mathcal{V}$ for $i \in [1, l]$. At each timestamp, there is only at most one value $\in \mathcal{V}$ in the subsequence [1]. Singleton-based mining is to find all frequent subsequences in the dataset which have their frequencies at least a given threshold θ.

In set-based mining, a sequence is represented as a *set-sequence* in form of "$G_1, G_2, ..., G_l$" where $G_i \subseteq \mathcal{V}$ for $i \in [1, l]$ [21, 8]. At each timestamp, there can be more than one value $\in \mathcal{V}$ in the set-sequence (which is represented by a set G_i instead of a value in $\mathcal{V}$) [1]. In this category, the concept of subsequence (or set-subsequence) is defined differently as follows. Given a set-sequence g in form of "$G_1, G_2, ..., G_l$" where $G_i \subseteq \mathcal{V}$ for $i \in [1, l]$ and another set-sequence h in form of "$H_1, H_2, ..., H_{l'}$" where $H_i \subseteq \mathcal{V}$ for $i \in [1, l']$, g is said to be a *set-subsequence* of h if there exist l integers, namely $j_1, j_2, ..., j_l$, such that (1) for each $i \in [1, l]$, $G_i \subseteq H_{j_i}$, and (2) $1 \leq j_1 < j_2 < ... < j_l \leq l'$. If g is a set-subsequence of h, then h is said to *contain* g. In set-based mining, each set-sequence in the dataset is in form of "$G_1, G_2, ..., G_l$". The frequency of a set-sequence g is equal to the total number of set-sequences in the dataset containing g. Set-based mining is to find all frequent set-subsequences in the dataset which have their frequencies at least a given threshold θ.

Note that our FASM problem is a special case of the set-based mining. That is, all sequences found in our FASM problem can be found in the set of set-sequences found in the set-based mining.

Since our FASM problem is a special case of the set-based mining, we adapt an existing algorithm in the literature of set-based mining for FASM, by using our efficient query operator in ASM. In the literature, most algorithms for set-based mining requires to enumerate some potential candidates as the output and count the total number of sequences in the dataset which are matched by each candidate (query sequence). Our operator for ASM can be used in the counting step. Whenever we need to obtain the count for each candidate, we can perform our operator. Since these algorithms involves a large set of candidates and their counting step is not optimized, if their counting step is replaced by our operator, the efficiency of the algorithms can be improved a lot. In the experiment, we use the algorithm in [8] to illustrate how the operator can improve the performance of the algorithm.

Our operator can also be used to solve a more general problem, the set-based mining, using the above approach. For each candidate in form of "$G_1, G_2, ..., G_l$" where G_i is a set of values for $i \in [1, l]$, we generate all possible sequences in form of "$v_1, v_2, ..., v_l$" where $v_i \in G_i$ for $i \in [1, l]$. Each generated sequence can be regarded as a query in ASM.

VII. Empirical Experiments

In order to verify the efficiency of our algorithm, we implemented three algorithms in C/C++, namely *Naive*, *MA* and *MAI*. In Section III, we described two possible straightforward approaches for problem ASM. Since the first approach takes an exponential time with respect to the length of the query length which is not scalable, we implemented the second straightforward approach (Algorithm 1) and call it *Naive*. *MA* is Modular Algorithm *without* inverted list which is Algorithm 7 where $C(q)$ in line 2 is replaced by S. *MAI* is Modular Algorithm *with* inverted list which is Algorithm 7. For *MA* and *MAI*, we adopt the compressed version of the synopsis because it occupies less storage and have nearly the same execution time in Phase Preprocessing and Phase Query compared with the bulky version.

All the experiments were performed on a 2.4GHz PC with 4.0GB RAM, on a Linux platform. We did experiments on both synthetic and real datasets. For the synthetic datasets,

TABLE IV
DEFAULT VALUES

Parameters	Values
n (Number of sequences)	**250k**, 500k, 750k, 1M
μ (Average sequence length)	20, **40**, 60, 80, 100
k (Query length)	5, **10**, 15, 20
d (Domain size of each attribute)	**100**, 200, 30, 40
m (Number of attributes)	**5**, 10, 15, 20

we first generate the length of the sequence following a given Gaussian distribution with its mean equal to μ and its standard derivation equal to 5 where μ is a user parameter representing the average length of a sequence. In addition to dataset size n and the average length of each sequence μ, the synthetic data generator also simulates the number of attributes m, the size of each attribute domain d and the length of each query k. We assume these three values are fixed for all the sequences in a single dataset. In order to find at least one matching in the whole dataset, we extract each query sequence from an arbitrary sequence in the dataset. The values of each parameter used in the experiments are given in Table IV, where the default values are in bold. Finally, we generate the synthetic datasets according to every distinct parameter setting in Table IV.

