Under consideration for publication in Knowledge and Information
Systems

# A Binary Decision Diagram Based Approach for Mining Frequent Subsequences

Elsa Loekito[1], James Bailey[1] and Jian Pei[2]

eloekito@csse.unimelb.edu.au, jbailey@csse.unimelb.edu.au, jpei@cs.sfu.ca
[1]National ICT Australia (NICTA), Department of Computer
Science and Software Engineering, University of Melbourne, Australia;
[2]School of Computing Science, Simon Fraser University, Canada

**Abstract.** Sequential pattern mining is an important problem in data mining. State of the art techniques for mining sequential patterns, such as frequent subsequences, are often based on the pattern-growth approach, which recursively projects conditional databases. Explicitly creating database projections is thought to be a major computational bottleneck, but we will show in this paper that it can be beneficial when the appropriate data structure is used. Our technique uses a canonical directed acyclic graph as the sequence database representation, which can be represented as a Binary Decision Diagram (BDD). In this paper, we introduce a new type of BDD, namely a Sequence BDD (SeqBDD), and show how it can be used for efficiently mining frequent subsequences. A novel feature of the SeqBDD is its ability to share results between similar intermediate computations and avoid redundant computation. We perform an experimental study to compare the SeqBDD technique with existing pattern growth techniques, that are based on other data structures such as prefix trees. Our results show that a SeqBDD can be half as large as a prefix tree, especially when many similar sequences exist. In terms of mining time, it can be substantially more efficient when the support is low, the number of patterns is large, or the input sequences are long and highly similar.

**Keywords:** Sequential pattern mining, Frequent subsequences, Binary Decision Diagram, Sequence Binary Decision Diagram, SeqBDD.

*Received Apr 03, 2008*
*Revised Apr 05, 2009*
*Accepted Aug 15, 2009*

# 1. Introduction

Mining sequential patterns is an important task in data mining. There are many useful applications, including analyzing time-stamped market basket data, finding web access patterns (WAP) from web logs (Pei, Han, Mortazavi-asl and Zhu, 2000), finding relevant genes from DNA sequences (Ma, Wang, Sasha and Wu, 2001) and classifying protein sequences (Ferreira and Azevedo, 2005; She, Chen, Wang, Ester, Gardy and Brinkman, 2003). The frequent subsequence is a fundamental type of sequential pattern. Given a minimum frequency threshold, it is defined as a frequently occurring ordered list of events or items, where an item might be a web page in a web access sequence, or a nucleotide in a DNA sequence.

Frequent subsequence miners must explore an exponentially sized search space (Pei, Han, Mortazavi-Asl, Wang, Pinto, Chen, Dayal and Hsu, 2004), which makes mining particularly challenging when a large number of long frequent subsequences exist, such as in DNA or protein sequence data sets which have a small alphabet, or in weblog data sets, which have a larger alphabet size, at a low minimum support (Pei et al., 2000). Popular techniques for mining frequent subsequences such as those in (Pei et al., 2004; Ezeife and Lu, 2005) adopt a *pattern growth* approach which is thought to be the most efficient category of approaches for sequential pattern mining. The generation of infrequent candidate patterns is avoided by projecting the database into smaller conditional databases recursively. However, there are some challenges which need to be addressed in such a technique: 1) Projecting numerous conditional databases takes considerable time as well as space, especially since item reordering optimizations do not make sense for sequential mining, like they do for frequent itemset mining. 2) Each conditional database is processed independently, although they often contain some common sub-structures.

One of the most popular data structures used in frequent pattern mining is the prefix tree. Its extensions for the sequence context have been proposed in (Ezeife and Lu, 2005). However, the benefit of using a prefix tree for mining frequent subsequences is questioned in (Pei et al., 2004), since its compactness relies on common prefix-paths, which cannot be fully exploited in the sequence context. Moreover, work in (Pei et al., 2004) proposed the PrefixSpan algorithm, which grows prefixes of the frequent patterns, also often called the prefix-growth approach. PrefixSpan is one of the best pattern growth algorithms, and its optimization technique completely avoids physically constructing database projections. When the alphabet size is small and the volume of patterns is huge, however, PrefixSpan is limited in not being able to exploit the similarity between the sequences or the candidates.

Instead of a prefix tree, we will demonstrate how a directed acyclic graph (DAG), represented as a Binary Decision Diagram(BDD) (Bryant, 1986), can be used as an alternate data structure for mining frequent subsequences. A BDD is a canonical representation of a boolean formula and it has been successfully used in other fields such as boolean satisfiability solvers (Aloul, Mneimneh and Sakallah, 2002), and fault tree analysis (Sinnamon and Andrews, 1996). BDDs potentially allow greater data compression than prefix trees, since node fan in (suffix sharing) as well as node fan out (prefix sharing) is allowed in a BDD, and identical sub-trees do not exist. Complementing their compactness, BDDs are also engineered to enforce reuse of intermediate computation results, which makes them potentially very useful for pattern growth sequence mining, in particular for compressing and efficiently manipulating the numerous conditional databases.

In this paper, we introduce a mining technique based on the use of a special kind of BDD, which can provide an overall compressed representation of the intermediate databases, and in contrast to previous thinking, benefit from their explicit construction by relying on sophisticated techniques for node sharing and caching. It is not straightforward to efficiently encode sequences into BDDs, however. To this end, we introduce an original variant, namely the **Sequence Binary Decision Diagram (SeqBDD)**, which is suitable for compactly representing sequences and efficiently mining frequent subsequences. Based on the canonical property of SeqBDDs, multiple databases may share sub-structures, which in turn allows high overall data compression to be achieved and redundant computations to be avoided. The questions which we address in this research are:

- *Can sequences be compactly represented using a BDD?*
- *Can the use of a BDD benefit frequent subsequence mining?*
- *Can a BDD-based frequent subsequence mining technique outperform state-of-the-art pattern growth techniques?*

To summarise, our contributions in this paper are three-fold:

- We introduce a compact and canonical DAG data structure for representing sequences, which we call a **Sequence Binary Decision Diagram (SeqBDD)**, and its weighted variant which allows the frequency (or support) of the sequences to be represented. Unlike prefix-trees, SeqBDDs allow node fan-out as well as fan-in, which is a novel data representation in the sequence mining literature.
- We show how a Weighted SeqBDD can be used for mining frequent subsequences in the prefix-growth framework. In particular, node-sharing across multiple databases is allowed, and BDD primitives which promote re-use of intermediate computation results are adopted in the mining procedure, allowing high data compression and efficient mining to be achieved. The ability to exploit any similarity between the conditional databases is a novel feature of our mining technique.
- We experimentally investigate the behavior of our technique for finding frequent subsequences in biological data sets which have a small alphabet domain, as well as weblog data sets with a large domain which are challenging when the support threshold is low, and several synthetic data sets. We compare its performance against the competitive pattern growth algorithms, PLWAP (Ezeife and Lu, 2005) and PrefixSpan (Pei et al., 2004). SeqBDDMiner is proven to be superior when the input sequences are highly similar, or when the minimum support threshold is low, where the conditional databases share many common sub-structures.

## 2. Preliminaries

This section gives the definition of some terminologies used in frequent subsequence mining.

Let $I$ be the set of distinct items. For instance, in a DNA data set, the set

$I$ contains alphabet letters $A$, $C$, $G$, $T$. A *sequence* $S$ over set $I$ is an ordered list of items, $e_1e_2\ldots e_m$ where $e_j \in I$, and each item $e_j$ is called an *element* of $S$, for $1 \leq j \leq m$. The $j$-th element that occurs in $S$ is denoted by $S[j]$. The number of elements in a sequence, $m$, is referred as the *length* of $S$, and it is denoted by $|S|$. Each number between 1 and $m$ is a *position* in $S$. Set $I$ is called as the *alphabet domain* of $S$. An item from the alphabet domain can occur multiple times as different elements of a sequence. A data set $D$ is a collection of sequences, defined upon a domain set of items, $I$. The number of sequences in $D$ is denoted by $|D|$.

A sequence $p = a_1a_2\ldots a_m$ is a *supersequence* of another sequence $q = b_1b_2\ldots b_n$ $(n \leq m)$, and $q$ is a *subsequence* of $p$, if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $q[1] = p[i_1]$, $q[2] = p[i_2]$, $\ldots$, $q[n] = p[i_n], and|q| \leq |p|$. We say that $p$ *contains* the subsequence $q$. The frequency of a sequence $p$ in $D$ is the number of sequences in $D$ which contain $p$. The *support* of $p$ is defined as the relative frequency of $p$ with respect to the number of sequences in $D$, i.e. $\frac{frequency(p)}{|D|}$.

Sequence $q$ is a *prefix* of $p$ if $q$ is a subsequence of $p$, and $q[i]$ is equal to $p[i]$ for all $1 \leq i \leq |q|$. Sequence $q$ is a *suffix* of $p$ if $q$ is a subsequence of $p$, and $q[i]$ is equal to $p[j]$ for all $1 \leq i \leq |q|$, where $j = (|p| - |q| + i)$. An *x-suffix* of a sequence $p$ is a suffix of $p$ which begins with item $x$. Moreover, it is a *longest x-suffix* in $p$ if it begins with the first occurrence of $x$ in sequence $p$.

Given a positive minimum support threshold *min_support*, $p$ is a *frequent subsequence* if the support of $p$ is not less than *min_support*, i.e. *support*$(p) \geq$ *min_support*, where $0 \leq$ *min_support* $\leq 1$. The task of mining frequent subsequences from a data set $D$ is defined as finding all subsequences from $D$ whose supports are at least *min_support*. Moreover, *closed frequent subsequences* (Wang and Han, 2004) are frequent subsequences such that none of their supersequences have the same frequency. In this paper, we focus only on finding frequent subsequences, which has a larger search space than the closed subsequences.

Frequent subsequences have an anti-monotonic property (Srikant and Agrawal, 1996). For every sequence $p$ and its subsequence $q$, *support*$(p) \leq$ *support*$(q)$. This anti-monotonic property is often known as the *apriori* property of frequent subsequences. Moreover, the prefix anti-monotonic property (Pei, Han and Want, 2002) also holds, since *support*$(p) \leq$ *support*$(q)$ is true for a sequence $p$ and its prefix $q$.

## 3. Overview of sequential pattern growth techniques

In this section, we will give an overview of the general pattern growth framework in frequent subsequence mining, which is adopted by our technique and in the state-of-the-art techniques such as Prefixspan (Pei et al., 2004), WAP-mine (Pei et al., 2000), and PLWAP (Ezeife and Lu, 2005). The pattern growth framework was introduced in frequent itemset mining (Han, Pei, Yin and Mao, 2004), which recursively divides the database into conditional databases, each of which is projected by a frequent item. For mining frequent subsequences, the pattern growth framework is applied in a similar manner. Other techniques for mining frequent subsequences exist, but they are not based on the pattern-growth framework. We will present a review of those techniques later in the paper.

*Prefixspan* (Pei et al., 2004) adopts the pattern growth framework for mining frequent subsequences, by growing prefixes of the frequent subsequences. For a prefix $x$, its conditional database contains the longest $x$-suffix from each sequence. Optimization techniques used by Prefixspan include pseudo-projection and bi-level database projection. The pseudo-projection technique uses *pointer-offset* pairs for finding the conditional databases, instead of physically creating them. However, it is only possible when the database fits in the main memory. The bi-level projection allows fewer and smaller databases being projected, by inducing a conditional database for each length-2 prefix.

*WAP-mine* uses a WAP-tree (Web Access Pattern tree) (Pei et al., 2000) database representation, which is similar to the FP-tree in (Han et al., 2004) for mining frequent itemsets. Each node in a WAP tree is labeled, and nodes with the same label are linked together. Each branch represents a sequence. Unlike the other algorithms, WAP-mine grows suffixes of frequent subsequences. The use of a prefix tree is aimed to achieve data compression, and quick identification of frequent prefixes. However, constructing the numerous databases in WAP-mine can be costly. PLWAP (Ezeife, Lu and Liu, 2005) is an improvement over WAP-mine, but it adopts a prefix growth framework and does not build new conditional databases. PLWAP uses a PLWAP-tree (Pre-order Linked Web Access Pattern tree) as the data representation, with a special code annotating each node which allows the initial database to be re-used for representing the conditional databases. However, such a technique is computationally expensive when long subsequences exist.

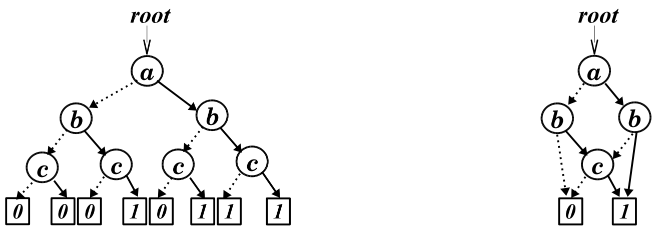
## 4. Overview of Binary Decision Diagrams

In this section we will give some background about Binary Decision Diagrams (BDDs), the data structure we will use, which are a compact and canonical graph representation of boolean formulae. We will also give an overview of a special type of BDD, namely a Zero-suppressed BDD (ZBDD) (Minato, 1993), which has been introduced for efficient representation of monotonic boolean functions and has also been used for mining frequent and emerging patterns (Loekito and Bailey, 2006; Minato and Arimura, 2006; Loekito and Bailey, 2007). A survey on other ZBDD applications can be found in (Minato, 2001). We will describe how BDDs can be adopted for efficiently mining frequent subsequences in the following section.

### 4.1. Binary Decision Diagrams

Binary Decision Diagrams(BDDs) (Bryant, 1986), are a canonical directed acyclic graph (DAG) data structure, which were introduced for compactly representing boolean formulae. A BDD can also be viewed as a binary decision tree with the identical sub-trees being merged.

For example, let $F$ be a boolean formula such that $F = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$. Figure 1(a) shows the binary decision tree for $F$, in which A solid line represents a true assignment, and a dotted line represents a false assignment to a variable. Each node at the lowest level represents the output of the function, where 1 means TRUE, 0 means FALSE. An example of a BDD representation for $F$ is



(a) Binary Decision Tree (b) Binary Decision Diagram

**Fig. 1.** Example of Binary Decision Tree and Binary Decision Diagram for boolean formula $F = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$ (a solid line represents a TRUE value of the variable; a dotted line represents a FALSE value of the variable)

shown in Figure 1(b), in which identical sub-trees do not exist, and node fan-in is allowed.

