MINING SEQUENTIAL PATTERNS IN DENSE DATABASES

Karam Gouda[1] and Mosab Hassaan[1]

[1]Faculty of Computers and Informatics, Information System Department, Benha University, Egypt
karam_g@hotmail.com and mosab_ha@yahoo.com

ABSTRACT

Sequential pattern mining is an important data mining problem with broad applications, including the analysis of customer purchase patterns, Web access patterns, DNA analysis, and so on. We show on dense databases, a typical algorithm like Spade algorithm tends to lose its efficiency. Spade is based on the used of lists containing the localization of the occurrences of pattern in the sequences and these lists are not appropriated in the case of dense databases. In this paper we present an adaptation of the well-known diffset data representation [12] with Spade algorithm. The new version is called dSpade. Since diffset shows high performance for mining frequent itemsets in dense transactional databases, experimental evaluation shows that dSpade is suitable for mining dense sequence databases.

KEYWORDS

Sequential Patterns, Data Mining, Dense Databases

1. INTRODUCTION

The sequential pattern mining problem is an important problem in the data-mining field with numerous practical applications, including consumer shopping transaction analysis, mining web logs, mining DNA sequences, and so on. For example, consider the sales database of a bookstore, where the objects represent customers and the attributes represent authors or books. Let's say that the database records the books bought by each customer over a period of time. The discovered patterns are the sequences of books most frequently bought by the customers. An example could be that, "70% of the people who buy introduction to visual Basic and introduction to C++ also buy introduction to Perl within a month." Stores can use these patterns for promotions, shelf placement, etc.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in [2]: Given a set of sequences, where each sequence consists of a list of elements and each element consists of set of items, and given a user-specified min_support threshold, sequential pattern mining is to find all of the frequent subsequences. i.e., the subsequences whose occurrence frequency in the set of sequences in no less than min_support.

In this paper, we consider the problem of sequential patterns in dense databases. We show on dense databases, a typical sequential pattern mining algorithm like Spade algorithm [10] tends to lose its efficiency. Spade is based on the use of lists containing the localization of the occurrences of pattern in the sequences and these lists are not appropriated in the case of dense databases and lead to increase extraction operation. For example, Figure 1 shows the behaviour of the Spade algorithm on dense datasets. The results of the experiments presented in Figure 1 correspond to extractions on two datasets: data1 and data2. data1 contain the same sequences in data2 but we increase only the average item per element in each sequence. This convert data1 to be dense dataset. The curves of Figure 1 represent the costs (in term of execution time) for the

DOI: 10.5121/ijdms.2011.3112

extraction of different amounts of frequent patterns on each dataset, i.e, for different support thresholds. From Figure 1, Spade execution time is much more importantly on data1 (dense dataset).

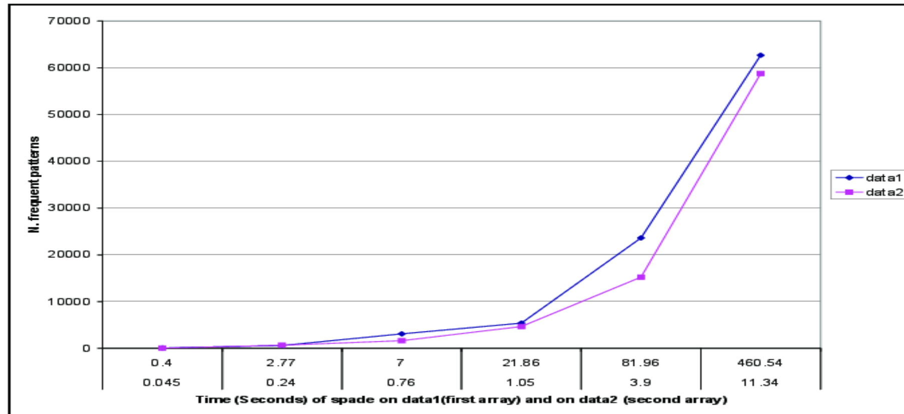

Figure 1. Evolution of SPADE execution time on dense dataset

The main contribution of this paper is to show that this extra extraction cost can be reduced drastically using a more compact information representation. We propose such a representation and represent an extension of Spade, called dSpade, that operates directly on it. dSpade uses diffseqs lists to find all frequent sequences. We show that in practice it can be used to mine efficiently the complete set of frequent sequences in dense databases. The rest of this paper is organized as follows. In section 2 we present the problem definition of mining sequential patterns and in section 3 we discuss the related work. Section 4 presents in a synthetic way the Spade-based algorithm before to introduce in section 5 our contribution which is a novel data representation called diffseq . Section 6 presents experimental results that illustrate how dSpade gains in efficiency compared to Spade in the case of dense database. We conclude in section 7 by a summary and directions for future work.

2. PROBLEM DEFINITION

Let $I = \{i_1, \ldots, i_m\}$ be a set of items. We call a subset $X \subseteq I$ an *itemset* or *transaction* and we call $|X|$ the *size* of X. A sequence is an ordered list of transactions. A *sequence* S is denoted by $S = < t_1, t_2, t_3, \ldots, t_n >$, where t_i is a transaction and it is also called an element of the sequence. An item can occur at most once in an element of a sequence, but it can occur multiple times in different elements of a sequence. The size, n, of a sequence S is the number of transactions in S, i.e. $|S|$. The length l of a sequence S is defined as $l = \sum_{i=1}^{n} |t_i|$. A sequence with length l is called an l-sequence (or l-pattern). A sequence $\alpha = < a_1, a_2, a_3, \ldots, a_n >$ is called a subsequence of another sequence $\beta = < b_1, b_2, b_3, \ldots, b_m >$ and β a supersequence of α, denoted as $\alpha \subset \beta$, if there exist integers $1 \le j_1 < j_2 < \ldots < j_n \le m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, $a_3 \subseteq b_{j3}$,..., $a_n \subseteq b_{jn}$. A sequence database D is a set of tuples $< Sid, S >$, where Sid is a sequence_id and S is a

sequence. A tuple $< Sid, S >$ is said to contain a sequence α, if α is a subsequence of S, i.e., $\alpha \subset S$.