In the experiments, we evaluate the algorithms with four measurements: (1) *Preprocessing Time*, (2) *Execution Time*, (3) *Storage* and (4) *Compression Ratio*. (1) Preprocessing time of *MA* and *MAI* corresponds to the time cost in Phase Preprocessing. *Naive* has no preprocessing step. So, we do not consider it. (2) Query time refers to the time an algorithm takes to answer 100 queries. Since the query time of *MA* (*MAI*) using the compressed synopsis is similar to the query time of *MA* (*MAI*) using the bulky synopsis, we only report *MA* (*MAI*) using the compressed synopsis. (3) Storage is the total memory consumption used for each data structure and the original dataset. The storage of *Dataset Size* is the memory occupied by the sequence data. The storage of *Inverted List* is the memory occupied by the inverted list. The storage of *Compressed Synopsis* is the memory occupied by the compressed version of X_s and V_s, while the storage of *Bulky Synopsis* is the storage occupied by the bulky version of X_s and V_s. (4) *Compression Ratio* is the ratio of the storage of Compressed Synopsis to that of Bulky Synopsis.

We study the effects of n, μ, k, d and m as follows.

Effect of database size n: Figure 1(a) shows that the preprocessing in *MAI* is slightly larger than that of *MA*. It is because that *MAI* needs to generate the inverted list, but *MA* does not. In Figure 1(b), when the execution time of *Naive* increased sharply when n increases, the execution time of *MA* and *MAI* increased slightly. As expected, the storage of *Bulky Synopsis* is much larger than that of *Dataset Size*, *Inverted List* and *Compressed Synopsis* in Figure 1(c). The compression ratio is around 24% as shown in Figure 1(d).

Effect of sequence length μ: Figures 2(a), (b) and (c) have similar trends as Figures 1(a), (b) and (c). In Figure 2(c), each data structure increases with μ. Note that the compressed synopsis also increases. In the compressed synopsis, each sequence is compressed into a 3-number synopsis which is independent of the sequence length. So, apparently, it seems that it should not increase with the sequence length. However, when the sequence becomes longer, the compressed synopsis needs much larger prime numbers and thus the synopsis representation needs more storage. Notice that, in Figure 2(d), as μ increased, the compression ratio decreased slowly, which means the longer the sequence is, the smaller storage the compressed synopsis occupies compared with the bulky synopsis.

Effect of k: As expected, the length of the query sequence does not affect the processing time and the storage of every data structure, as shown in Figure 3(a), Figure 3(c) and Figure 3(d). From Figure 3(b), we can see that the execution time of *MA* and *MAI* remained unchanged while that of *Naive* increased slightly.

Effect of d: As d increases, the storage of the compressed synopsis increased slightly, so that the compression ratio also increased, as shown in Figure 4(c) and Figure 4(d). When d increased, the diversity of the sequences in the database also increased sharply, each generated sequence is much more dissimilar to other sequences. Consequently, the execution time of *MA* and *MAI* decreased in Figure 4(b).

Effect of m: When m increased, the processing time and storage increased, as shown in Figure 5(a) and Figure 5(c). But the compression ratio remained around 24% in Figure 5(d). But in Figure 5(b), the execution time of *MAI* remained when m increases. However, the execution time of *Naive* increased significantly. It means that *MAI* can deal with sequences that have a large number of attributes efficiently.

Besides the synthetic datasets, we also did experiments on three real datasets: Netflix [22], BookX [23], and Genealogy [24]. (1) Netflix is a famous movie rental company. We process the rating record dataset provided by Netflix to generate a rating sequence dataset through grouping the ratings by *customerID* (the identification of a customer) and sorting them by the rating date. (2) BookX (BookCrossing) is an online book searching and rating website. We download the ratings dataset and use a similar method to generate a sequence for every reader. (3) Genealogy dataset is collected by ourselves, which contains biographic sequences of 1000 researchers. Some statistics of the three datasets are shown in Table V. Each query is generated by randomly selecting a subsequence of a sequence in a real dataset such that the length of the subsequence is equal to a specified length. generated by randomly selecting subsequences of sequences in each In this table, *No. of Elements* is the number of elements appearing in this dataset, and *Avg. Duplicates* is the average proportion of duplicate attribute values in one sequence.