Formally, a BDD is a canonical DAG, consisting of one source node, multiple internal nodes, and two sink nodes which are labeled as 0 and 1. Each internal node may have multiple parent nodes, but it has only two child nodes. A node $N$ with label $x$, denoted $N = node(x, N_1, N_0)$, encodes the boolean formula $N = (x \wedge N_1) \vee (\overline{x} \wedge N_0)$. The edge connecting node $N$ to $N_1$ (resp. $N_0$ ) is also called the *true*-edge (resp. *false*-edge) [1]. Each path from the root to sink-1 (resp. sink-0) gives a true (resp. false) output of the represented function. A BDD is called *ordered* if the label of each internal node has a lower index than the label of its child-nodes. Being canonical, each BDD node is unique, which can be obtained by merging any two identical nodes.

BDDs allow logical operations on Boolean formulae, such as AND, OR, and XOR, to be performed in polynomial time with respect to the number of nodes. Their efficiency is based on the following two important properties:

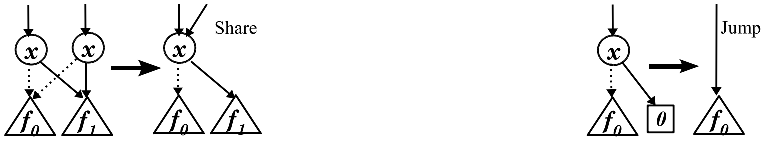
- **Canonical**: Equivalent subtrees are shared and redundant nodes are eliminated.
- **Caching principle:** Computation results are stored for future reuse.

**Canonical property of BDDs:** The canonical property of BDDs (i.e. the uniqueness of each node) is maintained by storing the unique nodes in a table of <*key*,*node*>-pairs, called the *uniquetable*. The *key* is a function of the node's label and the *keys* of its child nodes.

**Caching principle:** Each BDD primitive operation is associated with a cache, which is also called as the *computation table*, that maps the input parameters to the output of the operation. The cached output may be re-used if the same intermediate operation is re-visited, avoiding redundant computations.

Existing BDD packages allow fast uniquetable access. In particular, the package that we use, JINC (Ossowski and Baier, 2006), implements both the unique table and computation table using hash tables. The unique table, moreover, uses a separate chaining, i.e. multiple keys with the same hash value are stored in a linked list. In principle, the size of each list is $n/m$, where $n$ is the number of nodes, and $m$ is the size of the table. When $n \leq m$, insertion and lookup operation can be done in a constant time. Otherwise, it requires the list to be

[1] In their illustrations, the true-edges are shown as solid lines, the false edges are shown as dotted lines



(a) Merging rule

(b) Zero-suppression rule

**Fig. 2.** ZBDD Reduction Rules

traversed, which may have $O(n/m)$ complexity (based on a linear search). On the other hand, the computation table keeps only one key (instead of a list) per entry, which allows a constant lookup or insertion time complexity.

## 4.2. Zero-suppressed Binary Decision Diagrams

A Zero-suppressed Binary Decision Diagram (ZBDD) (Minato, 1993) is a special kind of BDD which is particularly efficient for manipulating sparse combinations. More specifically, a ZBDD is a BDD with two reduction rules (see Figure 2 for illustrations):

- **Merging rule**: If nodes $u$ and $v$ are identical, then eliminate one of the two nodes, and redirect all incoming edges of the deleted node to the remaining one.
- **Zero-suppression rule**: If the true-edge of a node $u$ points to sink-0, then $u$ is deleted and all incoming edges to $u$ are redirected to $u$'s 0-child.

By utilising these rules, a sparse collection of combinations can be represented with high compression. For an $n$ variable formula, the possible space of truth values is $2^n$, but the BDD/ZBDD can have exponentially fewer nodes. We will consider the use of ZBDDs for representing sequences, but first we consider the itemset case. An itemset $p$ is seen as a conjunction of the items, and the set of itemsets is seen as a disjunction of such conjunctions. A ZBDD node $N = (x, N_1, N_0)$ represents a set of itemsets $\mathbf{S}$ such that $\mathbf{S} = \mathbf{S_0} \cup (\mathbf{S_1} \times \{x\})$ where $N_1$ and $N_0$ represent $\mathbf{S_1}$ and $\mathbf{S_0}$, respectively. A sink-0 encodes the empty set ($\emptyset$), and sink-1 encodes the set of empty itemsets ($\{\emptyset\}$). For clarity, we omit sink-0 nodes from the illustrations in this paper. The basic, pre-defined, ZBDD primitives(Minato and Arimura, 2005) which will be used in our algorithm are listed in Table 1. They have polynomial time complexity with respect to the number of nodes.

There are a few works which study the use of ZBDDs for pattern mining, such as for mining frequent itemsets (Loekito and Bailey, 2007; Minato and Arimura, 2006), mining emerging patterns (Loekito and Bailey, 2006), and finding simple disjunctions (Minato, 2005). Work in (Loekito and Bailey, 2007), in particular, proposed a weighted ZBDD which allows an efficient frequent itemset mining technique to be developed.

## 4.3. ZBDD-based representation for sets of sequences

Directly representing sets of sequences as ZBDDs is not possible, since ZBDDs do not allow a variable to appear multiple times in any path. A natural way to encode a sequence is by introducing variables which encode each item and its

| ZBDD | Itemsets |
|---|---|
| 0 | The empty set, $\emptyset$ |
| 1 | The set of an empty itemset, $\{\emptyset\}$ |
| $add(P, Q)$ | Set-union of $P$ and $Q$, i.e. the set of itemsets which occur in either $P$ or $Q$ |
| $subtract(P, Q)$ | Set-subtraction of $Q$ from $P$, i.e. the set of itemsets which occur in $P$ but do not occur in $Q$ |

**Table 1.** ZBDD Primitives ($P$ and $Q$ are ZBDDs)

| Sequence | Itemset Encoded |
|---|---|
| $p_1 = aabac$ | $a_1, a_2, b_3, a_4, c_5$ |
| $p_2 = baba$ | $b_1, a_2, b_3, a_4$ |
| $p_3 = aaca$ | $a_1, a_2, c_3, a_4$ |
| $p_4 = bbac$ | $b_1, b_2, a_3, c_4$ |

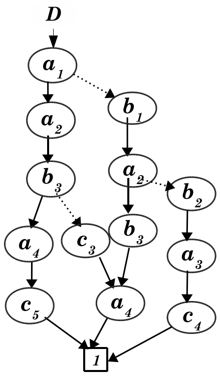


**Fig. 3.** Sequence to itemset translation using Naive encoding, and the resulting ZBDD representation of the sequence database

position in the sequence. In the following discussion, to differentiate the original sequential representation from its itemset representation, we refer to items in the original domain as alphabet letters.

Let $L$ be the maximum length of a sequence $S$, and $N$ be the size of the alphabet domain $A$. As a naive encoding, $S$ can be expressed as an itemset over a new domain $I'$ containing $L \times N$ items, each item represents an alphabet letter at a particular position in the sequence. Another encoding requires $L \times log_2 N$ items (Kurai, Minato and Zeugmann, 2007), which may be more efficient if the sequences contain many elements. We will give more details about each encoding shortly. We refer to the ZBDD representations based on the alternative encodings as ZBDD$_{naive}$, and ZBDD$_{binary}$, respectively.

### *4.3.1. Naive Encoding Scheme*

Let $x_i$ be an item in $I'$, where $i \in [1 \dots L]$, and $x \in A$. Item $x_i$ in the itemset representation of a sequence $S$ represents the occurrence of letter $x$ at the $i$'th position in $S$.

**Example 4.1.** Suppose we have an alphabet domain $A = \{a, b, c\}$, and the maximum length of the sequences is 5. The itemset domain contains 15 items. Let $D$ be a sequence database containing $p_1 = aabac$, $p_2 = baba$, $p_3 = aaca$, $p_4 = bbac$. The itemset representation of $p_1$ is $\{a_1, a_2, b_3, a_4, c_5\}$. Figure 3 shows the itemset encoded database.

| Symbol | Binary Code $(v_1, v_0)$ | Itemset Encoded |
|---|---|---|
| $a$ | $(0,1)$ | $x_{j.0}$ |
| $b$ | $(1,0)$ | $x_{j.1}$ |
| $c$ | $(1,1)$ | $x_{j.1}x_{j.0}$ |

| Sequence | Itemset Encoded |
|---|---|
| $p_1 = aabac$ | $x_{1.0}, x_{2.0}, x_{3.1}, x_{4.0}, x_{5.1}, x_{5.0}$ |
| $p_2 = baba$ | $x_{1.1}, x_{2.0}, x_{3.1}, x_{4.0}$ |
| $p_3 = aaca$ | $x_{1.0}, x_{2.0}, x_{3.1}, x_{3.0}, x_{4.0}$ |
| $p_4 = bbac$ | $x_{1.1}, x_{2.1}, x_{3.0}, x_{4.1}, x_{4.0}$ |

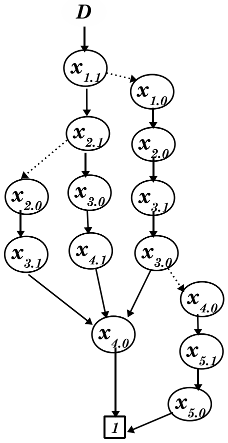


**Fig. 4.** Sequence to itemset translation using binary coding, and the resulting ZBDD representation of the sequence database

### *4.3.2. Binary Encoding Scheme*

The binary encoding scheme represents each alphabet letter by $n = \lceil log_2(N+1) \rceil$ binary variables. Suppose there are 3 letters, $a$, $b$, and $c$, they are represented using 2-bit binary variables $v_1, v_0$, such that $(v_1, v_0) = (0, 1)$ represents $a$, $(1, 0)$ represents $b$, and $(1, 1)$ represents $c$. At a given position $i \in [1 \dots L]$ in the sequence, and $j \in [0, \dots, n]$, item $x_{i.j}$ encodes the binary coding of a sequence element $(v_n, \dots, v_0)$ such that $v_j = 1$. Figure 4 shows the encoded database $D$ and its ZBDD representation.

**Compactness:** The work in (Kurai et al., 2007) uses a lexicographic variable ordering for the ZBDD. Other variable orderings may be used, but it does not have much influence on the compactness of the ZBDD, due to the position-specific item representation. The binary encoding allows more flexible node sharing between different alphabet representations, at the cost of using more nodes in each path than the naive encoding. In the context of sequential patterns, the ZBDDs are used to store frequent subsequences, which may occur in different positions in various sequences in the database. Using the position-specific itemset encoding, there is no quick way to identify the occurrence of a subsequence in several sequences. Based on this limitation, we next propose a new type of BDD which is more suitable, and more compact, for representing sequences.

## 5. Sequence Binary Decision Diagrams

We have shown in the previous section that Zero-suppressed Binary Decision Diagrams have limitations for representing sequences. In this section we will describe our proposed data structure, namely the **Sequence Binary Decision Diagram** or SeqBDD for short, and its weighted variant which allows more efficient manipulation of sequences.

In SeqBDDs, only the 0-child nodes are ordered with respect to their parent nodes, and a variable is allowed to occur multiple times in a path, which is not possible using any of the existing BDD variants. Analogous to ZBDDs, both the node-merging and the zero-suppression rules are employed in SeqBDDs, which allow a compact and canonical graph representation of sequences.

In order to use SeqBDDs for mining sequential patterns such as frequent subsequences, the frequency of the sequences need to be represented in the database. Inspired by the weighted-variant of ZBDD for frequent itemset mining (Loekito and Bailey, 2007), we introduce **Weighted Sequence Binary Decision Diagrams (Weighted SeqBDDs)**.

In SeqBDD semantics, a path in a SeqBDD represents a sequence, in which the nodes are arranged in-order to the positions of their respective variables in the sequence. More specifically, the top node corresponds to the head of the sequence, and the successive 1-child nodes correspond to the following elements, respectively. A SeqBDD node $N = node(x, N_1, N_0)$ denotes an internal node labeled by variable $x$, and $N_1$ (resp. $N_0$) denotes its 1-child (resp. 0-child). The label of node $N$ has a lower index (appear earlier in the variable ordering) than the label of $N_0$. We denote the total number of descendant nodes of node $N$, including $N$ itself, by $|N|$.

Let $x$ be an item and $P$ and $Q$ be two sets of sequences. We define an operation $x \times P$ which appends $x$ to the head of every sequence in $P$, and a set-union operation $P \bigcup Q$ which returns the set of sequences which occur in either $P$ or $Q$.

**Definition 5.1.** A SeqBDD node $N = node(x, N_1, N_0)$ represents a set of sequences $S$ such that $S = S_{\overline{x}} \bigcup (x \times S_x)$, where $S_x$ is the set of sequences which begin with element $x$ (with the head elements being removed), and $S_{\overline{x}}$ is the set of sequences which do not begin with $x$. Node $N_1$ represents set $S_x$, and node $N_0$ represents set $S_{\overline{x}}$.

Moreover, every sequence element in the SeqBDD can be encoded into a set of binary variables, using a similar binary encoding scheme used in ZBDD$_{binary}$ which was discussed in Section 4.3.2. However, each path is much likely longer, and the improvement in terms of node sharing is relatively small. Thus, in most cases, the database representation with the binary encoding is less compact. In the following discussion, we will compare the compactness of SeqBDDs with ZBDDs under the naive itemset encoding.

**Compactness of a SeqBDD**: By removing the variable ordering between each node and its 1-child, the SeqBDD's merging rule allows common suffix-paths between sequences to share nodes, regardless of their length. This is something which is not possible in the ZBDD representations. We employ a lexicographic variable ordering for the SeqBDD. Other variable orderings may be used, however, this would not have much influence on the compactness, since it is only employed partially.

The following theorems state the compactness of SeqBDDs relative to ZBDDs and prefix trees, where the size is proportional to the number of nodes.