The support of a sequence S, denoted by $sup(S)$ is the total number of tuples in database that contain this sequence. This support is called the *absolute support* of this pattern and we use it throughout this paper while the *relative support* is defined as the percentage of tuples in database that contain S. we will only use the relative support in experimental results.

Problem statement Given a user-specified support threshold *min_sup*, the sequence S is called *frequent* if $sup(S) \geq min_sup$, the problem of mining sequential patterns is to find the complete set of frequent patterns in a database D with respect to a given support threshold *min_sup.*

Example 2.1 *Given sequence database* D *in Table 1 and min_support = 3. The set of items in the database is* $I = \{a, b, c, d, e\}$.

Table 1. A sequence database, D

Sequence_id	Sequence
10	$< abc, abc, abd, ae >$
20	$< abc, abc, acd >$
30	$< abe, abce, abe >$
40	$< abc, ab >$

sequence $< abc, abc, abd, ae >$ has four elements: $(abc), (abc), (abd)$ and (ae), where items a, b and c appear more than once respectively in different elements. It is also a 11-sequence since there are 11 instances appearing in that sequence. Item a happens four times in this sequence, so it contributes 4 to the length of the sequence. However, the whole sequence contributes only one to the support of $< a >$. Also, sequence $< abc, ab >$ is a subsequence of $< abc, abc, abd, ae >$. Since sequences 10, 20, 30 and 40 contain subsequence $S = < abc, ab >$, i.e. $sup_D(S) = 4 > 3 = min_sup$ then S is a sequential pattern (frequent sequence) of size 2 and length 5.

3. RELATED WORK

In data mining community the computation of the sequential patterns has been studied since 1995, e.g.[2, 9, 6, 10, 8, 3]. It has lead to several algorithms that can process huge sets of sequences. These algorithms use three different types of algorithms approaches according to the way they evaluate the support of sequential pattern candidates.

Horizontal Approach: These are exemplified by GSP (**G**eneralized **S**equential **P**attern) algorithm [9]. GSP is a bottom-up, breadth first search. The structure of the GSP algorithm for finding sequential patterns is very similar to the Apriori Algorithm [1], that is, it is Apriori-based algorithm for sequential pattern mining. The algorithm makes multiple passes over the data. In the first pass it determines the support of each item. Frequent items (items with support greater than or equal to min_sup) compose a 1-element frequent sequences. Each subsequent pass starts with a seed set "the frequent sequences found in the previous pass". The seed set is

used to generate new potentially frequent sequences, called candidate sequences. The support for these candidate sequences is found during the pass over the data. At the end of the pass, the algorithm determines which of the candidate sequences are actually frequent. These frequent sequences become the seed for the next pass. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated. The algoritms in [2, 6] also follow a horizontal approach.

Vertical Approach: Zaki proposed another approach for mining frequent sequential patterns, called Spade (**S**equential **PA**ttern **D**iscovery using **E**quivalence Classes) [10]. The main idea in this approach is a clustering of the frequent sequences based on their common prefixes and the enumeration of the candidate sequences, thanks to a rewriting of the database (loaded in main memory). Spade needs only three database scans in order to extract the sequential patterns. The first scan aims at finding the frequent items, the second at finding the frequent sequences of length two and the last one associate to frequent sequences of length two, a table of the corresponding *sequence_id* and *itemset_id* (or *transaction_id*) in the database (called id-lists). Based on this representation in main memory, the support of the candidate of length k is the result of join operations on the tables related to the frequent sequences of length $k-1$ to generate this candidate (so, every operation after the discovery of frequent sequences having length two is done in memory). The detail of spade will descried in Section 4. Spam (**S**equential **PA**ttern **M**ining) [3] is also a vertical approach uses bit-vectors to represent the id-lists.

Projection Approach: PrefixSpan (**PREFIX**-project Sequential **PA**tter**N** **M**ining) [8] follows a database projection approach, which is a hybrid between the horizontal and vertical approachs. Given any prefix sequence P, the main idea is to project the horizontal database, so that the projected (or conditional) database contains only those sequences that have prefix P. The frequency of extensions of P can be directly counted in the projected database. Via recursive projections all frequent sequences can be enumerated. PrefixSpan is a hybrid method, since the projected database is equivalent to a horizontal representation of the id-lists of sequences that share a given prefix P.

4. THE SPADE ALGORITHM [10]

In this section, we recall the principle of the Spade algorithm. Spade is an algorithm proposed to find frequent sequences using efficient lattice search techniques and simple joins. All the sequences are discovered with only three passes over the database, it also decomposes the mining problem into smaller subproblems, which can be fitted in main memory. In this approach, the sequence database is transformed into a vertical id-list database format, in which each item is associated with a list of all sequence identifer (Sid) and transaction identifer (Tid). The vertical database of Table 1 is shown in Table 2. From Table 2, the support count of item e is 2 since it occured in sequences 10 and 30. By scanning the vertical database, frequent 1-sequences can be generated with the minimum support. For 2-sequence, the original database is scanned again and the new vertical to horizontal database is created by grouping those items with Sid and in increase order of Tid [10]. By scanning the vertical to horizontal database, 2-sequences are generated. All the 2-sequence found are used to construct the lattice, which is quite large to fitted in main memory. However the lattice can be decomposed to different classes, sequences that have the same prefix items belong to the same class. By decomposing, the lattice is partitioned into small parts that can be fitted in main memory. During the third scanning of the database all those longer sequences are enumerated by using joining over relevant id-lists.