The first four columns in Table VI shows the execution time on the three real datasets. The execution time of *MAI* is much smaller than that of *Naive* in every real dataset. The last column in Table VI shows that the compression ratio of Netflix and BookX is around 30%, a little higher than that of the synthetic datasets. We summarize the other statistics of the experiments on real datasets. The greatest prime numbers used in encoding the three real datasets is 4,863,427. The greatest number used in the compressed synopsis contains 2100 digits.

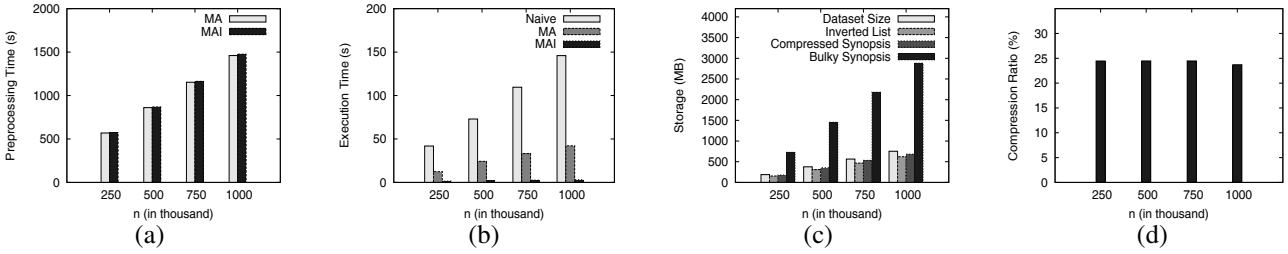


Fig. 1. Effect of n (Dataset size)

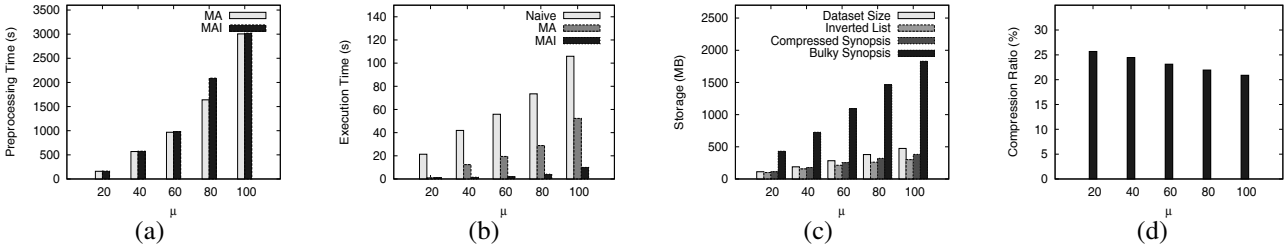


Fig. 2. Effect of μ (Average length of a sequence)

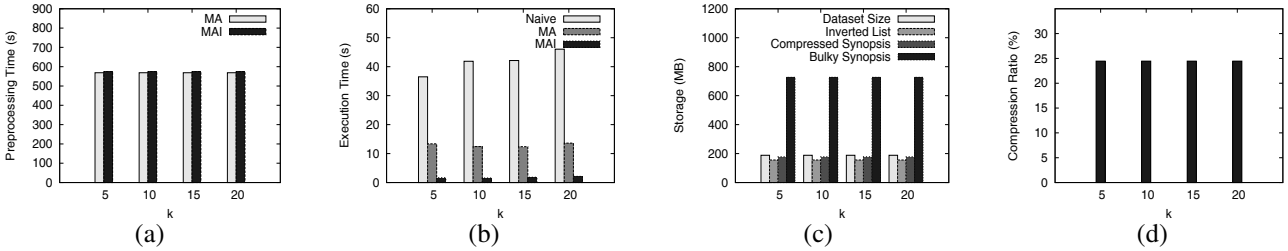


Fig. 3. Effect of k (Length of a query sequence)