**Theorem 5.1.** Given a sequence database, the sizes of the SeqBDD, ZBDD, and prefix tree representations satisfy the following relation: *SeqBDD* $\leq$ *ZBDD* $\leq$ *Prefix tree*, where the sequences in the ZBDD are encoded by the naive itemset encoding, and SeqBDD is the most compact.

*Proof.* The proof for this theorem follows from Lemma 5.1. □

**Lemma 5.1.** Given a sequence database, nodes in the ZBDD have a many-to-one relationship with the SeqBDD, and nodes in the prefix tree have a many-to-one relationship with the ZBDD.

*Proof.* Since the merging rule criteria affects either the ZBDD or the SeqBDD only if there is some common suffix between the sequences, assume there are two non-identical sequences in the input database, $p$ and $q$, which share a common suffix namely *suffix*. Let $P_Z$ and $Q_Z$ be two suffix paths in the ZBDD which correspond to *suffix* in $p$ and $q$, respectively, $P_{Seq}$ and $Q_{Seq}$ be corresponding paths in the SeqBDD, and $P_{tree}$ and $Q_{tree}$ be the corresponding paths in the prefix tree.

If $|p| \neq |q|$, the suffixes of $p$ and $q$ are represented using different sets of variables and $P_Z$ and $Q_Z$ can be mapped to $P_{tree}$ and $Q_{tree}$ respectively. If $|p| = |q|$, however, $P_Z$ and $Q_Z$ are merged, and both $P_{tree}$ and $Q_{tree}$ correspond to one path in the ZBDD.

Regardless of the length of $p$ and $q$, the suffix *suffix* is encoded similarly in both sequences and $P_{Seq}$ and $Q_{Seq}$ are merged, which shows the many-to-one mapping between the ZBDD nodes to SeqBDD. □

## 5.1. Weighted Sequence Binary Decision Diagrams

A Weighted SeqBDD is a SeqBDD with weighted edges. In particular, every edge in a Weighted SeqBDD is attributed by an integer value, and each internal node's incoming edge corresponds to the total frequency of all sequences in that node. Thus, the weight of the incoming link is monotonically decreasing as the node is positioned lower in the structure. We define a Weighted SeqBDD node by the following definition.

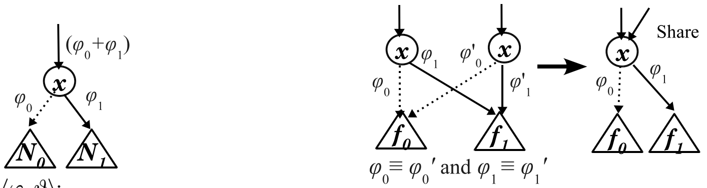
**Definition 5.2.** A Weighted SeqBDD node $N$ is a pair of $\langle \varphi, \vartheta \rangle$ where $\varphi$ is the weight of $N$, and $\vartheta$ is a SeqBDD node.

The weight of node $N$, i.e. $\varphi$, represents the weight of the incoming link to $N$. For a node $N$, we define a function $\mathit{weight}(N) = \varphi$, which gives the total frequency of the sequences in $N$. If $N$ is an internal node, $N_0$ and $N_1$ correspond to two partitions of the database, and $\mathit{weight}(N)$ is the sum of the total frequencies of the sequences in $N_0$ and in $N_1$:

$$\mathit{weight}(N) = \mathit{weight}(N_0) + \mathit{weight}(N_1) \tag{1}$$

Two nodes in a Weighted SeqBDD are merged only if they have the same label, the same respective child-nodes, and also the same weights on the outgoing edges respectively. Hence, a Weighted SeqBDD may be less compact than the non-weighted SeqBDD, since two nodes which contain similar sequences cannot be merged in the Weighted SeqBDD if their respective frequencies are different. Figure 5 illustrates a Weighted SeqBDD node and the Weighted SeqBDD's merging rule.

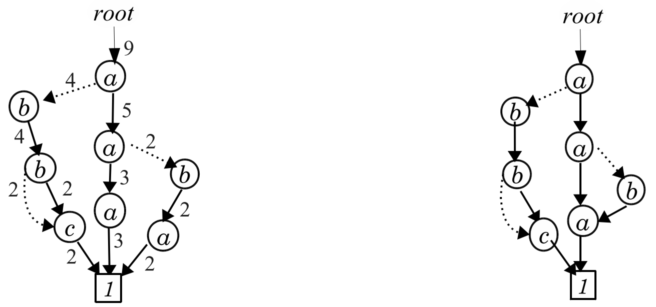
**Example 5.1.** Consider a set of sequences, with their respective frequencies, $S = \{aaa : 3, aba : 2, bc : 2, bbc : 2\}$. Figure 6(a) shows an example of a Weighted SeqBDD representation of $S$. Assuming a lexicographic ordering, the prefix-path $a$ is shared between sequences $aaa$ and $aba$, and prefix $b$ is shared between sequences $bc$ and $bbc$. The lower node representing suffix $c$ is shared between sequences $bc$ and $bbc$, but the bottom $a$-nodes are not merged because they have



(a) Node $N = \langle \varphi, \vartheta \rangle$; $\vartheta = \text{node}(x, N_1, N_0)$; $\varphi = \varphi_1 + \varphi_0$.

(b) Weighted SeqBDD's Merging Rule applies only if $\varphi_0 \equiv \varphi_0'$ and $\varphi_1 \equiv \varphi_1'$

**Fig. 5.** Illustration of Weighted SeqBDDs



(a) Weighted SeqBDD

(b) SeqBDD

**Fig. 6.** Weighted SeqBDD vs SeqBDD for a set of sequences $S = \{aaa : 3, aba : 2, bc : 2, bbc : 2\}$, using a lexicographic variable ordering

different frequencies. As a comparison, the SeqBDD for the same sequences, being 1 node smaller, is shown in Figure 6(b).

**Monotonic property of weights in a Weighted SeqBDD**: When a Weighted SeqBDD is used for storing subsequences, we can use the weights to efficiently prune sequences which do not satisfy the minimum support constraint due to the monotonic property of the weight function. In particular, its monotonicity is described by the following theorem.

**Lemma 5.2.** Given a Weighted SeqBDD node $N$, $\mathit{weight}(N_0) \leq \mathit{weight}(N)$ and $\mathit{weight}(N_1) \leq \mathit{weight}(N)$

*Proof.* The proof for this lemma is straightforward from Equation 1. Since the weight of each node is a positive integer, and it is the sum of the weights of its child nodes, i.e. $N_0$ and $N_1$, the weight of a node is no smaller than the weight of its child nodes. □

Based on Lemma 5.2, if the weight of a node $N$ is less than the minimum support, node $N$ and its child-nodes correspond to infrequent sequences. Thus, they can be safely removed and replaced by the sink-0 node. Being monotonic, the weights in Weighted SeqBDDs may be used for representing other monotonic functions, other than the total frequency which we have defined, such as the maximum (or minimum) length of the sequences.

**Weighted SeqBDD primitive operations:** Basic ZBDD operations (listed in Table 1) such as *set-union* and *set-subtraction* can be adapted for Weighted SeqBDDs (and for the general SeqBDDs). In the following discussion, we show how the weight function can be integrated with ZBDD's set-union operation. When a sequence co-exists across the input SeqBDDs, its frequencies are added.

If both of its inputs are sink-1 nodes, the output is also a sink-1 with the weights of the input nodes being added. Moreover, the weight of each node in the output is computed using Equation 1. Below, we show that the operation is correct, with references made to the corresponding lines in the SeqBDD's *add*() procedure shown in Algorithm 1.

**Theorem 5.2. Correctness of the *add*() procedure:** Given two Weighted SeqBDDs $P$ and $Q$, the $add(P, Q)$ correctly adds the sequences in $P$ and $Q$ and their corresponding frequencies.

*Proof.* Consider the following cases:

1. If $P$ is a sink-0, i.e. $P$ is empty, the output consists of all sequences in $Q$. Similarly, if $Q$ is a sink-0, $P$ is returned as output (line 1-2).
2. If both $P$ and $Q$ are sink-1 nodes, i.e. each of $P$ and $Q$ consists of an empty sequence, then the output also consists of an empty sequence (line 3-5). The weight of the output node is the total frequency between $P$ and $Q$ (line 4).
3. If both $P$ and $Q$ are internal nodes with labels $x$ and $y$.

   (a) Suppose $x = y$ (line 7-8). Since $x$ is the lowest indexed item among the head item of all sequences in $P$ and $Q$, $x$ should be the label of the output node. The 1-child of the output node should contain all sequences which begin with $x$, which are present in $P_1$ and $Q_1$, and the 0-child of the output node contains the remaining sequences.

   (b) Suppose $x$ has a lower index[2] than $y$ (line 9-10), $x$ has a lower index than the head of all sequences in $Q$. Therefore, the output node should be labeled with $x$, and all sequences which begin with $x$ present in $P_1$. The remaining output sequences exist in $P_0$ and $Q$.

   (c) Suppose $x$ has a higher index than $y$ (line 11). This condition is opposite to condition 3(b), since the operation is commutative.

In each of the above cases, any sequence which co-occurs in $P$ and $Q$ is identified when the sink-1 nodes are found. Hence, when a co-occurring sequence is found, its total support is computed bottom-up beginning with the sink nodes, followed by the parent nodes respectively (line 12). □

**Properties of Weighted SeqBDDs:** A Weighted SeqBDD is similar to a prefix tree, except that identical sub-trees are merged, and the frequency values are represented as edge weights instead of node attributes. Table 2 shows a comparison between their structural properties. Weighted SeqBDDs do not have side-links which are needed for finding database projections in the prefix trees. Moreover, node fan-out is allowed in both Weighted SeqBDDs and prefix trees, but node fan-in is only allowed in Weighted SeqBDDs. Node fan-in is maintained in a Weighted SeqBDD by the node merging rule. Thus, without the merging rule, a Weighted SeqBDD would look like a Binary Decision Tree, and contains the same number of nodes as a prefix tree. The caching mechanism in Weighted SeqBDDs allows the cost of projecting the databases to be reduced, by making use of their node fan-in property. Without the caching mechanism, therefore, the traversal cost of a Weighted SeqBDD would be the same as the traversal cost of a prefix tree.

[2] A sink-1 is considered an internal node with the highest index.

**Algorithm 1** SeqBDD's **add()** operation, returns the set-union between two sets of sequences

```
Input: P and Q are the input Weighted SeqBDDs, each of which represents a set of sequences
Output: ⟨Z.weight, Z⟩ : the Weighted SeqBDD containing set-union between P and Q
  Procedure:
  case $P$ is a sink-0 node : return $Z = Q$
  case $Q$ is a sink-0 node : return $Z = P$
  case $Q = P$ is a sink-1 node :
      Calculate weight: $Z.weight = weight(P) + weight(Q)$
      return $\langle Z.weight, Z\rangle$
  Let $Z$ = node($Z.var$, $Z_1$, $Z_0$), $P$ = node($x$, $P_1$, $P_0$), $Q$ = node($y$, $Q_1$, $Q_0$)
  case $x = y$ :
      $Z.var = x$ ; $Z_1$ = add($P_1$, $Q_1$) ; $Z_0$ = add($P_0$, $Q_0$)
  case $x$ has a higher index than $y$ :
      $Z.var = x$ ; $Z_1 = P_1$ ; $Z_0$ = add($P_0$, $Q$)
  case $x$ has a lower index than $y$ : return add($Q$,$P$)
  Calculate weight: $Z.weight = weight(Z_0) + weight(Z_1)$
  return $\langle Z.weight, Z\rangle$
```

An example of a Weighted SeqBDD and a prefix tree, presented as a PLWAP tree (Ezeife and Lu, 2005), is shown in Figure 7, both of which represent a database containing sequences $\{aabac, baba, aaca, bbac\}$. The Weighted SeqBDD is smaller than the prefix tree by 5 nodes, since it merges suffix-path *ac* between sequences *aabac* and *bbac*, and suffix-path *a* between sequences *baba* and *aaca*.

Table 3 shows a comparison between a Weighted SeqBDD and the other types of BDD. When there are no duplicate sequences in a Weighted SeqBDD, the structure has the same number of nodes as a non-weighted SeqBDD, because each sequence has a frequency of 1, and each node which is shared in the non-weighted SeqBDD is also shared in the weighted SeqBDD. The zero-suppression rule is employed in ZBDDs and SeqBDDs, and their weighted variants (Loekito and Bailey, 2007). BMDs (Bryant and Chen, 1995) also have weighted edges, but they use a different weight function. In terms of variable ordering, SeqBDDs employ a global ordering which is applied only upon the 0-child nodes, whereas FreeBDDs (Gergov and Meinel, 1994) allow different paths to have different orderings. The other BDDs apply a global ordering between each node and both of its child nodes.