Table 2. Id-lists for 1-sequences

id-list(a)		id-list(b)		id-list(c)		id-list(d)		id-list(e)	
Sid	**Tid**	**Sid**	**Tid**	**Sid**	**Tid**	**Sid**	**Tid**	**Sid**	**Tid**
10	1	10	1	10	1	10	3	10	4
10	2	10	2	10	2	20	3	30	1
10	3	10	3	20	1	---		30	2
10	4	20	1	20	2	---		30	3
20	1	20	2	20	3	---		---	
20	2	30	1	30	2	---		---	
20	3	30	2	40	1	---		---	
30	1	30	3	---		---		---	
30	2	40	1	---		---		---	
30	3	40	2	---		---		---	
40	1	---		---		---		---	
40	2	---		---		---		---	

4.1 Frequency Counting

Given prefix class or subclass, one performs joining of the id-lists of all pairs of class elements, and checks if minimum support is met . There are two major operations : *Merge* and *Join* operations. A k-pattern X having ($k-1$)-pattern p as prefix and item s as suffix is called an *event pattern*, denoted $< ps >$, if s occurs at the last itemset of p. On the other hand, if s stands alone as the last itemset in X, X is called a *sequence pattern* and is denoted $< p, s >$. For example, pattern $< ab, c, bdf >$ having pattern $< ab, c, bd >$ as prefix and item f as suffix is an event pattern. Pattern $< ab, c >$ whose prefix is ab and suffix c is a sequence pattern. Let A_i and A_j be two patterns having the same prefix p with respective suffix s_1 and s_2. The merge operation used to generate a new pattern R depends on the form of A_i and A_j (i.e., an event pattern or a sequence pattern). The form of R determines the kind of join performed to compute id-list(R) from id-list(A_i) and id-list(A_j). If R is an event pattern (resp. a sequence pattern) the join is made using a procedure called Equality Join (resp. Temporal Join). We first present the generation cases (merge operations) and then describe the join operations.

- **Case I:** When A_i and A_j are event patterns. That is, they are of the form $A_i =< ps_1 >$ and $A_j =< ps_2 >$. The pattern generated by merge is $R =< ps_1 s_2 >$ and its id-list(R)=Equality-Join(id-list(A_i), id-list(A_j)).
- **Case II:** When A_i is an event pattern and A_j a sequence pattern. That is, they are of the form $A_i =< ps_1 >$ and $A_j =< p, s_2 >$. The pattern generated by merge is $R =< ps_1, s_2 >$ and we have id-list(R)=Temporal-Join(id-list(A_i), id-list(A_j)).
- **Case III:** When A_i and A_j are sequence patterns. That is, they are of the form $A_i =< p, s_1 >$ and $A_j =< p, s_2 >$. If $s_1 \neq s_2$, three patterns are generated:

1-Merge produces $R =< p, s_1 s_2 >$ and id-list(R)=Equality-Join(id-list(A_i), id-list(A_j)).

2-Merge produces $R =< p, s_1, s_2 >$ and id-list(R)=Temporal-Join(id-list(A_i), id-list(A_j)).

3-Merge produces $R =< p, s_2, s_1 >$ and id-list(R)=Temporal-Join(id-list(A_j), id-list(A_i)).

On the other hand, if $s_1 = s_2$, only one pattern is generated:

1- Merge produces $R =< p, s_1, s_2 >$ and id-list(R)=Temporal-Join(id-list(A_i), id-list(A_j)).

Note that in all cases where temporal joining is required, R is a sequence pattern, that is, it is constructed in a sequence-extension step: The suffix of A_j (or A_i) stand as a new itemset in R. Also, in all cases where equality joining is required, R is an event pattern, that is, it is constructed in an itemset-extension step: The suffix of A_j is added to the last itemset of A_i to produce R. Before going on and explain both Equality and Temporal Join, please note that, id-list(A_i) and id-list(A_j) store the positions where the patterns A_i and A_j occur in every database sequence. Then by joining these two id-lists, we have to determine where R occurs. That is, id-list(R) should only maintain the information needed to compute the support of R and the id-lists of the patterns that will be generated using R.

Here we explain each of the joining functions.

•**Temporal-Join**(id-list(A_i), id-list(A_j)): To compute id-list(R), we need to check for a follow relationship. That is, for a given pair (s', t') in id-list(A_i), we check whether there exists a pair (s'', t'') in id-list(A_j) with the same $s' = s''$, but with $t'' > t'$. If this is true, it means that the suffix item of A_j follows the suffix item of A_i in sequence R. In other words, the sequence s' contains the pattern R, and the pair (s'', t'') is added to its id-list.

• **Equality-Join**(id-list(A_i), id-list(A_j)): To compute id-list(R), we simply need to check for equality of (Sid, Tid) pairs in id-list(A_i) and id-list(A_j).

Example 4.1 *Let us consider the Id-list of items a and b represented in Table 2, the Id-list of patterns $< ab >$ (Equality-Join) and $< a, b >$ (Temporal-Join) is represented in Table 3. The support of a sequence is the cardinality of sequences in its id-list then we have* $sup(< ab >) = sup(< a, b >) = 4$.