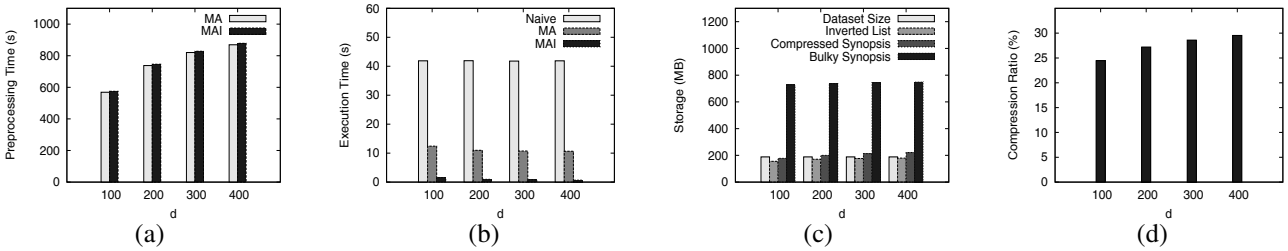


Fig. 4. Effect of d (Size of the domain of each attribute)

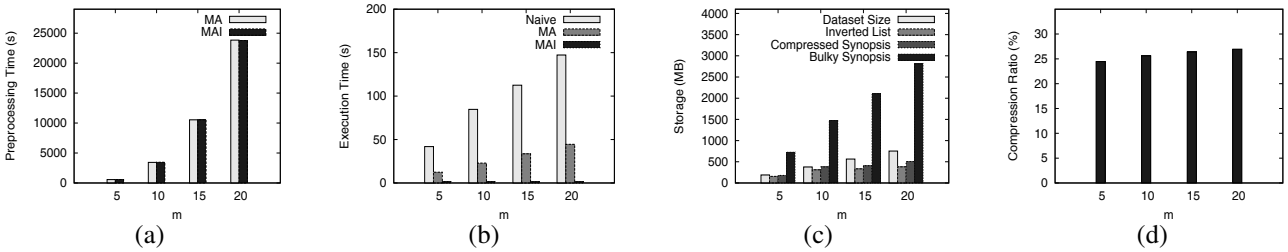


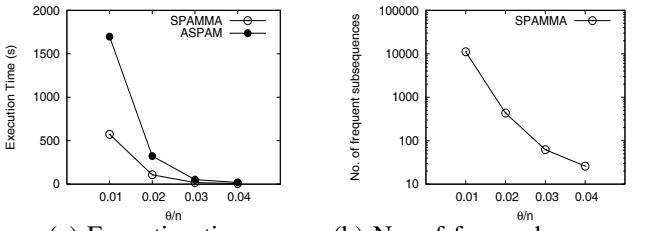
Fig. 5. Effect of m (Number of attributes of each object)

TABLE V
STATISTICS OF REAL DATASETS

Datasets	n	m	μ	No. of Elements	Avg. Duplicates
Netflix	478905	3	157	7653	11.2%
BookX	91400	3	11.7	262554	4.2%
Genealogy	1000	3	2.5	1553	13.3%

TABLE VI
EXECUTION TIME ON REAL DATASETS

Dataset	**Execution Time (s)**			**Compression Ratio (%)**
	Naive	**MA**	**MAI**	
Netflix	205.939	85.446	6.744	31.382
BookX	2.489	2.034	0.681	30.978
Genealogy	0.011	0.004	0.003	21.699



(a) Execution time (b) No. of freq. subsequences

Fig. 6. Result for FASM in BookX by *SPAMMA*

Although this number is large, the modular operation over this number can be done efficiently with the GMP library [20].

Result for FASM: We conducted experiments for the FASM problem on the BookX dataset. We insert *MA* as an operator in SPAM [8] for FASM. Due to the bitmap representation of database, SPAM cannot process attribute-based sequences of length more than 64, which are common in Netflix and BookX. The *SPAM* using our operator MA is denoted by *SPAMMA*. We compare *SPAMMA* with the *SPAM* algorithm without using our *MA* operator, denoted by *ASPAM*. The experimental results can be found in Figure 6 where θ/n is the frequency threshold in fraction.

In Figure 6(a), the execution time of *SPAMMA* is much smaller than *ASPAM*. When θ is larger, fewer subsequences are checked whether they are frequent according to the given threshold θ. So, the resulting frequent subsequence set is also smaller, as shown in Figure 6(b).

Case Study for FASM: By running *SPAMMA*, we found some interesting frequent subsequences. For example, when θ/n is set to 0.01, we can find "<Deutscher Taschenbuch Verlag>, <Piper>" as a frequent subsequence. "Deutscher Taschenbuch Verlag" is a publisher, while "Piper" is a book. Note that "Deutscher Taschenbuch Verlag" and "Piper" belong to different attributes. Another interesting case is a frequent subsequence, "<She's Come Undone>". It is a book with two versions published by two different publishers: one is Washington Square Press, and the other is Pocket Books. In the result, "<She's Come Undone, Washington Square Press>" is also a frequent subsequence, but "<She's Come Undone, Pocket Books>" is not. It means that compared with the Pocket Books version, readers prefer the Washington Square Press version.