Table 4 shows some statistics of the Weighted SeqBDD and ZBDD representations for three real data sets (description of these data sets is given in Section 7), when used for representing the frequent subsequences and their frequency values. We do not include the size comparison between the data structures for representing the input sequences, since there are no duplicates in the input data (Note that duplicates may appear in the projected conditional databases). Comparing the size of the data structures for storing the output patterns, which contain many more sequences than any conditional database, is sufficient for analysing their compactness. For *yeast.L200* data set, the Weighted SeqBDD (W-SeqBDD) contains 304 nodes for storing 886 sequences with an average length of 3.3, being about 50% smaller than the Weighted ZBDDs (WZBDDs) which are labeled as WZBDD$_{naive}$ and WZBDD$_{binary}$. For the other two data sets, which contain longer sequences, the W-SeqBDDs are 65%-75% smaller than the WZBDDs. more especially for the snake data set which contains thousands of frequent subsequences. For all the three data sets, the WZBDD$_{binary}$ is always larger than WZBDD$_{naive}$. When the alphabet size is large, such as in the *davinci* data set,

| Property | Weighted SeqBDD | Prefix Tree |
|---|---|---|
| Support storage | As edge weight | In every node |
| Side-links | No | Yes |
| Fan-out | Yes | Yes |
| Fan-in | Yes | No |

**Table 2.** Comparison between Weighted SeqBDD and Prefix Tree

| Data Structure | Zero-suppressed | Weighted Edges |
|---|---|---|
| SeqBDD | Yes | No |
| Weighted SeqBDD | Yes | Yes |
| Weighted ZBDD | Yes | Yes |
| BMD | No | Yes |
| FreeBDD | No | No |

| Data Structure | Variable Ordering |
|---|---|
| SeqBDD | A global variable ordering is enforced between each node and its 0-child |
| ZBDD | A global variable ordering |
| BMD | A global variable ordering |
| FreeBDD | A variable ordering is enforced on the 1-child nodes in each path, different paths may have different variable orderings |

**Table 3.** Comparison between (Weighted) SeqBDD and other BDD Variants

| Dataset | minsup | Num. of subsequences | Average length |
|---|---|---|---|
| yeast.L200 | 0.8 | 886 | 3.3 |
| snake | 0.3 | 4693 | 4.6 |
| davinci | 0.0015 | 785 | 4 |

| Dataset | $\vert\text{W-SeqBDD}\vert$ | $\vert\text{WZBDD}_{naive}\vert$ | $\vert\text{WZBDD}_{binary}\vert$ | $\vert\text{PrefixTree}\vert$ |
|---|---|---|---|---|
| yeast.L200 | 304 | 693 | 669 | 885 |
| snake | 983 | 4212 | 5257 | 4692 |
| davinci | 357 | 1000 | 1901 | 783 |

**Table 4.** Characteristics of the represented subsequences, and size comparison between Weighted SeqBDD (W-SeqBDD), WZBDD$_{naive}$, and WZBDD$_{binary}$, in terms of the number of nodes

the WZBDD$_{binary}$ contains twice the number of nodes of the WZBDD with the naive encoding. Compared to prefix trees, the Weighted SeqBDDs allow more than 50% compression, whereas the WZBDD representations are less compact when the alphabet size is large or a large number of sequences exist.

**Complexity analysis:** In the following discussion, we discuss the computational complexity for creating and manipulating nodes in the general SeqBDDs, which is also applicable to the Weighted SeqBDDs. The computational cost for creating a SeqBDD node requires one look-up operation to the *uniquetable*, which is the same as that in a BDD.

**Theorem 5.3.** The time complexity for creating a SeqBDD that consists of $N$ nodes is $O(N)$.

*Proof.* When creating a node, the hash key for that node is computed and one lookup operation is performed to find a pre-existing entry in the unique table. Given that nodes with the same hash key are stored as a sorted linked list, the lookup operation requires a single traversal of that list which has an average size of $\frac{n}{m}$, where $n$ is the number of nodes and $m$ is the size of the table. Based on the existing hash table implementation in JINC (Ossowski and Baier, 2006), $\frac{n}{m}$ is kept to a small value, hence, allowing $O(1)$ time complexity for insertion and lookup operations. Thus, when creating $N$ nodes, the overall time complexity is $O(N)$. □

**Theorem 5.4.** Given a database containing $k$ sequences with a maximum length of $L$. The number of nodes in the SeqBDD representation is bounded by $kL$, and the height of the SeqBDD is bounded by $L$.

*Proof.* In the given database, there are $kL$ elements in total. In the worst case, each of those elements is represented by a node in the SeqBDD, thus, the SeqBDD consists of $kL$ unique nodes. The height refers to the maximum length of any path in the SeqBDD, which is $L$, which refers to the maximum length of the input sequence since each path corresponds to a unique sequence. □

Note: in practice, the number of nodes is much smaller than $kL$, since many nodes in a SeqBDD which correspond to common prefixes, or common suffixes, may be shared across multiple sequences. Since the number of nodes of the SeqBDD is $O(kL)$, hence, its construction has $O((kL))$ time complexity.

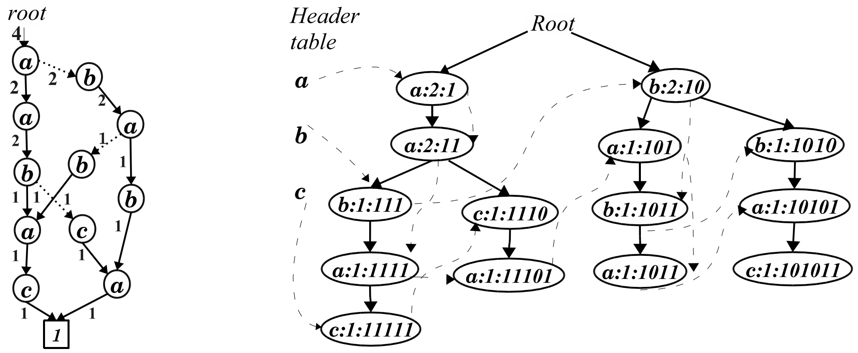
The basic operations, such as set-union and set-subtraction, have polynomial time-complexity with respect to the number of nodes in the input SeqBDDs. Our algorithm for mining frequent subsequences employ the primitive SeqBDD's *add*() operation which adds two sets of sequences and combines their frequencies. The complexity analysis of such operation follows.

**Theorem 5.5.** Given two SeqBDDs $P$ and $Q$, and a binary operation <*op*>($P$, $Q$), where <*op*> is one of the primitive operations *add*, or *subtract*. The time complexity of <*op*>($P$,$Q$) is $O(|P| + |Q|)$.

*Proof.* The primitive operations visit each node from the two input nodes at most once. Hence, the number of recursive calls is $O(|P| + |Q|)$. Since in each recursive call, a new node may be created, the total number of unique nodes (including the input and the output nodes) is $O(|P| + |Q|)$. In the worst case, no nodes are merged between the input and the output SeqBDDs. Hence, the overall time complexity for the operation is $O(|P| + |Q|)$. □

**Caching mechanism:** To optimise and reduce computational cost, SeqBDDs employ the same type of caching mechanism as is used by BDDs. In our implementation, we use the caching library of JINC (Ossowski and Baier, 2006).

Note the term "caching", here, has somewhat different meaning from previous work on frequent pattern mining, such as (Ghoting, Buehrer, Parthasarathy, Kim, Nguyen, Chen and Dubey, 2005), which describes cache conscious trees for mining frequent itemsets. There, the intention is to minimise access to physical memory, by storing frequently accessed nodes of of the data structure. On the other hand, the SeqBDD's caching mechanism has a different aim, which is to store the *results* of intermediate computations, corresponding to subtrees and then make them available for re-use when the subtree is visited along a different path in the SeqBDD. Caching is automatically provided by the SeqBDD library implementation, and thus it is not explicitly shown in the algorithm described shortly.



(a) Weighted SeqBDD

(b) Prefix tree

**Fig. 7.** Weighted SeqBDD vs Prefix tree for a data set containing sequences $\{aabac, baba, aaca, bbac\}$, using a lexicographic variable ordering. The SeqBDD contains 4 fewer nodes than Prefix tree through merging of lower $a$-$c$ nodes and lower $a$-node.

Note that SeqBDD caching is not possible for tree like structures (such as a prefix tree), since subtrees in a prefix tree can be visited by only one path, whereas subtrees in a SeqBDD can be visited by several paths (and thus cached for reuse between alternate paths). Without a caching mechanism, the time complexity of manipulating a BDD, as well as a SeqBDD, is similar to that of a binary tree, since the computation of a node would need to be repeated as many times as the number of prefix-paths through which that node is visited.

## 6. Mining Frequent Subsequences

We call our algorithm for mining frequent subsequences **SeqBDDMiner**. SeqBDDMiner follows a prefix growth mechanism similar to Prefixspan in (Pei et al., 2004), but uses a Weighted SeqBDD for representing the database. In the remainder of the paper, unless otherwise stated, we refer to the Weighted SeqBDD by its general term SeqBDD. Unlike Prefixspan, our algorithm physically creates the conditional databases but this benefits mining, since the SeqBDD allows nodes to be shared across multiple databases, and allows the results to be shared between similar intermediate computations. Using the SeqBDD's caching principle, moreover, construction of the conditional databases can be performed efficiently.

The initial SeqBDD representation of the database is built by *add*-ing the input sequences. The construction procedure is shown in Algorithm 2. For an item $x$, the $x$-conditional database contains suffixes of the first occurrence of $x$ from each sequence in the input database. We find an *f-list* for each item $x$, which is an ordered list of elements which are frequent in the conditional database. This list allows early pruning of the conditional database by removing items which do not appear in the *f-list*. For efficiency purpose, the ordering in this list is inverted from the SeqBDD's global variable ordering, which allows the output node to be built bottom-up incrementally.

For finding the frequency of an item, we define a SeqBDD-based operation which follows a divide-and-conquer strategy. Given an input SeqBDD database $P$, and an item $x$. Let $var$ be the label of $P$. The frequency of $x$ is found by recursively adding its frequency in the two database partitions: $P_1$ and $P_0$. The

**Algorithm 2** Algorithm **buildDB** for creating the initial Weighted SeqBDD from a sequence data set

**Input:** $D$ : a sequence data set.
**Output:** *initDB* : the Weighted SeqBDD containing sequences in the input data set $D$.
**Procedure:**

```
1: (Initialise $initDB$ to be a sink-0 node.) $initDB = 0$
2: for each sequence $s$ in $D$ do
3:     (Build a SeqBDD which contains a sequence $s$.)
       $P_s$ = a Weighted SeqBDD containing sequence $s$
4:     (Add $P_S$ to $initDB$.) $initDB = initDB \bigcup P_s$
5: end for
```

Note: $A \bigcup B$ denotes the $add(A, B)$ operation where $A$ and $B$ are SeqBDDs.

first partition, $P_1$, contains sequences which begin with $var$, the second partition contains the remaining sequences. If $var = x$, $weight(P_1)$ gives the frequency of $x$ in the first database partition. On the other hand, if $var \neq x$, the frequency of $x$ in the first database partition is computed recursively in $P_1$. The recursion in $P_1$ terminates at the first occurrence of $x$ in each branch. The similar recursion procedure is performed in $P_0$. We define the operation $frequency(P, x)$ for finding the frequency of item $x$ in a SeqBDD $P$ as the following. If $P$ is a sink node, $frequency(P, x) = 0$, otherwise,

$$frequency(P, x) = \begin{cases} frequency(P_1, x) + frequency(P_0, x) & \text{if } P.var \neq x \\ weight(P_1) + frequency(P_0, x) & \text{if } P.var = x \end{cases}$$

For an item $x$, finding the $x$-suffixes (and the conditional databases), which is a major component in our mining algorithm, we define a SeqBDD operation *suffixTree* using a similar recursive strategy as the above discussed *frequency* operation. Given a SeqBDD $P$, and an item $x$, $suffixTree(P, x)$ is the set of $x$-suffixes which are contained in $P$, excluding the head elements, which we call as the *x-suffix tree*. If $P$ is a sink node, $suffixTree(P, x) = \emptyset$, otherwise,

$$suffixTree(P, x) = \begin{cases} suffixTree(P_1, x) \bigcup suffixTree(P_0, x) & \text{if } P.var \neq x \\ P_1 \bigcup suffixTree(P_0, x) & \text{if } P.var = x \end{cases}$$

**Optimizations:** The first optimization is called *infrequent database pruning*, which is based on the monotonic property of the weights in a SeqBDD. If the weight of the top node is less than *min_support*, then the conditional databases can be safely pruned. Additionally, SeqBDDMiner has a number of optimizations which rely on the use of the SeqBDD's caching library:

- **Caching of intermediate results.** The frequent subsequences from each conditional database are stored in a cache table called the *patternCache*, to be re-used if the same conditional database is projected by some other prefixes. We define an operation, *patternCache*[$P$], to obtain the cached output for a database $P$.
- **Caching of pruned conditional databases**. For a given item $x$, and an input database, the (pruned) *conditionalDB* is stored in a cache table, using a similar mechanism as the *patternCache*. The procedure for obtaining the *conditionalDB* comprises of 3 operations: 1) find the $x$-suffix tree, 2) find its *f-list*, 3) prune infrequent items from the $x$-suffix tree. Each operation is associated with its own cache, so that various databases, which may share common sub-structures, may share the results of their intermediate computations.

**Algorithm 3 SeqBDDMiner** for mining frequent subsequences

```
Input: $inputDB$ : a SeqBDD containing the input database
    $min\_support$ : the minimum support threshold
    $f$-$list$ : a list of items which occur frequently in $inputDB$
Output: $allFS$ : a SeqBDD containing the frequent subsequences
Procedure:
 1: (Infrequent database pruning.) if ( $weight(inputDB) < min\_support$), then return 0
 2: (Terminal case.) if ( $inputDB$ is a sink node ), then return $inputDB$
 3: (Do cache lookup.) if ( $patternCache[inputDB]$ is not empty ), then
    return $patternCache[inputDB]$
 4: (Initialise the output node.) $allFS = 1$
 5: for each item $x$ in $f$-$list$ do
 6:   (Find the $x$-suffix tree.)
      $x$-suffix tree = compute $suffixTree(inputDB, x)$
 7:   (Find the frequent items in $x$-suffix tree. )
      $f|_x$-list = the list of frequent items in $x$-suffix tree
 8:   (Prune infrequent items from $x$-suffix tree. )
      $x$-condDB = remove items which do not appear in $f|_x$-list from
      $x$-suffix tree
 9:   (Find frequent subsequences with prefix $x$ from the conditional DB.)
      $x$-FS = $x \times$ SeqBDDMiner($x$-condDB, $min\_support$, $f|_x$-list)
10:   (Incrementally build the output node.) $allFS = add(x\text{-}FS, allFS)$
11: end for
12: (Cache the output patterns.) $patternCache[inputDB] = allFS$
13: return $allFS$
```

Note : $x \times P$ appends item $x$ to the head of every sequence in $P$, by creating a $node(x, P, 0)$, where $P$ is a SeqBDD.

**Details of the mining algorithm:** The SeqBDDMiner algorithm is shown in Algorithm 3, which we will explain line-by-line. The procedure begins with the input SeqBDD containing the initial data set with the infrequent items being removed, the *f-list* contains the frequent items which are ordered in reverse to the SeqBDD's variable ordering. Firstly, the infrequent database pruning is applied if the total frequency of the sequences is less than the minimum support (line 1). The function terminates if the database is empty (line 2), or if the output is found in the *patternCache*. In the latter case, a pointer to the cached output is returned (line 3). For each item $x$ in *f-list*, it projects the *x-suffix tree* (line 6). Prior to its processing, the $x$-conditional database is pruned in 2 steps (line 7-8): i) Find the frequent items in *x-suffix tree* using the pre-defined SeqBDD's *frequency*() operation; ii) Remove the infrequent items from *x-suffix tree*. Then, the function is called recursively on the $x$-conditionalDB (line 9). When the locally frequent patterns are returned from the $x$-conditionalDB, $x$ is appended to the head of every pattern. The output node is built incrementally by adding the intermediate outputs for all such $x$ (line 10).