Table 3: Id-lists for $< ab >$ and $< a,b >$

id-list($< ab >$)		**id-list($< a,b >$)**	
Sid	**Tid**	**Sid**	**Tid**
10	1	10	2
10	2	10	3
10	3	20	2
20	1	30	2
20	2	30	3
30	1	40	2
30	2	---	
30	3	---	
40	1	---	
40	2	---	

5. DIFFSEQ DATA REPRESENTATION

In this section, we present an adaptation of the well-known diffset data representation [12] to be used in sequence mining. To the best of our knowledge, this is the first time one adjusts diffset data structure to be used in mining sequential patterns. Our new structure is referred to as *diffseq*.

5.1 Introducing Diffseq

To explain the idea we first consider the database as consisting of only one database sequence S. Let P be a sequential pattern, define a diffseq vertical data representation associated with P with respect to S as follows:

$$diffseq_s(P) = \{P^s_{f_0}\} \cup d_s(P)$$

where $P^s_{f_0}$ is an integer represents the first occurrence of the last itemset of P in the given database sequence s and $d_s(P)$ is the set of itemset ids which are greater than $P^s_{f_0}$ and do not contain the last itemset of P, i.e., the diffset of P. As an example consider the database sequence $s =< ab, bd, abcd, e >$, the diffseq of the sequential pattern $< a >$ with respect to s is given as follows: $diffseq_s(< a >) = \{< a >^s_{f_0}\} \cup d_s(< a >) = \{1\} \cup \{2,4\} = \{1,2,4\}$ where $(< a >)^s_{f_0} = 1$ is the id of first transaction in s containing a, the last itemset of the pattern $< a >$, and $d_s(< a >) = \{2,4\}$ is the set of transaction ids which are greater than $< a >^s_{f_0}$ and do not contain a. Likewise $diffseq_s(< b >) = \{1,4\}$ and $diffseq_s(< c >) = \{3,4\}$.

5.2. Joining Diffseqs

Now we discuss how to get the diffseqs of longer patterns that constructed in sequence-extension or Itemset extension steps.

5.2.1. The diffseqs of longer patterns that constructed using Sequence-extension step:
Suppose that the sequence $A =< px >$ is extended in a sequence step to get the pattern $C =< px, y >$, where p stands for the prefix, $|p| \geq 0$, and x and y are database items. There are two methods by which we can construct $diffseq_s(C)$. The first one is by joining $diffseq_s(A)$ with $diffseq_s(< y >)$, where y is the item used in extending A. The second method is by joining $diffseq_s(A)$ with $diffseq_s(B)$, where $B =< p, y >$, as in the equivalence class approach [10]. From the diffseq definition, the diffseq of a given pattern P is completely determined by defining the two terms: $P^s_{f_0}$ and $d_s(P)$. Thus in the two methods we have to provide definitions to both terms.

Method I: The following equations define the two terms:

$$C^s_{f_0} = \begin{cases} < y >^s_{f_0}, & if (< y >)^s_{f_0} > A^s_{f_0} \\ u, & (u: the - first - Tid) > A^s_{f_0}, u \notin d_s(< y >). \end{cases}$$

$$d_s(C) = \{u: u \in d_s(< y >), u > C^s_{f_0}\}$$

Example 5.1 *Given database sequence* $s =< ab, bd, abcd, e >$. *The diffseq of* $< a, c >$, $diffseq_s(< a, c >)$, *can be constructed using* $diffseq_s(< a >)$ *and* $diffseq_s(< c >)$ *by a sequence-extension step. Since* $(< c >)^s_{f_0} > (< a >)^s_{f_0}$, *then* $(< a, c >)^s_{f_0} =< c >^s_{f_0} = 3$.
$d_s(< a, c >) = \{u : u \in d_s(< c >), u > (< a, c >)^s_{f_0}\} = \{u : u \in \{4\}, u > 3\} = \{4\}$ .Then,
$diffseq_s(< a, c >) = \{(< a, c >)^s_{f_0}\} \cup d_s(< a, c >) = \{3,4\}$.

Method II: The following equations define the two terms:

$$C^s_{f_0} = \begin{cases} B^s_{f_0}, & if B^s_{f_0} > A^s_{f_0} \\ u, & (u: the - first - Tid) > A^s_{f_0}, u \notin d_s(B). \end{cases}$$
$$d_s(C) = \{u : u \in d_s(B), u > C^s_{f_0}\}$$

Example 5.2 *Given database sequence* $s =< ab, bd, abdcd, e >$. *The diffseq of* $< ab, c >$, $diffseq_s(< ab, c >)$, *can be constructed using* $diffseq_s(< ab >)$ *and* $diffseq_s(< a, c >)$ *by a sequence-extension step. Since* $(< a, c >)^s_{f_0} = 3 > 1 = (< ab >)^s_{f_0}$, *then* $(< ab, c >)^s_{f_0} = (< a, c >)^s_{f_0} = 3$.
$d_s(< ab, c >) = \{u : u \in d_s(< a, c >), u > (< ab, c >)^s_{f_0}\} = \{u : u \in \{4\}, u > 3\} = \{4\}$.
$diffseq_s(< ab, c >) = \{(< ab, c >)^s_{f_0}\} \cup d_s(< ab, c >) = \{3\} \cup \{4\} = \{3,4\}$.

5.2.2 The diffseqs of longer patterns that constructed using Itemset-Extension:
Suppose that the sequence $A =< px >$ is extended in an itemset-extension step to get the pattern $C =< pxy >$, where p stands for the prefix, $|p| \geq 0$, and x and y are database

items. There are two methods by which we can construct $diffseq_s(C)$. The first one is by joining $diffseq_s(A)$ with $diffseq_s(<y>)$, where y is the item used for extending A. The second method is by joining $diffseq_s(A)$ with $diffseq_s(B)$, where $B=<py>$ as in the equivalence class approach [10].