Summary: *Compressed Synopsis* gives a very low compression rate. On average, the compression rate is about 25%. In most cases, the execution time of *MAI* is an order of magnitude better than that of *Naive* . In the real dataset Netflix, the execution time of *MAI* is 3% of that of *Naive*.

VIII. Conclusion

In this paper, we propose a new problem called *Attribute-based Subsequence Matching Problem* which has many applications. We propose an efficient algorithm for this problem using Chinese Remainder Theorem to compress each sequence into a triplet of numbers. We also illustrate how this problem can be used for mining frequent subsequences. Finally, we conducted experiments to show that our algorithm is very efficient, nearly two orders of magnitude better than the straightforward method. There are a lot of possible future directions. One direction is to consider the problem with additional constraints like gap constraints [6] commonly adopted in bioinformatics. Another direction is to study how other data mining problems about subsequence matching can be extended when the property table is considered.

Acknowledgements: The research of Yu Peng, Raymond Chi-Wing Wong and Liangliang Ye is supported by HKRGC GRF 621309 and Direct Allocation Grant DAG11EG05G. The research of Philip S. Yu is supported by US NSF through grants DBI-0960443, IIS 0905215, OISE-0968341 and OIA-0963278, and Google Mobile 2014 Program.

References

[1] G. Dong and J. Pei, *Sequence Data Mining (Advances in Database Systems)*. Springer-Verlag New York, Inc., 2007.
[2] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *SIGMOD Rec.*, 1994.
[3] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos, "Approximate embedding-based subsequence matching of time series," in *SIGMOD*, 2008.
[4] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang, "Ranked subsequence matching in time-series databases," in *VLDB*, 2007.
[5] L. Boasson, P. Cegielski, I. Guessarian, and Y. Matiyasevich, "Window-accumulated subsequence matching problem is linear," in *PODS*, 1999.
[6] M. Zhang, B. Kao, D. Cheung, and K. Yip, "Mining periodic patterns with gap requirement from sequences," in *SIGMOD*, 2005.
[7] S. lyong Lee, S. ju Chun, D. hwan Kim, J. hong Lee, and C.-W. Chung, "Similarity search for multidimensional data sequences," in *ICDE*, 2000.
[8] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *KDD*, 2002.
[9] R. She, F. Chen, K. Wang, M. Ester, J. L. Gardy, and F. S. L. Brinkman, "Frequent-subsequence-based prediction of outer membrane proteins," in *KDD*, 2003.
[10] A. Pol and T. Kahveci, "Highly scalable and accurate seeds for subsequence alignment," in *BIBE: Proceedings of the Fifth IEEE Symposium on Bioinformatics and Bioengineering*, 2005.
[11] G. D. Stormo, "Dna binding sites: representation and discovery." in *Bioinformatics*, 2000.
[12] X. Liu, D. L. Brutlag, and J. S. Liu, "Bioprospector: Discovering conserved dna motifs in upstream regulatory regions of co-expressed genes," in *Pac. Symp. Biocomput*, 2001.
[13] C. Wu, M. Berry, S. Shivakumar, and J. McLarty, "Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition," in *Machine Learning*, 1995.
[14] P. Geurts, A. B. Cuesta, and L. Wehenkel, "Segment and combine approach for biological sequence classification," in *In: Proceedings of IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, 2005.
[15] T.F.Smith and M.S.Waterman, "Identification of common molecular subsequences," in *Journal of Molecular Biology*, 1981.
[16] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," in *Journal of molecular biology*, 1990.
[17] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang, "Exploratory mining and pruning optimizations of constrained associations rules," in *SIGMOD Rec.*, 1998.
[18] J. Pei and J. Han, "Can we push more constraints into frequent pattern mining?" in *KDD*, 2000.
[19] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. Oxford, England: Clarendon Press, 1979.
[20] "http://gmplib.org/."
[21] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *EDBT*, 1996.
[22] Y. Koren, "Collaborative filtering with temporal dynamics," in *KDD*. New York, NY, USA: ACM, 2009, pp. 447–456.
[23] "http://www.informatik.uni-freiburg.de/∼cziegler/bx/."
[24] "http://www.cse.ust.hk/∼raywong/genealogy/."