**Example 6.1.** Reconsider the set of sequences $D$ in Example 4.1. Let *min_support* be 3. Construction of the initial SeqBDD database is shown in Figure 8, which incrementally adds the individual SeqBDD of each sequence. Let $S_i$ be the SeqBDD which contains sequence $p_i$, where $i = \{1, 2, 3, 4\}$. The conditional databases for item $c$ and $b$ are shown in Figure 9. The first item in *f-list* is $c$, hence, *c-suffix tree* is built by finding suffixes $\{\}$ from $S_1$, $a$ from $S_3$, and $\{\}$ from $S_4$, and then adding them together. The *f-list* in *c-suffix tree* is empty. Thus, the resulting *c-conditionalDB* contains an empty set $\{\}$ with a support of 3, no more patterns

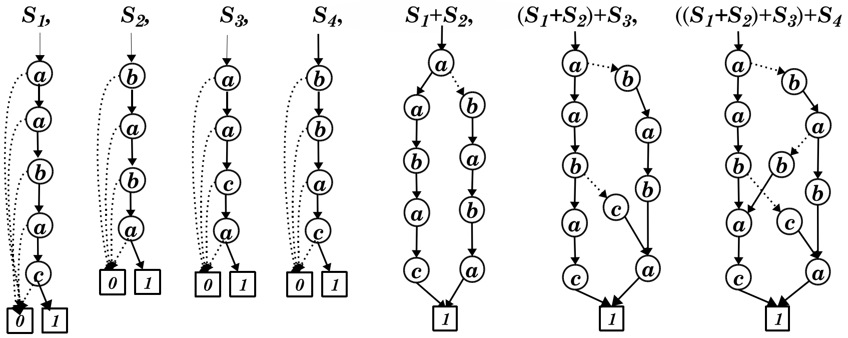


**Fig. 8.** SeqBDDs representing $S_1 = \{aabac\}$, $S_2 = \{baba\}$, $S_3 = \{aaca\}$, $S_4 = \{bbac\}$, $S_1 + S_2$, $(S_1 + S_2) + S_3$, and $((S_1 + S_2) + S_3) + S_4$ with lexicographic variable ordering

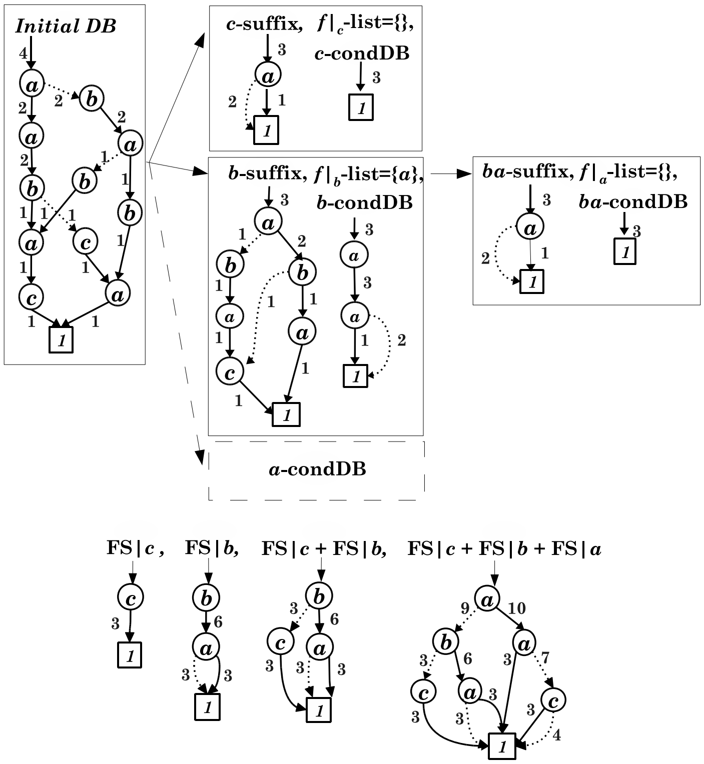


**Fig. 9.** The initial database, and conditional databases when growing prefix $c$ and prefix $b$, and the corresponding locally frequent subsequences, labeled as $FS|_c$ and $FS|_b$, respectively. $FS|_c + FS|_b$ shows the combined frequent subsequences. $FS|_c + FS|_b + FS|_a$ shows the globally frequent subsequences. ( $minsup = 3$ )

can be grown. The output node for prefix $c$, containing $\{c : 3\}$ is shown under label $FS|_c$.

The next item in *f-list* is item $b$. *b-suffix tree* contains suffixes $ac, aba, bac$, whose $f$-list (labeled as *$f|_b$-list*) contains only item $a$. Then, SeqBDDMiner() is called on *b-conditionalDB*. Consequently, prefix $a$ is being grown from *b-conditionalDB*, resulting frequent subsequences $b : 3$ and $ba : 3$, labeled as $FS|_b$. The output node, *allFS*, which previously contains $FS|_c$, is now combined with $FS|_b$. The new output node is labeled $FS|_c + FS|_b$. Since for every prefix item $x$, the highest node of *allFS* always has a lower index than $x$ (due to the reversed item-ordering in *f-list*), $FS|_c + FS|_b$ can be obtained by simply appending *allFS* to the 0-child of $FS|_b$. The same procedure is performed for the last item, $a$, obtaining patterns $\{a : 4, ac : 3, aa : 3\}$.

**Soundness and completeness of the algorithm:** We will show that the algorithm is sound and complete, based on properties of the Weighted SeqBDD. As earlier discussed, the output patterns are grown recursively based on the frequent items from each conditional database. Thus, we need to show that the frequent items are correctly found from the given database. We show this by proving the correctness of the *frequency* SeqBDD operation.

**Theorem 6.1.** Given a SeqBDD $P$ and an item $x$. The function *frequency*$(P, x)$ correctly calculates the frequency of item $x$ in the given database $P$.

*Proof.* The total frequency of an item $x$, can be derived from paths which contain $x$, since each path maps to a unique sequence. Firstly, if the database contains only a single path, which may contain multiple occurrences of $x$, the weight of the highest $x$-node gives the frequency of $x$ in the database, since any lower $x$-node belongs to the same sequence. Secondly, if the database contains multiple paths and there are two $x$-nodes $A$ and $B$ such that each one is the highest $x$-node in their corresponding paths, $A$ and $B$ represent two separate partitions of the database and thus, the total weights of $A$ and $B$ gives the frequency of $x$ in the overall database. □

**Theorem 6.2.** The *SeqMiner* algorithm is sound and complete.

*Proof.* We will prove this theorem in two parts: (a) the algorithm is sound, and (b) the algorithm is complete.

**Soundness:** We will show that growing prefix $x$ from an item in the $x$-conditional database generates a frequent subsequence pattern. Given a prefix $x = x_1, x_2, \ldots x_n$, and its conditional database $P$, by definition, $P$ contains items which are frequent in the longest $x$-suffixes from the original database. The frequency of each item in the original database is no smaller than its frequency in $P$. More specifically, the frequency of each item $i$ in $P$, is the number of sequences in the original database which contain subsequence $x_1, x_2, \ldots x_n, i$. Thus, if $i$ occurs in $P$, growing $x$ with item $i$ generates subsequence $x_1, x_2, \ldots x_n, i$ which is frequent in the original database.

**Completeness:** We will show by contradiction that there exists no frequent subsequence patterns which contain an item which does not occur in the conditional database of any of its prefixes.

Suppose there exists a frequent subsequence pattern $q = q_1, q_2, \ldots q_{n-1}, q_n$ which is not found by the algorithm, because item $q_n$ does not occur in the conditional database of one of the prefixes of $q$. Let $x$ and $y$ be two prefixes of $q$, such that $x = q_1, q_2, \ldots q_m$, where $m < (n-1)$, and $y = q_1, q_2, \ldots q_m, \ldots q_{n-1}$. Let

$P$ be the $x$-conditional database. Suppose item $q_n$ does not occur in $P$ because $z = q_1, q_2, \ldots q_m, q_n$ is infrequent. The subsequence $z$ is also a subsequence of the frequent pattern $q$, which contradicts the anti-monotonic property which says that all subsequences of a frequent subsequence pattern are also frequent. Therefore, such a subsequence pattern $q$ does not exist, and the algorithm is complete. □

**Complexity analysis:** To analyse the complexity of our mining algorithm, we consider the space complexity for constructing the initial database, the final output, and the conditional databases, in terms of the number of nodes. The time complexity can be derived from the space complexity by Theorem 5.3, which is $O(N)$ where $N$ is the number of nodes.

Given a SeqBDD $P$ which represents a sequence database, and a SeqBDD $S$ which represents the frequent subsequences in $P$. Let $L$ be the maximum length of the sequences in $P$, $k$ be the number of sequences in $P$.

**Theorem 6.3.** The total number of nodes for constructing the initial database is $O(k^2.L)$.

*Proof.* The initial database is constructed by incrementally adding the SeqBDD representation of each sequence to the database. Let $A$ and $B$ be two SeqBDDs, where $A$ contains the SeqBDD for the first $n-1$ sequences, $B$ contains the SeqBDD for the $n$-th sequence, where $n = \{1, 2, \ldots k\}$,. Since $|A|$ is bounded by $(n-1).L$, and $|B|$ is bounded by $L$, then the output of $add(A, B)$ contains $O(|A| + |B|)$ nodes, which is equal to $O(n.L)$. So, the overall time complexity to build the complete database is $\sum_{n=1}^{n=k} O(n.L)$, which is $O(k^2.L)$. □

**Theorem 6.4.** The total number of nodes for constructing the conditional databases is $O(k^2.L^2.2^L)$.

*Proof.* The number of frequent subsequences is $O(k.2^L)$, and each subsequence contains $O(L)$ elements. Therefore, the total number of nodes in the output SeqBDD is $O(k.L.2^L)$, which corresponds to the number of conditional databases. Since the size of an $x$-conditional database is bounded by the $x$-suffix tree, let us consider the following cases when processing *suffixTree(P,x)*. Let $S = \mathit{suffixTree}(P, x)$. It is straightforward to show that $|S| = O(|P|)$ if $P$ is a sink node or $P$ contains only one sequence. If $P$ is not a sink node, the output is obtained by adding $\mathit{suffixTree}(P_1)$ (or $P_1$ if the label of $P$ is $x$) with $\mathit{suffixTree}(P_0)$, each of which contains $O(|P|)$ nodes. Hence, the overall size of the output is $O(|P|)$, which is $O(k.L)$ by Theorem 5.4, and the total number of nodes used by the conditional databases is $O(k^2.L^2.2^L)$. □

Note: In practice, many nodes are shared across multiple conditional databases, and many conditional databases are pruned by the infrequent database pruning. Hence, the total number of nodes is much less than $k^2.L^2.2^L$. Moreover, SeqBDD's caching principle avoids redundant node constructions by allowing any of the computation results (the suffix trees and conditional database) for each subtree to be cached and re-used when needed.

## 7. Performance Study

In this section we present experimental results to compare the performance of our **SeqBDDMiner** algorithm, which is based on our proposed Weighted Sequence

| Data set name | $\lvert D \rvert$ | $N$ | $L$ | $C$ | Category |
|---|---|---|---|---|---|
| yeast.L200 | 25 | 4 | 129 | 35 | Short |
| DENV1 | 9 | 4 | 50 | 50 | Long |
| snake | 174 | 20 | 25 | 25 | Short |
| PSORTb-ccm | 15 | 20 | 50 | 50 | Long |
| gazelle | 29369 | 1451 | 652 | 3 | V.short |
| davinci | 10016 | 1108 | 416 | 2 | V.short |
| C2.5.S5.N50.D40K | 17808 | 50 | 42 | 3 | V.short |

$\lvert D \rvert$ = Number of sequences in the data set
$N$ = Number of items in the domain
$L$ = Maximum sequence length
$C$ = Average sequence length
V.short = Very short

**Table 5.** Data set characteristics and a proposed categorisation based on the average sequence length

Binary Decision Diagrams, with the state-of-the-art prefix-growth algorithms such as PLWAP (Ezeife et al., 2005), and Prefixspan which has been shown superior in (Pei et al., 2004). We implemented our SeqBDDMiner algorithm using the core library functions from existing BDD package, JINC[3]. All implementations were coded in C++, all tests were performed on four 4.0 GHz CPUs with 32 GB RAM, running Redhat Linux 5, with a CPU time-out limit of 100,000 seconds per mining task. The JINC library provides some utility for maintaining the uniqueness of each node, and for maintaining the cache tables for each type of SeqBDD operations. We use the default table parameters as provided by the author of the package.

Our experiments aim to analyze the following factors: 1) **SeqBDD's compactness** : the amount of data compression which can be achieved by the (Weighted) SeqBDD due to its fan-out and fan-in; 2) **Runtime performance**: the runtime performance of our SeqBDD-based algorithm in comparison to the other encodings discussed in Section 4.3, and in comparison to the existing prefix growth algorithms. We will also analyze the effects of increasing similarity of the sequences, which would increase the length (and volume) of the patterns. 3) **Effectiveness of pattern caching and node sharing**: how much database projection is avoided due to pattern caching and the effects of node sharing in the running time of SeqBDDMiner. We analyse three types of real data sets: i) DNA sequence data sets, ii) protein data sets, and iii) weblog data sets. Their characteristics are shown in Table 5. Detailed descriptions of each type of data set are provided shortly. Note that the characteristics of these datasets push the limits of state of the art frequent subsequence mining algorithms. It would not be feasible to use datasets which contain both i) very long and ii) very many input sequences, due to the massive number of output patterns that would need to be generated.