Method I: The following equations define the two terms:

$$C_{f_0} = \begin{cases} (<y>)_{f_0}, & if\,(<y>)_{f_0} = A_{f_0} \\ u & : |s| \geq u \geq max(A_{f_0}, <y>_{f_0}), u \notin (d_s(A) \cup d_s(<y>)) \end{cases}$$

$$d_s(C) = \{u : u \in (d_s(A) \cup d_s(<y>)), u > C_{f_0}\}$$

Example 5.3 *Given database sequence* $s = <ab, bd, abcd, e>$. *The diffseq of* $<ab>$, $diffseq_s(<ab>)$, *can be constructed using* $diffseq_s(<a>)$ *and* $diffseq_s(<b>)$ *by an itemset-extension step. Since* $(<a>)^s_{f_0} = (<b>)^s_{f_0}$, *then* $(<ab>)^s_{f_0} = (<a>)^s_{f_0} = 1$.

$d_s(<ab>) = \{u : u \in (d_s(<a>) \cup d_s(<b>)), u > (<ab>)^s_{f_0}\}$

$d_s(<ab>) = \{u : u \in (\{2,4\} \cup \{4\}), u > 1\} = \{2,4\}$.

$diffseq_s(<ab>) = \{(<ab>)^s_{f_0}\} \cup d_s(<ab>) = \{1\} \cup \{2,4\} = \{1,2,4\}$. By the same way $diffseq_s(<ac>) = \{3,4\}$.

Method II: The following equations define the two terms:

$$C_{f_0} = \begin{cases} A_{f_0}, & if\,A_{f_0} - B_{f_0} \\ u & : |s| \geq u \geq max(A_{f_0}, B_{f_0}), u \notin (d_s(A) \cup d_s(B)) \end{cases}$$

$$d_s(C) = \{u : u \in (d_s(A) \cup d_s(B)), u > C_{f_0}\}$$

Example 5.4 *Given database sequence* $s = <abc, bcd, abcd, e>$. *The diffseq of* $<abc>$, $diffseq_s(<abc>)$, *can be constructed using the* $diffseq_s(<ab>)$ *and* $diffseq_s(<ac>)$ *by itemset-extension step.*

Since $<ab>^s_{f_0} \neq <ac>^s_{f_0}$, *then* $(<abc>)^s_{f_0} = max(<ab>^s_{f_0}, <ac>^s_{f_0}) = 3 \notin (d_s(<ab>) \cup d_s(<ac>))$ *and* $3 \leq 4 = |s|$. *Also* $d_s(<abc>) = \{4\}$ *since 4 is the only element in the union (* $d_s(<ab>) \cup d_s(<ac>)$ *) greater than* $(<abc>)_{f_0}$. *Thus we have* $diffseq_s(<abc>) = \{3\} \cup \{4\} = \{3,4\}$.

Consider the database D that consists of more than one sequence. Define diffseq of the pattern P as:

$$diffseq_D(P) = \bigcup_{sid=1}^{|D|} \{\{-sid\} \cup diffseq_{sid}(P)\}$$

Where we use here negative numbers ($-sid$) instead of sid. The negative sign here works as separator between sequence blocks in $diffseq_D(P)$. That frees memory that used previously for maintaining sid with tid in elements of each sequence block (as in id-lists). In order to locate sequence blocks in $diffseq_D(P)$ to be intersected we have to check for equality of $-sid$ and intersect diffseqs that have the same $-sid$.

Example 5.5 *The database D in Table 1 is updated in vertical diffseq representation as follows in Table 4. Also let us consider the diffseq of items a and b represented in Table 4, the diffseq of patterns $< ab >$ (Equality-Join) and $< a,b >$ (Temporal-Join) is represented in Table 5*

Table 4. The Diffseqs of Database Items of D in Table 1

diffseq(a**)**	**diffseq(**b**)**	**diffseq(**c**)**	**diffseq(**d**)**	**diffseq(**e**)**
-10	-10	-10	-10	-10
1	1	1	3	4
---	4	3	---	---
---	---	4	---	---
-20	-20	-20	-20	---
1	1	1	3	---
---	3	---	---	---
---	4	---	---	---
-30	-30	-30	---	-30
1	1	2	---	1
---	---	3	---	---
-40	-40	-40	---	---
1	1	1	---	---

Table 5. diffseqs for $< ab >$ and $< a,b >$

diffseq($< ab >$**)**	**diffseq(**$< a,b >$**)**
-10	-10
1	2
4	4
-20	-20
1	2
3	3
-30	-30
1	2
-40	-40
1	2

The support of any sequence A is given by the number of different ($-sids$) in $diffseq(A)$. We have, $sup(< ab >) = sup(< a,b >) = 4$. Note that Tables 4 and 5 contain 58 entries in

total. Compare this number with the 102 entries if the id-list reprsentation is used (see Tables 2 and 3). This example shows that diffseq reprsentation is 2 times better in space than the id-list reprsentation. The less space of diffseq reprsentation will lead to faster joning also.

Theorem 5.1 *(correctness) For all patterns in a sequence database D, the support that determined by diffseqs is the same support that determined by id-lists.*

5.3 dSpade Algorithm

To illustrate the power of diffseqs-based mining, we have integrated diffseqs with the vertical mining algorithm Spade [10], which mines frequent sequences. Our enhancement is called dSpade. In dSpade frequent sequences are generated by computing diffseqs for all distinct pairs of sequences in a given equivalence class and checking the support of the resulting sequences. The dSpade algorithm is presented as follows:

dSpade: *Find Sequential Patterns using Counting Method Based on Diffseq.*
Input: Sequence Database D and min_sup.
Output: Frequent Subsequences in D.
Method: Figures $2-5$ in Section 8 (Appendix).