**DNA sequence data sets** typically contain long sequences which are defined over 4 letters, i.e. $A, C, G, T$. Due to the small alphabet size, the sequences may be highly similar and a large number of long frequent subsequences ex-

[3] JINC was developed by the author of (Ossowski and Baier, 2006) for studying a different type of weighted BDDs

ist. We choose 2 data sets from the NCBI's website (NCBI, n.d.): *yeast.L200*[4], which contains the first 25 sequences, with a maximum length of 200 elements, and DENV1[5], which contains genes from dengue virus. We remove any sequence duplicates for our experiments. Due to the large number of patterns, we only use the first 50 elements from each sequence, to allow mining to complete within a reasonable time with low support thresholds.

**Protein sequence data sets** are defined over 20 letters which are also relatively dense. The two data sets are *snake* (Wang and Han, 2004), and *PSORTb-ccm* (She et al., 2003) which is smaller and more dense. Due to the length of the input sequences, we only use the first 25 elements from each sequence in the *snake* dataset, and 50 elements from each sequence in the *PSORTb-ccm* data set.

**Weblog data sets:** Compared to the biological data sets, weblog data sets have a larger domain and the sequences are relatively shorter. In particular, mining frequent subsequences in the weblog data sets is challenging when the minimum support is low due to the large number of sequences. We choose two weblog data sets : i) *gazelle* (Wang and Han, 2004), ii) *davinci* (Ezeife and Lu, 2005).

**The synthetic data sets** were generated using the sequential synthetic data generator in (IBM, n.d.) The first data set, $C2.5.S5.N50.D40K$ consists of 40,000 sequences, defined over 50 items, with average sequence length of 2.5, and the average length of maximal potentially frequent sequence is 5. This data set contains shorter sequences than the weblog data sets, although it has a smaller domain. Secondly, we use the synthetic data generator to generate data sets with a varied value of $N$ (i.e. number of items in the domain) to analyse the effects of the alphabet size to the algorithm's performance. Moreover, to analyse the effects of the similarity of the sequences, we choose the protein *PSORTb-ccm* and append an increasing length of synthetically-generated common prefixes and common suffixes to each sequence.

### 7.1. Compactness of SeqBDDs due to fan-out and fan-in

In this subsection, we examine the compactness of SeqBDDs for compressing a sequence database, due to their node fan-out and node fan-in. To calculate the fan-out compression factor, we implement a Sequence Binary Decision Tree (SeqBDTree), which is a relaxed type of SeqBDD with no node fan-in. We then calculate the compression being achieved due to node fan-out, i.e. *fanOut* and *fanIn*. [6], *fanOut* is the number of nodes in the SeqBDTree, counted as a proportion of the the total number of elements in the data set. *fanIn* is the node-count difference between SeqBDD and SeqBDTree, as a proportion of the total number of nodes in the SeqBDTree (which does not have fan-in). Table 6 shows for each data set, the *fanOut* and *fanIn* compression factors. The mining times using either data structures for a few representative data sets will also be shown shortly.

[4] *yeast.L200* is obtained from NCBI's website using query: *yeast [organism] AND 1:200 [sequence length].*

[5] *DENV1* is obtained from NCBI's website using query: *dengue virus type 1 AND 1:100 [sequence length]*;

[6] $fanOut = 1 - \frac{|SeqBDTree|}{total\_number\_of\_elements}$, $fanin = \frac{|SeqBDTree| - |SeqBDD|}{|SeqBDTree|}$, $|SeqBDTree| =$ the number of nodes in the *SeqBDTree*

| Dataset name | $fanOut$ | $fanIn$ | Category |
|---|---|---|---|
| yeast.L200 | 0.22 | 0.12 | Similar |
| DENV1 | 0.53 | 0.07 | Similar |
| snake | 0.52 | 0.10 | Highly similar |
| PSORTb-ccm | 0.024 | 0.004 | Dissimilar |
| gazelle | 0.42 | 0.16 | Similar |
| davinci | 0.59 | 0.03 | Similar |
| C2.5.S5.N50.D40K | 0.52 | 0.18 | Highly similar |

**Table 6.** Fan-out and Fan-in compression factors of SeqBDD and a proposed data set categorisation

We find that the compression factors are dependent on the similarity or the length of the sequences. Intuitively, for data sets which contain long sequences, the SeqBDDs are likely to have more node-sharing. But it is not always the case. The *PSORTb-ccm* data set has very low *fanOut* and *fanIn* compared to the other data sets, indicating that although the sequences are long, they do not share many prefixes nor suffixes. On the other hand, there are data sets such as *davinci* and *C2.5.S5.N40.D40* which contain short sequences, but their SeqBDDs have a significant node sharing as shown by their large *fanIn* or *fanOut* factors. Based on these compression metrics, we roughly classify the data sets into the following categories:

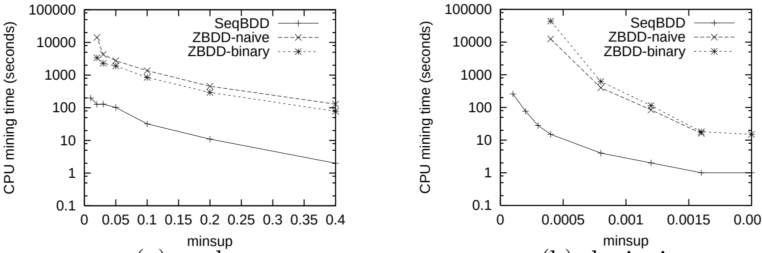
- Highly similar: *fanOut* $\geq 0.5$ and *fanIn* $\geq 0.1$.
- Similar: *fanOut* $\geq 0.5$, or *fanIn* $\geq 0.1$.
- Dissimilar sequences: *fanOut* $< 0.5$ and *fanIn* $< 0.1$.

## 7.2. Runtime performance of the mining algorithm

In Table 4 (Section 5.1), we showed some statistics in terms of the number of nodes in (Weighted) SeqBDD, and in ZBDD (with two alternative itemset encoding scheme) for representing the frequent subsequence patterns. We now show the mining time comparison between either type of data representations when they are used for mining frequent subsequences in *snake* and *davinci* data sets. Figure 10 shows that SeqBDDMiner is uniformly superior in both data sets, being at least 10 times faster than either ZBDD representation. Interestingly, ZBDD$_{naive}$ is faster than ZBDD$_{binary}$ in the *davinci* data set, but slower in the *snake* data set. The statistics in Table 4 show that ZBDD$_{binary}$ is faster when its size is not much larger than ZBDD$_{naive}$.

**DNA sequence data sets:** The runtime performance comparisons are shown in Figure 11. In the *yeast.L200* data set, SeqBDDMiner is 10 times slower than PLWAP and 100 times slower than Prefixspan when the support threshold is 70% or larger, but its running time grows exponentially slower as the threshold decreases. SeqBDDMiner has, moreover, higher scalability since it can finish mining given a support threshold as low as 5% in under 1000 seconds, whereas both PLWAP and PrefixSpan could not finish within the CPU time limit given a support threshold.

In the *DENV1* data set, which contains similar sequences, SeqBDDMiner is



(a) snake (b) davinci

**Fig. 10.** Mining time comparison between SeqBDD, $\text{ZBDD}_{naive}$, and $\text{ZBDD}_{binary}$

substantially the most efficient, whereas PLWAP and Prefixspan are exponentially slower with respect to a decreasing minimum support. For support threshold values less than 60%, Prefixspan and PLWAP could not complete within the CPU time limit. This shows the benefits of using SeqBDDs for mining highly similar sequences, for which both PLWAP and Prefixspan have more limited scalability.

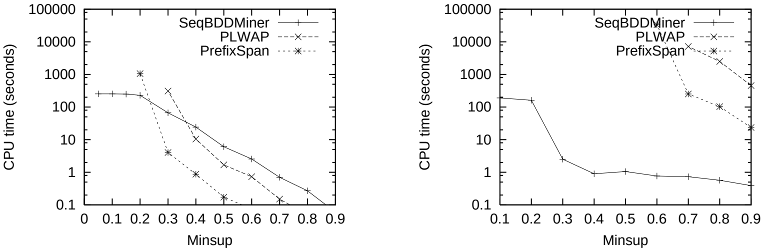
**Protein sequence data sets:** The runtime performance comparisons are shown in Figure 12. For both datasets, SeqBDDMiner is more scalable than the other algorithms when the support threshold value is low. More specifically in the *snake* data set, given a support threshold value as low as 2%, SeqBDDMiner completes mining within 500 seconds which is 4 times faster than PrefixSpan, and PLWAP could not complete mining within the CPU time limit. In general, the runtimes of both PLWAP and Prefixspan grow exponentially slower than SeqBDDMiner as the support threshold decreases. Similar trends are found in the small and dense data set, *PSORTb-ccm*.

**Weblog data sets:** The runtime performance comparisons are shown in Figure 13. Overall, the runtime of SeqBDDMiner is slower than both PLWAP and Prefixspan for high support threshold values, but SeqBDDMiner grows exponentially slower than the other algorithms with respect to a decreasing support threshold value.

In the *gazelle* data set, for support threshold larger than 0.05%, SeqBDDMiner is up to 10,000 times slower than the both PLWAP and Prefixspan. But for a support threshold value as low as 0.02% (lower for PrefixSpan), SeqBDDMiner spends about 50,000 seconds, whilst PLWAP could not complete mining within the CPU time limit. In the *davinci* data set, which has a larger *fanOut* factor than *gazelle*, SeqBDDMiner is up to 100 times faster than PLWAP and Prefixspan for support threshold 0.03% or lower.

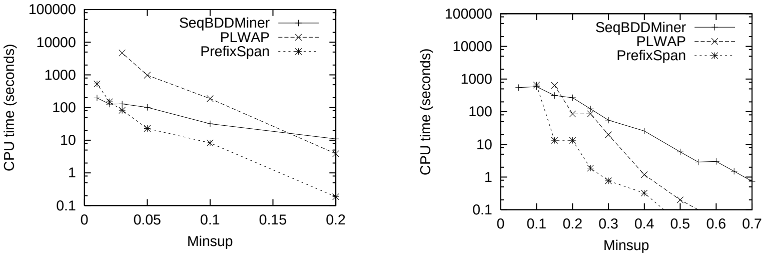
**Synthetic data sets:** The mining time for mining frequent subsequences in $C2.5.S5.N50.D40K$ data set is shown in Figure 14(a). Prefixspan has the best runtime performance in either data set, being 250 times faster than SeqBDDMiner. When compared against PLWAP, the runtime of SeqBDDMiner grows slower than PLWAP as the support threshold decreases, and more specifically, SeqBDDMiner is up to four times faster than PLWAP when the support threshold is lower than 0.04%.

Figure 14(b) shows the effects of increasing the domain size on various synthetic data sets, each of which is generated using a fixed $S10.D10K$ parameter and the domain size is varied between $4, 6, 8, 10, 20, 30, 40$, and $50$, with a minimum support threshold being 25%. Having fewer items in the domain conse-



(a) yeast.L200 (b) DENV1

**Fig. 11.** Mining times in DNA data sets

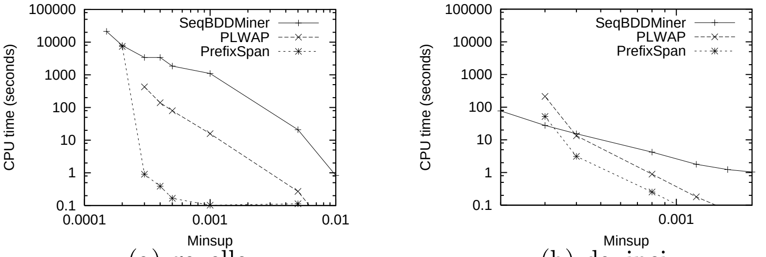


(a) snake (b) PSORTb-ccm

**Fig. 12.** Mining times in protein data set

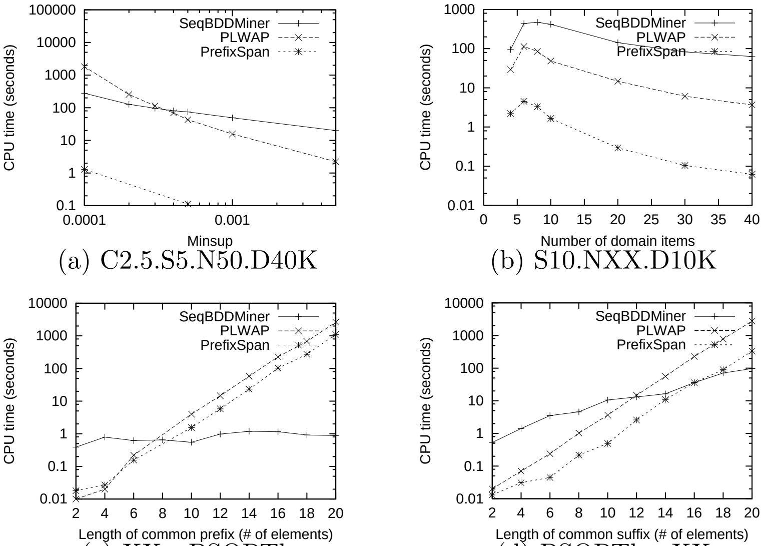
quently generates more similar sequences, and longer frequent subsequences. It shows that PLWAP and Prefixspan have similar relative runtime performance, but SeqBDDMiner becomes more competitive as the number of domain items decreases.