6 Experimental Evaluation

In this section, we present the results of our experiments on the performance of dSpade and Spade [10]. The source code of Spade is avariable (http://www.cs.rpi.edu/zaki/software/). All the experiments were performed on a 2.4GHz Intel Celern Pentium 4 PC machine with 512MB of RAM and running RetHat Linux 8.0 operating system. The algorithms were coded in C++. Furthermore, the times for all the vertical methods involved in the experiments include all costs, including the conversion of the original database from a horizontal to a vertical format required for the vertical algorithms. The peak memory usage was measured with the *memusage* program. The output of the algorithms was turned off to make the comparison fare. Also to make the time measurements more reliable, no other applications were running on the machine while doing the experiments.

All the experiments were performed on a sysnthetic dataset generated with the IBM AssocGen program [2].The synthetic datasets were widely used in the domains of frequent sequence and item mining [2, 10, 3]. Therefore they became suitable for algorithms comparison. The parameters used to generat the dataset are summarized in Table 6.

Table 6. Parameters of Sequence Data Sets

Symbol	Meaning	Value
D	Number of sequences in 000's	100k
C	Average number of transactions per sequences	10
T	Average number of items per Transaction	2.5
S	Average length of maximal frequent sequences	4
I	Average size of Itemsets in maximal frequent sequences	1.25
N	Number of items in 000's	10k

Figure 6 (in Section 8 (Appendix)) reports the total execution time obtained by running dSpade and Spade on one sparse dataset, C10T10S4I4N0.1kD1k, and on three dense datasets, C10T30S4I4N0.1kD1k, C10T50S4I4N0.1kD1k, and C10T60S4I4N0.1kD1k as a function of the support threshold. The figure shows that dSpade outperforms Spade on the three dense datasets by more than 2 factors. The reason of this behavior is that the diffset (the origin of diffseq) is proved to be suitable for mining dense data sets in previous research. Thus on dense datasets, the size of diffseqs is small compared with the size of id-lists and this will lead to faster joning. while Spade outperforms dSpade on sparse dataset, C10T10S4I4N0.1kD1k.

In terms of memory usage we compared the memory consumption between dSpade and Spade on the above four datasets as shown in Figure 7 in Section 8 (Appendix). This figure shows that dSapde is efficient in memory usage compared with Spade on both sparse and dense datasets by more than 3 factors. Since for dense datasets, the size of diffseqs is small compared with the size of id-lists and for sparse dataset, as we mentioned before, we use negative numbers ($-sid$) instead of sid. The negative sign works as separator between sequence blocks in $diffseq_D(P)$ for any for any sequence P. That frees memory that used previously for maintaining sid with tid in elements of each sequence block (as in id-lists).

7. CONCLUSION

In this paper we have presented an adaptation of the well-known diffset data representation [12] with Spade algorithm called diffseqs. To illustrate the power of diffseqs-based mining, we have integrated diffseqs with the vertical mining algorithm Spade [10], which mines frequent sequences. Our enhancement is called dSpade. Since diffset shows high performance for mining frequent itemsets in dense transactional databases, experimental evaluation shows that dSpade is suitable for mining dense sequence databases in terms of time and memory.

In real life applications, one needs to extract sequential patterns under specific time constraints like Time-windows, minimum and maximum gap between consecutive transactions of a sequence. Such constraints have been introuced in [11], but very little work has been done in this field [9, 6, 11, 5, 7]. We are currently investigating on incorporating time constraints into dSpade.

REFERENCES

[1] R. Agrawal and H. Mannila and R. Srikant and H. Toivonen and I. Verkamo, Fast Discovery of Association Rules. In U. M. Fayyad, G. P. Shapiro, P. Smyth and R. Uthurusamy, editors, Advances in Knowledge Discovery and Data Mining, AAAI/MIT press, pages 307-328, 1996

[2] R. Agrawal and R. Srikant, Mining Sequential Patterns. In Proc. of the ICDE Conference, pages 3-14, Washington, DC, USA, 1995. IEEE Computer Society.

[3] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. In Proc. of the 8th ACM SIGKDD, pages 429-435. ACM Press, 2002

[4] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In Proc. of the 6th ACM SIGKDD, pages 355-359. ACM Press, 2000.

[5] M.-Y. Lin, S.-Y. Lee, and S.-S. Wang. Delisp: Efficient discovery of generalized sequential patterns by delimited pattern-growth technology. In Proc. of the 6th PAKDD, pages 198-209. Springer-Verlag, 2002.

[6] F. Masseglia, F. Cathala, and P. Poncelet. The PSP approach for mining sequential patterns. In Proc. of the 2nd PKDD, pages 176-184. Springer Verlag, 1998.

[7] S. Orlando, R. Perego, and C. Silvestri. A new algorithm for gap constrained sequence mining. In Proc. of the 2004 ACM SAC, pages 540-547. ACM Press, 2004.

[8] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proc. of the 17th ICDE, pages 215-226. IEEE Computer Society, 2001.
[9] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In Proc. of the 5th ICEDT, pages 3-17, London, UK, 1996. Springer- Verlag.
[10] M. J. Zaki. Efficient enumeration of frequent sequences. In Proc. of the 7th ICIKM, pages 68-75. ACM Press, 1998.
[11] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In Proc. of the 9th CIKM, pages 422-429. ACM Press, 2000.
[12] M. J. Zaki and and K. A. Gouda. Fast Vertical Mining Using Diffsets. In Proc. of the 9th KDD, pages 422-429. ACM Press, 2003.