Lastly, we analyse the effects of similarity of the input sequences to the algorithms' performance by appending an increasing length of synthetic common prefix, or common suffix, to the *PSORTb-ccm* data set. Figure 14(c) and (d) show the trends of running time for each scenario given a support threshold of 80%. It shows that SeqBDDMiner is superior, having an almost constant (i.e. very small increase) in its mining time as the length of common prefix increases up to 20 items, whereas PLWAP and PrefixSpan increase exponentially. As the length of common suffix increases, SeqBDDMiner benefits from sharing of com-



(a) gazelle (b) davinci

**Fig. 13.** Mining times in weblog data sets



(a) C2.5.S5.N50.D40K (b) S10.NXX.D10K

(c) XX—PSORTb (d) PSORTb—XX

**Fig. 14.** Mining times in synthetic data sets

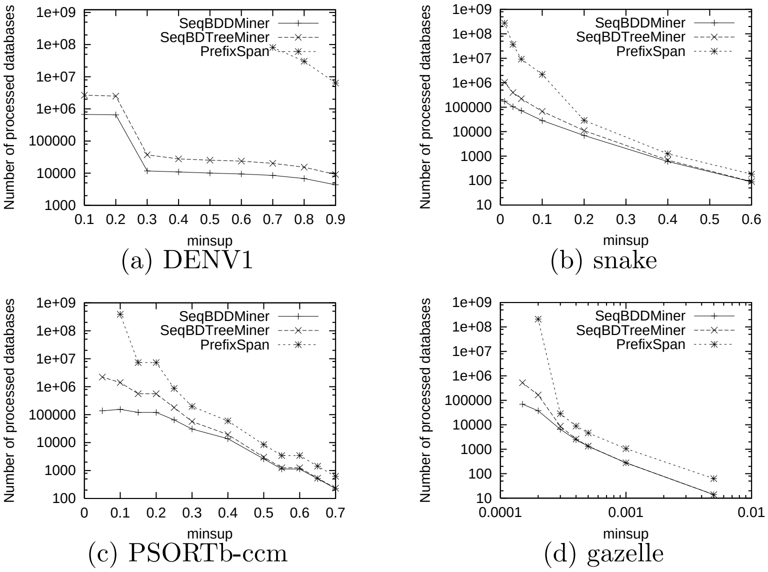
mon sub-trees with a linear growth of running time whilst the other algorithms have an exponentially increasing running time.

### 7.3. Effectiveness of SeqBDDMiner due to pattern caching and node sharing

In order to analyse the effectiveness of the BDD's caching ability, we compare the number of database projections performed by SeqBDDMiner against PrefixSpan. For the case of SeqBDDMiner, if a conditional database exists in the *patternCache* cache, then no further projections are performed for that database. We also count the number of conditional databases for the case of just using a prefix tree, which we refer to as SeqBDTreeMiner. As representative data sets, we show the comparison for *DENV1*, *snake*, *PSORTb-ccm*, and *gazelle* data sets in Figure 15. It shows that SeqBDDMiner always projects the smallest number of conditional databases. Moreover, in cases where there is a huge reduction in terms of the database projections, such as in *DENV1* data set, and in other data sets with low support threshold, SeqBDDMiner projects significantly fewer databases than PrefixSpan. When compared to SeqBDTreeMiner, it shows that the caching mechanism in SeqBDDMiner reduces the number of database projections by up to 12 times for all data sets, being more effective when the minimum support threshold is low.

We now examine the hit rate of the cached patterns from each conditional database, by counting the number of conditional databases which are skipped because they exist in the cache table. Figure 16(a) shows that at a support threshold as high as 90% in the highly similar *DENV1* data set, SeqBDDMiner is able to achieve a high hit rate of 53%.

In the *snake* data set which contains short sequences, and *PSORTb-ccm* data set which contains long but dissimilar sequences, SeqBDDMiner achieves a low hit rate except when the support threshold is relatively low (Figure 16(b) and



(a) DENV1 (b) snake

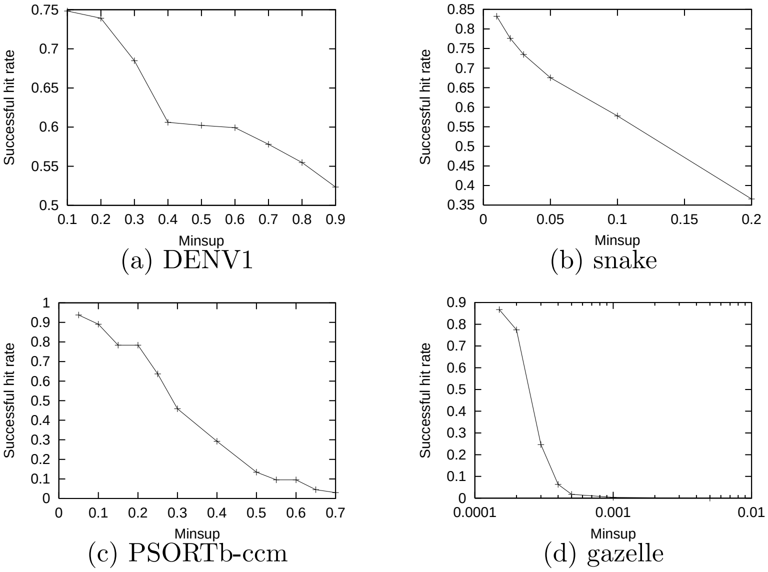
(c) PSORTb-ccm (d) gazelle

**Fig. 15.** Number of projected conditional databases

(c)). This shows that a large amount of node-sharing among the conditional databases at a low support threshold value can still be achieved, even though the sequences are short or dissimilar. In the *gazelle* data set which is highly similar but contains short sequences, the hit rate does not even reach 30% for a support threshold as low as 0.03% (Figure 16(d)), which corresponds to the poorer time performance of SeqBDDMiner, except when the support threshold value is low.

Figure 17 shows the respective mining times using either SeqBDDs, with or without caching, and SeqBDTrees. The caching mechanism is not applicable for SeqBDTrees (prefix trees), since it only has any effect when fan-in is allowed. Firstly, to study the effects of the use of cache in SeqBDDMiner, for all of its intermediate computations, we compare its running times against SeqBDDMiner-noCache in Figure 17. It shows that SeqBDDMiner is at least twice as fast as SeqBDDMiner-noCache for the *PSORTb-ccm* data set, as well as for the *snake* and *gazelle* data sets given low support threshold values, i.e. less than 20% for the *snake* data set, and less than 0.04% for the *gazelle* data set, since the pattern cache has a high successful rate in those circumstances (Figure 16(b)). For the *DENV1* data set, moreover, SeqBDDMiner is 1000 times faster, which is explained by the high successful hit rate of the pattern cache for all support thresholds (shown earlier in Figure 16(a)).

Secondly, to study the effects of node fan-in on the mining time of SeqBDDMiner, we compare the running times of SeqBDDMiner-noCache (with the caching mechanism turned off) and SeqBDTreeMiner. Since both of them are not using the caching mechanism, they differ only in the cost for building SeqBDDs or SeqBDTrees representing the conditional databases and the output patterns. In all scenarios, SeqBDDMiner-noCache is at least 2-3 times faster than SeqBDTreeMiner, and SeqBDTreeMiner could not complete when the support threshold is relatively low, due to excessive memory usage. Though they



(a) DENV1

(b) snake

(c) PSORTb-ccm

(d) gazelle

**Fig. 16.** Hit rate of cached patterns in SeqBDDMiner

project the same number of conditional databases, it shows that building and maintaining SeqBDDs, being more compact, is faster than SeqBDTrees, which do not allow node fan-in.

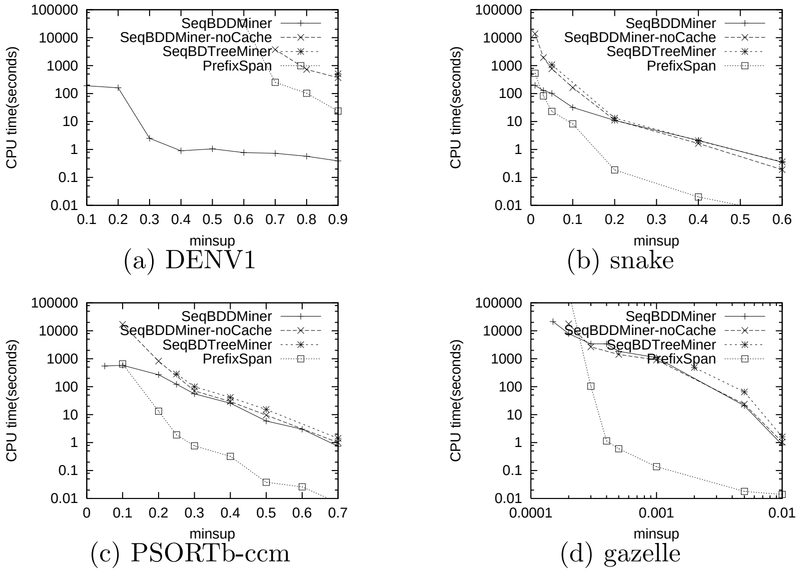
Finally, to study the effects of node fan-out, we compare the running times of SeqBDTreeMiner against PrefixSpan, where a SeqBDTree is similar to a prefix tree which allows node fan-out (but not node fan-in), whilst PrefixSpan is a memory-based algorithm. When there exist many sequences, such as in the *gazelle* data set, the running time of SeqBDTreeMiner grows exponentially slower than PrefixSpan as the minimum support decreases, having a similar running time when the minimum support is less than 0.02% (Figure 17(d)).

## 8. Discussion

In this section, we provide a detailed discussion of the performance of our SeqBDD-based algorithm in terms of the effectiveness of its caching utility which is affected by the amount of node sharing between the databases, which leads us to identify two interesting circumstances according to similarity characteristics of the input data set.

**(Highly) similar sequences:** In a data set which contains (highly) similar sequences, its input SeqBDD has a large amount of node fan-out or node fan-in and the conditional databases are also more likely to share many nodes, especially when the sequences are long.

Our experimental results show that SeqBDDMiner achieves a high hit rate of the pattern cache and the best runtime performance when mining the highly similar DNA or protein sequence data sets, especially at a low support threshold if the sequences are relatively short. Consider the following situation. Suppose $p$ and $q$ ($p \neq q$) are two frequent subsequences. If every sequence which con-



(a) DENV1 (b) snake

(c) PSORTb-ccm (d) gazelle

**Fig. 17.** Mining time comparison between SeqBDDMiner, with or without caching, and SeqBDTreeMiner

tains $p$ also contains $q$, then both conditional databases are identical and the pre-computed patterns can be re-used. Otherwise, the two conditional databases may still share common sub-trees given the input sequences are highly similar. When performing database projections, the databases share a lot of similar computations due to their large degree of node-sharing.

However, if the sequences are very short, the patterns are more likely to be dissimilar and the amount of node-sharing among the conditional databases may not be significant. This is proven by the poor hit rate of the cached patterns when mining the weblog *gazelle* data set. In this circumstance, the construction of the conditional databases is costly, whereas PLWAP or Prefixspan can have a better performance since they do not physically build the conditional databases.

**Dissimilar sequences:** In a data set which contains dissimilar sequences, its input SeqBDD has little node fan-out and node fan-in. In general, being dissimilar, the conditional databases are also dissimilar and caching effectiveness decreases, since not many node re-use is allowed. If the support threshold is low, however, similarity of the frequent subsequences increases and the node-sharing among the conditional databases also increases, as shown by the increased hit rate of the pattern cache in our experiments with the *DNA.Homologene554*, and *DENV2* at a very low support threshold value.

**Summary of results:** In the beginning of this paper we posed three questions which we aim to answer in this paper, as follows:

*Can a BDD be used for compactly representing sequences?* We showed that ZBDDs have a limited data compression ability for representing sequences. In this paper, we have proposed a more suitable type of BDD, namely Sequence BDDs, which allow sequences of various lengths to share nodes representing their common prefixes as well as suffixes, through the sharing of common sub-trees. In our experiments, we showed that a SeqBDD can be half as large compared to a prefix tree representation. Furthermore, we found that the total amount of node sharing across the conditional databases is proportional to the compactness of the initial SeqBDD database.

*Can the use of a SeqBDD benefit frequent subsequence mining?* The key features of our proposed algorithm are SeqBDD's canonical structure and its caching ability. We performed experiments for examining the effects of caching in our SeqBDDMiner, and showed that regardless of the compactness of the initial SeqBDD database, maintaining the canonicity across multiple SeqBDDs is advantageous since many of the intermediate databases do share common sub-trees. Thus, redundancy can be avoided by allowing the same sub-trees to re-use their computation results.

*Can our proposed SeqBDD-based miner outperform state-of-the-art pattern growth techniques in frequent subsequence mining?* When the input sequences are long and similar, SeqBDDMiner outperforms the state-of-the-art pattern growth techniques such as PLWAP and Prefixspan. When the input sequences are short, or dissimilar, SeqBDDMiner is less competitive due to the low node-sharing across the conditional databases, except when the support threshold value is low for which SeqBDDMiner has a higher scalability than the other techniques.

## 9. Related Work

Sequential patterns are useful in a number of applications, such as sequence classification (Exarchos, Tsipouras, Papaloukas and Fotiadis, 2008), and protein localization (Zaiane, Wang, Goebel and Taylor, 2006). Our mining technique is based on the prefix-growth framework which suits prefix-monotone constraints (Pei et al., 2002). Such constraints include the minimum frequency (considered in this paper), minimum length, gap constraint (Ji, Bailey and Dong, 2007), similarity constraint (measured by the longest common subsequences) (Mitasiunaite and Boulicaut, n.d.), and many more. There also exist tough constraints (Pei et al., 2002), which are not prefix-monotone, such as sum or average constraint, and regular expressions. Work in (Pei et al., 2002) showed that the prefix-growth framework, as well as our technique, can be extended to handle such constraints. Apart from the prefix-growth, there also exists a highly scalable approach which is based on a sampling technique (Luo and Chung, 2008). Their technique uses a prefix tree structure to accommodate frequency counting of the candidate patterns. Our proposed data structure, the weighted SeqBDDs, could possibly be adopted by such a technique.

*Apriori, AprioriAll, AprioriSome* (Srikant and Agrawal, 1996) were the first techniques in sequential pattern mining, based on the *apriori* property of frequent subsequences. Subsequently, more efficient techniques were proposed, such as *GSP* (Agrawal, 1996), *PSP* (Masseglia, Cathala and Poncelet, 1998) and *SPADE* (Zaki, 2001). Work in (Pei et al., 2004) shows that PrefixSpan is generally the most efficient and scalable for mining long sequences.

*GSP (Generalized Sequential Pattern)*(Agrawal, 1996) follows the APRIORI, candidate generation-and-test, framework. It generates candidates of frequent $(k+1)$-sequences by performing a join join on the frequent $k$-sequences. Support counting is the major cost in the GSP algorithm, which requires one database scan for each pattern candidate. *PSP* (Prefix Sequential Pattern) (Masseglia et al., 1998) is similar to GSP, except that PSP introduces the use of prefix-tree to perform the procedure. Work in (Lin and Lee, 2005), moreover, extends the

GSP algorithm to find sequential patterns with time constraints, such as time gaps and sliding time windows.