8 APPENDIX

In this section, we provide pseudo code of the dSpade algorithm, see Figures (2-5) and Comparison between dSpade and Spade (Time at Figure 6 and Memory Usage at Figure 7).

dSpade(D, min_sup)
1. F_1 = { frequent items or 1-sequences }
2. F_2 = { frequent 2-sequences }
3. ξ = { equivalence classes $[X]_{\theta_1}$ }
4. **for** all $[X] \in \xi$ **do Enumerate-Frequent-Seq**($[X]$)

Figure 2: dSpade Algorithm

Enumerate-Frequent-Seq(S)
1. **for** all atoms $A_i \in S$ **do**
2. $T_i = \varnothing$
3. **for all atoms** $A_j \in S$ **, with** $j > i$ **do**
4. $R = Merge(A_i, A_j)$
5. **if** (**Prune**(R) == FALSE) **then**
6. **if** (itemset-extension) **then**
7. $diffseq(R)$ =**Equality-Join** $(diffseq(A_i), diffseq(A_j))$
8. **if** (sequence-extension) **then**
9. $diffseq(R)$ =**Temporal-Join** $(diffseq(A_i), diffseq(A_j))$
10. **if** $\sigma(R) \geq min_sup$ **then**
11. $T_i = T_i \cup \{R\}$; $F_{|R|} = F_{|R|} \cup \{R\}$
12. **if** DFS **then Enumerate-Frequent-Seq**(T_i)
13. **if** BFS **then for** all $T_i \neq \phi$ **Enumerate-Frequent-Seq**(T_i)

Figure 3: Enumerate Frequent Sequences Function

Equality-Join $(diffseq(X), diffseq(Y))$
1. $diffseq(R) = \varnothing$; $sup = 0$
2. **for** each sequence s that supports X and Y **do**
3. $R^s_{f_0} = 0$
4. **if** $X^s_{f_0} = Y^s_{f_0}$ **then** $R^s_{f_0} = X^s_{f_0}$
5. **else if** there exist integer k , $|s| \geq k \geq max(X^s_{f_0}, Y^s_{f_0})$,
6. and $k \notin (d_s(X) \cup d_s(Y))$ **then** $R^s_{f_0} = k$
7. **if** $R^s_{f_0} > 0$ **then**
8. $d_s(R) = \varnothing$
9. **for** each $m \in (d_s(X) \cup d_s(Y))$, $m > R^s_{f_0}$ **do**
10. $d_s(R) = d_s(R) \cup \{m\}$
11. $diffseq_s(R) = \{-sid\} \cup \{R^s_{f_0}\} \cup d_s(R)$
12. $diffseq(R) = diffseq(R) \cup diffseq_s(R)$
13. $sup{+}{+}$
14. **return** $diffseq(R)$, sup

Figure 4: Equality Join Function

Temporal-Join $(diffseq(X), diffseq(Y))$
1. $diffseq(R) = \varnothing$; $sup = 0$
2. **for** each sequence s that supports X and Y **do**
3. $R^s_{f_0} = 0$
4. **if** $X^s_{f_0} < Y^s_{f_0}$ **then** $R^s_{f_0} = Y^s_{f_0}$
5. **else if** there exist integer k , $k > X^s_{f_0}$ and $k \notin d_s(Y)$ **then** $R^s_{f_0} = k$
6. **if** $R^s_{f_0} > 0$ **then**
7. $d_s(R) = \varnothing$
8. **for** each $m \in d_s(Y)$, $m > R^s_{f_0}$ **do**
9. $d_s(R) = d_s(R) \cup \{m\}$
10. $diffseq_s(R) = \{-sid\} \cup \{R^s_{f_0}\} \cup d_s(R)$
11. $diffseq(R) = diffseq(R) \cup diffseq_s(R)$
12. $sup{+}{+}$
13. **return** $diffseq(R)$, sup

Figure 5: Temporal Join Function

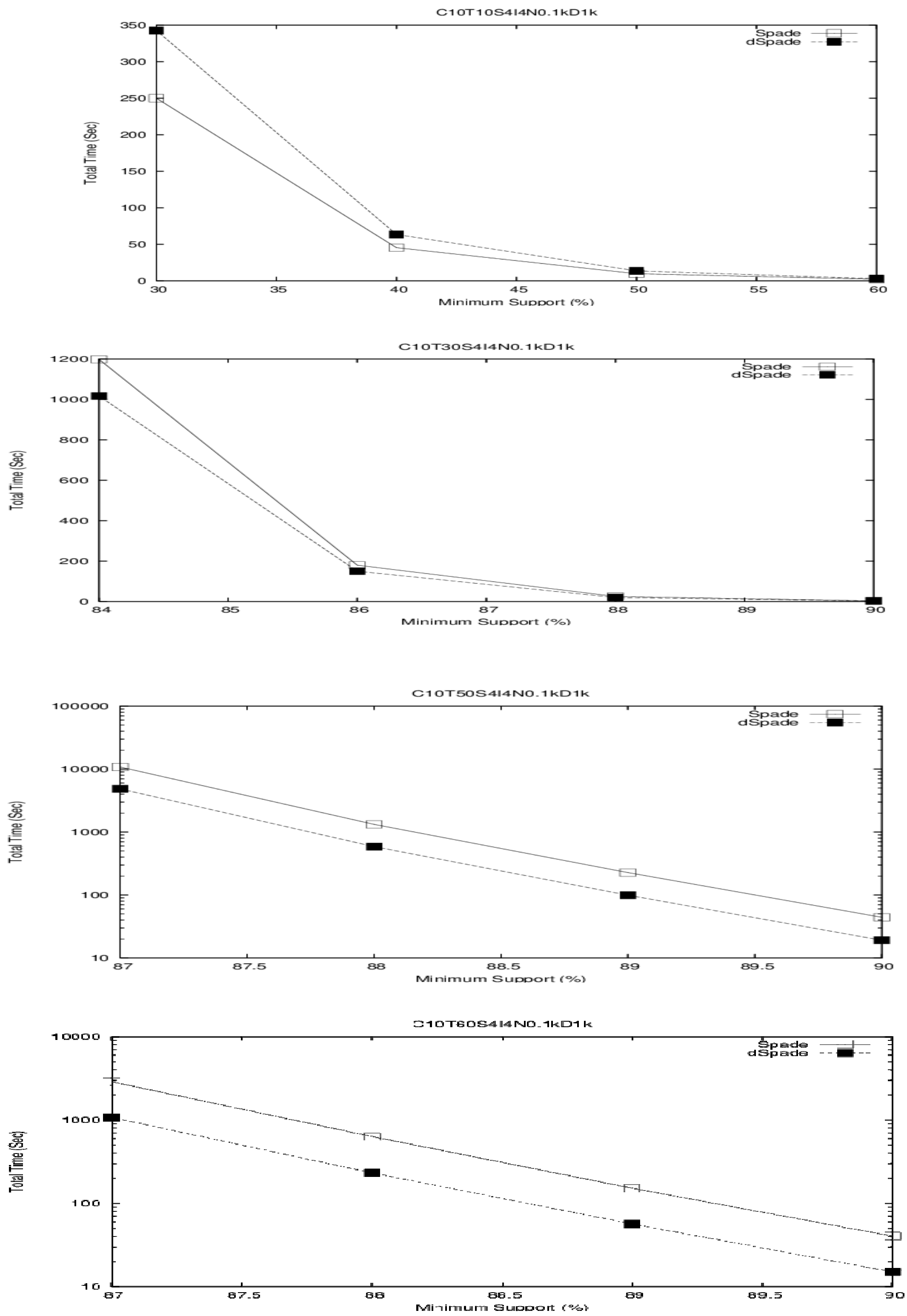

Figure 6: Comparison between dSpade and Spade (Time)

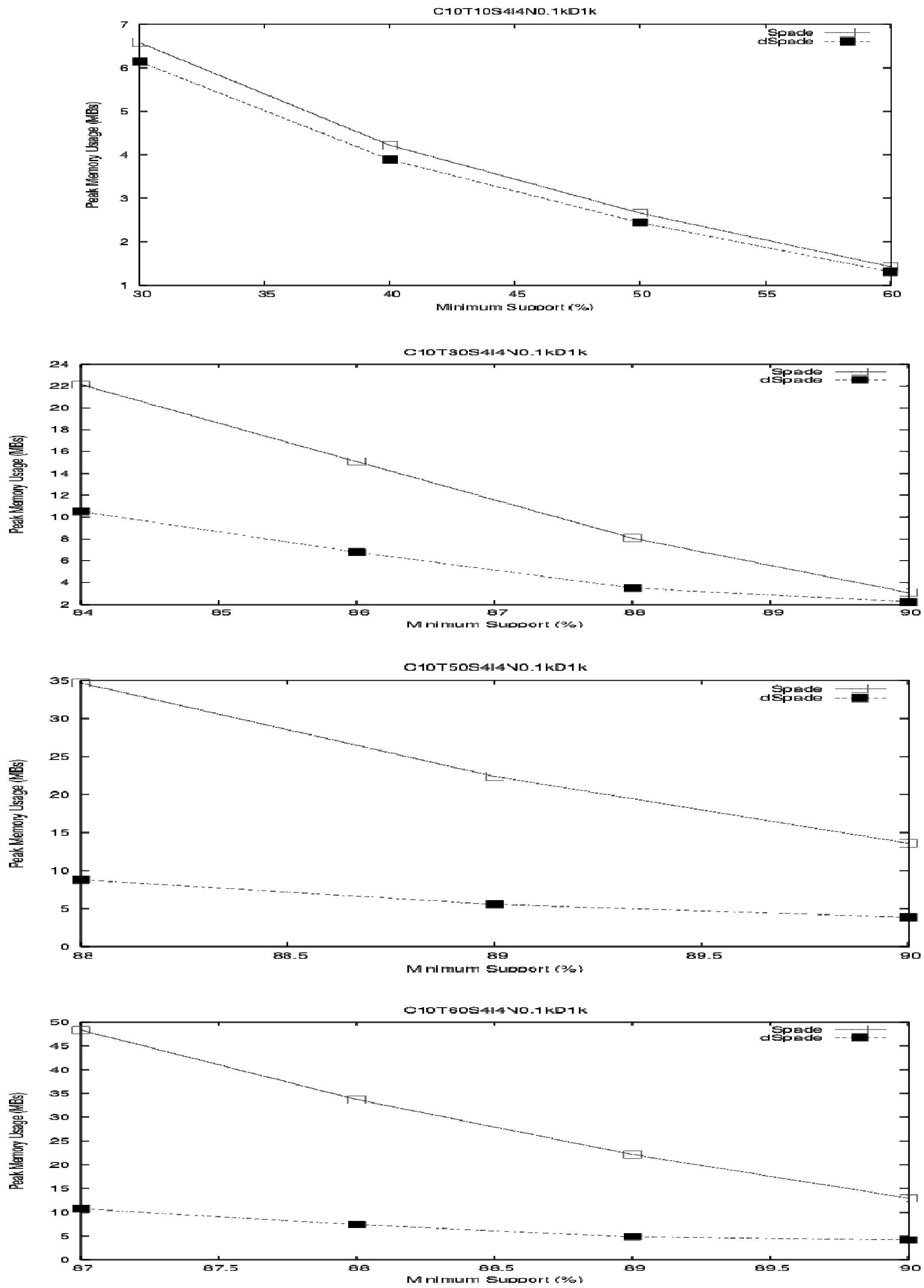

Figure 7: Comparison between dSpade and Spade (Memory Usage)