*SPADE*, proposed in (Zaki, 2001), is based on decomposing the pattern lattice into smaller sub-lattices, and the mining task is decomposed into mining in those smaller sub-lattices. For counting support, SPADE uses a *vertical*, instead of a horizontal, data representation . Each item, and consequently, each sequence is represented using its *id* list where each *id* corresponds to an item and the timestamp. The support of a sequence is then obtained by joining the idlist of its items.

Effective and efficient sequential pattern mining is generally tackled in two orthogonal aspects. First, efficient algorithms are developed to enumerate sequential patterns and count the supports. Our approach falls into this category. Second, pruning techniques are developed to narrow down the search space to only the closed sequential patterns. *CloSpan* (Yang, Han and Afshar, 2003) and *BIDE* (Wang and Han, 2004) are extensions of the prefix-growth framework for closed subsequences. Extending our algorithm for mining closed subsequences is also potentially possible, since suffixes of a sequence which have the same frequency may share nodes in the weighted SeqBDD representation. Based on our findings, SeqBDDs allow efficient processing of the conditional databases, which is particularly beneficial when dealing with dense data which contains many patterns. There also exists other work on mining closed subsequences with different constraints, such as top-$k$ closed sequential patterns (Tzvetkov, Yan and Han, 2005) which are based on minimum length constraint, top-$k$ constraint, and closure.

There appears to be little work that considers the use of BDDs for storing and manipulating sequences. Work in (Kurai et al., 2007) addresses the problem of capturing/enumerating all possible n-grams (sequences without gaps), such as in a text file. Their approach is based on the use of ZBDDs and sequence-to-itemset encoding. More discussion about this approach (and some of its limitations) can be found in Section 4. Other BDD-variants exist for analysing sequential events in fault-tree analysis (Sinnamon and Andrews, 1996), but they consider pseudo-sequential events since any event does not occur more than once in each fault-path. There exists a type of unordered BDD, namely the Free BDDs(Gergov and Meinel, 1994), but unlike SeqBDDs, they do not allow a variable to appear multiple times in any path.

The combinatorial pattern matching community has studied the use of subsequence automata (Hirao, Hoshino, Shinohara, Takeda and Arikawa, 2000), which are inspired by Directed Acyclic Subsequence Graphs (DASGs) (Baeza-Yates, 1991), for solving subsequence matching problems. Similar to SeqBDDs, identical sub-trees are merged, but every node in DASGs may have $m$ outgoing edges, where $m$ is the size of the alphabet. Such a technique can be extended for finding frequent subsequences, but it does not have SeqBDDMiner's ability to avoid infrequent candidate generations, and to re-use intermediate computation results which we have shown to be particularly advantageous in our study.

## 10. Future Work and Conclusion

In this paper, we have introduced Weighted Sequence Binary Decision Diagrams for efficient representation of sequences and shown how they may be used as the basis for mining frequent subsequences. A primary objective has been to investigate situations where the use of a Sequence BDD is superior to the prefix tree style approaches. In on our experimental results, we have shown that SeqBDDs can be highly effective in improving the efficiency of frequent subsequence mining, for cases when the input sequences or intermediate computations are similar, the sequences are long, or when the mining is at low support. Based on this evidence, we believe SeqBDDs are an important and worthwhile data structure for sequence data mining.

As future work, it would be interesting to investigate a hybrid style approach that uses a combination of Prefixspan and SeqBDD, according to estimated properties of the input data set. It is also promising to further extend our method to mining closed sequential patterns.

**Acknowledgements.** This paper was partially supported by NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## References

Agrawal, R. (1996), Mining sequential patterns: Generalizations and performance improvements, *in* 'Proc. of the 5th Int'l Conf. on Extending Database Technology (EDBT'96)', pp. 3–17.

Aloul, F. A., Mneimneh, M. N. and Sakallah, K. (2002), ZBDD-based backtrack search SAT solver, *in* 'Int'l Workshop on Logic Synthesis', University of Michigan.

Baeza-Yates, R. A. (1991), 'Searching subsequences', *Theoretical Computer Science* **78**(2), 363–376.

Bryant, R. E. (1986), 'Graph-based algorithms for boolean function manipulation', *IEEE Trans. on Computers* **35**(8), 677–691.

Bryant, R. E. and Chen, Y.-A. (1995), Verification of arithmetic circuits with binary moment diagrams, *in* 'DAC'95: Proc. of the 32nd ACM/IEEE Conf. on Design Automation', pp. 535–541.

Exarchos, T. P., Tsipouras, M. G., Papaloukas, C. and Fotiadis, D. I. (2008), 'An optimized sequential pattern matching methodology for sequence classification', *Knowledge and Information Systems (KAIS)* .

Ezeife, C. I. and Lu, Y. (2005), 'Mining web log sequential patterns with position coded preorder linked WAP-tree', *Int'l Journal of Data Mining and Knowledge Discovery (DMKD)* **10**(1), 5–38.

Ezeife, C. I., Lu, Y. and Liu, Y. (2005), PLWAP sequential mining: open source code, *in* 'OSDM'05: Proc. of the 1st Int'l Workshop on Open Source Data Mining', pp. 26–35.

Ferreira, P. and Azevedo, A. P. (2005), Protein sequence classification through relevant sequences and bayes classifiers, *in* 'Proc. of Progress in Artificial Intelligence', Vol. 3808, pp. 236–247.

Gergov, J. and Meinel, C. (1994), 'Efficient analysis and manipulation of OBDDs can be extended to FBDDs', *IEEE Trans. on Computers* **43**(10), 1197–1209.

Ghoting, A., Buehrer, G., Parthasarathy, S., Kim, D., Nguyen, A., Chen, Y.-K. and Dubey, P. (2005), Cache-conscious frequent pattern mining on a modern processor, *in* 'Proc. of the 31st Int'l Conf. on Very Large Data Bases', pp. 577–588.

Han, J., Pei, J., Yin, Y. and Mao, R. (2004), 'Mining frequent patterns without candidate generation: A frequent-pattern tree approach', *Data Mining and Knowledge Discovery* **8**(1), 53–87.

Hirao, M., Hoshino, H., Shinohara, A., Takeda, M. and Arikawa, S. (2000), 'A practical algorithm to find the best subsequence patterns', *Discovery Science* pp. 141–154.

IBM (n.d.), 'Synthetic data generation code for association rules and sequential patterns. intelligent information systems, IBM almaden research center'.
**URL:** *http://www.almaden.ibm.com/software/quest/resources*

Ji, X., Bailey, J. and Dong, G. (2007), 'Mining minimal distinguishing subsequence patterns with gap constraints', *Knolwedge and Information Systems (KAIS)* **11**(3), 259–286.

Kurai, R., Minato, S. and Zeugmann, T. (2007), N-gram analysis based on Zero-suppressed BDDs, *in* 'New Frontiers in Artificial Intelligence (LNCS 4384)'.

Lin, M.-Y. and Lee, S.-Y. (2005), 'Efficient mining of sequential patterns with time constraints by delimited pattern growth', *Knowledge and Information Systems (KAIS)* **7**(4), 499–514.

Loekito, E. and Bailey, J. (2006), Fast mining of high dimensional expressive contrast patterns using Zero-suppressed Binary Decision Diagrams, *in* 'Proc. of the 12th Int'l Conf. on Knowledge Discovery and Data Mining (KDD)', pp. 307–316.

Loekito, E. and Bailey, J. (2007), Are Zero-suppressed Binary Decision Diagrams good for mining frequent patterns in high dimensional datasets?, *in* 'Proc. of the 6th Australasian Data Mining Conf. (AusDM)', pp. 139–150.

Luo, C. and Chung, S. M. (2008), 'A scalable algorithm for mining maximal frequent sequences using a sample', *Knowledge and Information Systems (KAIS)* **15**(2), 149–179.

Ma, Q., Wang, J., Sasha, D. and Wu, C. (2001), 'DNA sequence classification via an expectation maximization algorithm and neural networks: a case study', *IEEE Trans. on Systems, Man and Cybernetics, Part C* **31**(4), 468–475.

Masseglia, F., Cathala, F. and Poncelet, P. (1998), 'The PSP approach for mining sequential patterns', *Proc. of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery* **1510**, 176–184.

Minato, S. (1993), Zero-suppressed BDDs for set manipulation in combinatorial problems, *in* 'Proc. of the 30th Int'l Conf. on Design Automation', pp. 272–277.

Minato, S. (2001), 'Zero-suppressed BDDs and their applications', *Int'l Journal on Software Tools for Technology Transfer (STTT)* **3**(2), 156–170.

Minato, S. (2005), Finding simple disjoint decompositions in frequent itemset data using Zero-suppressed BDD, *in* 'Proc. of ICDM Workshop on Computational Intelligence in Data Mining', pp. 3–11.

Minato, S. and Arimura, H. (2005), Combinatorial item set analysis based on Zero-suppressed BDDs, *in* 'IEEE Workshop on Web Information Retrieval WIRI', pp. 3–10.

Minato, S. and Arimura, H. (2006), Frequent pattern mining and knowledge indexing based on Zero-suppressed BDDs, *in* 'The 5th Int'l Workshop on Knowledge Discovery in Inductive Databases (KDID'06)', pp. 83–94.

Mitasiunaite, I. and Boulicaut, J.-F. (n.d.), Looking for monotonicity properties of a similarity constraint on sequences, *in* 'Proceedings of the 2006 ACM Symposium on Applied Computing', pp. 546–552.

NCBI (n.d.), 'Entrez, the life sciences search engine'.
**URL:** *http://www.ncbi.nlm.nih.gov/sites/entrez*

Ossowski, J. and Baier, C. (2006), Symbolic reasoning with weighted and normalized decision diagrams, *in* 'Proc. of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning', pp. 35–96.

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U. and Hsu, M.-C. (2004), 'Mining sequential patterns by pattern-growth: The PrefixSpan approach', *IEEE Trans. on Knowledge and Data Engineering* **16**(11), 1424–1440.

Pei, J., Han, J., Mortazavi-asl, B. and Zhu, H. (2000), Mining access patterns efficiently from web logs, *in* 'PAKDD'00: Proc. of the 2000 Pacific-Asia Conf. on Knowledge Discovery and Data Mining', pp. 396–407.

Pei, J., Han, J. and Want, W. (2002), Mining sequential patterns with constraints in large databases, *in* 'Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM)', pp. 18–25.

She, R., Chen, F., Wang, K., Ester, M., Gardy, J. L. and Brinkman, F. S. L. (2003), Frequent-subsequence-based prediction of outer membrane proteins, *in* 'Proc. of the 9th Int'l Conf. on Knowledge Discovery and Data Mining (KDD)', Washington D.C., pp. 436–445.

Sinnamon, R. M. and Andrews, J. (1996), 'Quantitative fault tree analysis using binary decision diagrams', *European Journal of Automation* **13**(8).

Srikant, R. and Agrawal, R. (1996), Mining sequential patterns: generalizations and performance improvements, *in* 'Proc. of the 5th Int'l Conf. on Extending Database Technology: Advances in Database Technology', pp. 3–17.

Tzvetkov, P., Yan, X. and Han, J. (2005), 'Tsp: Mining top-*k* closed sequential patterns', *Knowledge and Information Systems (KAIS)* **7**(4), 438–457.

Wang, J. and Han, J. (2004), BIDE: Efficient mining of frequent closed sequences, *in* 'ICDE'04 Proc. of the 20th Int'l Conf. on Data Engineering', p. 79.

Yang, X., Han, J. and Afshar, R. (2003), Clospan: Mining closed sequential patterns in large databases, *in* ‘Proceedings of the International Conference on Data Mining (SDM)’, pp. 166–177.

Zaiane, O. R., Wang, Y., Goebel, R. and Taylor, G. (2006), ‘Frequent subsequence-based protein localization’, *Data Mining for Biomedical Applications* pp. 35–47.

Zaki, M. J. (2001), ‘SPADE: An efficient algorithm for mining frequent sequences’, *Machine Learning* **42**(1-2), 31–60.

## Author Biographies

**Elsa Loekito** Elsa Loekito received her Bachelor of Computer Science (Hons) and PhD degree in Computer Science from the University of Melbourne, Australia, in 2004 and 2009. She is currently a research assistant in the Department of Computer Science and Software Engineering, University of Melbourne. Her main research interests are pattern mining, contrast pattern mining, classification, and the use of binary decision diagrams for data mining. She has published research papers in the proceedings of a number of international conferences, such as ACM SIGKDD 2006, ACM CIKM 2008, and PAKDD 2009.

**James Bailey** is a Senior Lecturer at the University of Melbourne. He received his PhD from the University of Melbourne in 1998 and BSc and BE from the same University in 1993 and 1994 respectively. His research interests are in data mining and machine learning, bioinformatics, health informatics and database systems. He has published over 80 papers in leading conferences and journals in computing and has received best paper awards at IEEE ICDM in 2005 and IAPR PRIB in 2008.

**Jian Pei** Jian Pei is an Associate Professor and the Director of Collaborative Research and Industry Relations in the School of Computing Science at Simon Fraser University, which he joined in 2004. From 2002 - 2004, he was an Assistant Professor at the State University of New York at Buffalo. He received his Ph.D. degree in Computing Science from Simon Fraser University in 2002. He also received B. Eng. and M. Eng. degrees from Shanghai Jiao Tong University in 1991 and 1993. His research interests can be summarized as developing effective and efficient data analysis techniques for novel data intensive applications. Since 2000, he has published one monograph and over 120 research papers in refereed journals and conferences. He is an associate editor of IEEE Transactions of Knowledge and Data Engineering and a senior member of the ACM and IEEE. He is the recipient of the British Columbia Innovation Council 2005 Young Innovator Award, an NSERC 2008 Discovery Accelerator Supplements Award, an IBM Faculty Award (2006), a KDD Best Application Paper Award (2008), and an IEEE Outstanding Paper Award (2007).

*Correspondence and offprint requests to*: James Bailey, Department of Computer Science and Software Engineering, University of Melbourne, Australia. Email: jbailey@csse.unimelb.edu.au