Suffix-DDs: Substring Indices Based on Sequence BDDs for Constrained Sequence Mining

Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato

Graduate School of Information Science and Technology, Hokkaido University,
N-14 W-9, Sapporo, 060-0814, Japan
{denzumi,arim,minato}@ist.hokudai.ac.jp

Abstract. In this paper, we study an efficient index structure, called *Suffix Decision Diagrams* (SuffixDDs), for knowledge discovery in large sequence data. Recently, Loekito, Bailey, and Pei (KAIS, 2009) proposed a new data structure for sequence data, called Sequence Binary Decision Diagram (SeqBDD), which is an extension of Zero-suppressed Binary Decision Diagrams (ZDDs) for sequences. SuffixDD is a compact substring indices based on SeqBDD for efficiently representing the set of all substrings of a given string. Furthermore, SuffixDD provides a rich collection of operations for sets of sequences inherited from ZDDs and SeqBDDs, which are useful for implementing sequence mining algorithms. Then, we present an efficient algorithm for constructing a SuffixDD for a given text, and then, we show the correctness and the complexity. Furthermore, we present a set of extended operations for manipulating SuffixDDs with application to constrained sequence mining. Finally, we give experimental results on the efficiency of SuffixDD.

Keywords: Frequent substring mining, Sequence mining, Sequence Binary Decision Diagram, SeqBDD, SuffixDD, Suffix tree

1 Introduction

The development of information networks has raised the necessity for data mining to discover useful patterns from massive online data. *Substring indices* are efficient data structures for storing all the substrings of a given text. Suffix trees, Suffix arrays [6], and Directed acyclic word graphs (DAWGs) [3] are such data structures, representing all the substrings contained in a text of length n. Suffix trees and DAWGs can be constructed in $O(n)$ time by the algorithm of [20]. For the last years, these suffix indices have attracted much attention in the field of sequence mining [6, 9, 19, 17, 18]. However, none of existing index structures satisfy the representation and manipulation powers required for these applications. Hence, the study of suffix indices suitable to a wide range of sequence mining is still open issue.

SeqBDD [12] is a new ZDD-based data structure recently proposed by E. Loekito, J. Bailey, and J. Pei [12] . BDDs [2] are widely used for representing and

manipulating Boolean functions inside a computer, and ZDDs [15, 13] are special type of BDDs, which are suitable for handling large-scale sets of combinations. SeqBDDs are string indices based on ZDDs, they are able to represent large sets of sequences compactly, and use a variety of efficient operations.

In this paper, we present a new suffix index based on SeqBDD, called SuffixDD (Suffix Decision Diagram), for storing a huge number of substrings contained in a given input string. The SuffixDD is not only as compact as other suffix indices such as suffix trees [20], suffix arrays [6], and DAWG [3], but allows a rich set of operations for manipulating sets of strings, inherited from ZDD and SeqBDD. These operations are freely combined each other to generate a new sequence mining operators specific to particular application domain. Moreover, structure-sharing and hashing techniques of BDD families help efficient computation on compressed representation. Hence, using SuffixDDs, we can mix mining operators with various constraints on, e.g., the size and the inclusion/exclusion of patterns and data. These features makes SuffixDD a powerful tool for knowledge discovery from large sequence data, especially, constrained sequence mining.

As related works, there are a number of previous researches on manipulation operations on suffix indices, e.g., [1, 7]. However, they mainly focus on efficiency, but not on support for general ad hoc mining queries. In the framework of inductive databases, Lee and De Raedt [11] propose a framework for sequence mining with constraints. Takeda *et al.* [19] applied an extension of DAWG to natural language text mining. Sagot [17] and Shimozono *et al.* [18] study efficient sequence mining built on the top of substring indices. As future works, combination of substring and spatial indices, as in [18], is an interesting problem.

Organization of this paper is as follows. Section 2 gives basic definitions on BDD families. Section 3 introduces an efficient substring indices SuffixDDs. Then, we present two construction algorithms for SuffixDDs, one being a simple technique and the other a more efficient implementation. In Section 4, some computational experiments are reported. Section 5 concludes the paper.

2 Preliminaries

This section introduces BDD families in essentially self-contained manner, assuming no prior knowledge of BDDs according to [4, 12, 13]

2.1 Basic definitions on strings

Let Σ be a finite alphabet and Σ^* be the set of all strings over Σ. We denote the length of string $x \in \Sigma^*$ by $|x|$. The string whose length is 0 is denoted by ϵ and called the *empty string*, that is $|\epsilon| = 0$. The concatenation of two strings x_1 and $x_2 \in \Sigma^*$ is denoted by $x_1 \cdot x_2$, also written simply as x_1x_2 if no confusion occurs.

Strings x, y, and z are said to be the *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively. The ith symbol of a string w is denoted by $w[i]$, and the substring of w that begins at position i and ends at position j is denoted

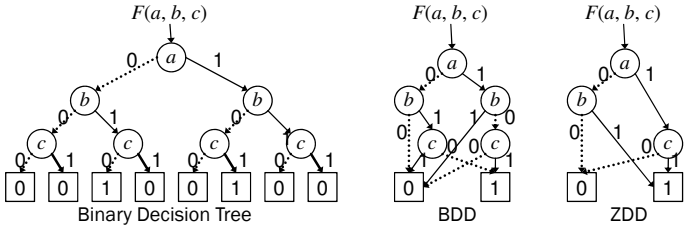

Fig. 1. Binary Decision Tree, BDDs and ZDDs

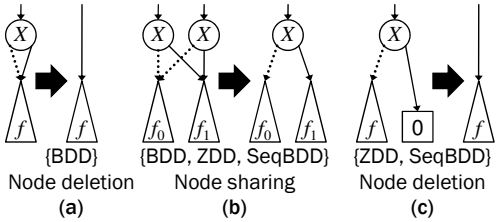

Fig. 2. Reduction rules

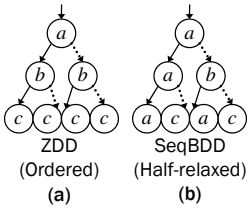

Fig. 3. Ordering rule

by $w[i \dots j]$. For convenience, we let $w[i \dots j] = \epsilon$ for $j < i$. We also denote by $Prefix(w)$ the set of all prefixes of the string $w \in \Sigma^*$, and denote by $Substr(w)$ the set of all substrings of $w \in \Sigma^*$.

2.2 BDDs

A BDD [4] is a directed-graph representation of a Boolean function, as illustrated in Fig. 1for $F(a, b, c) = a\bar{b}c \vee \bar{a}b\bar{c}$. It is derived by reducing the binary tree graph representing a recursive *Shannon's expansion*.The following reduction rules yield a BDD, which can efficiently represent the Boolean function (see [8] for details).

- Share all equivalent subgraphs (see Fig. 2(a)).
- Delete all redundant nodes whose two edges point to the same node (see Fig. 2(b)).

BDDs provide canonical forms for Boolean functions when the variable order is fixed. Most research on BDDs is based on the above reduction rules. [1]

A set of multiple BDDs can be shared with each other under the same variable ordering. In this way, we can handle a number of Boolean functions simultaneously in the one memory space.

Using Bryant's algorithm [4], we can efficiently construct a BDD for the result of a binary logic operation (i.e. AND, OR, XOR), given a pair of operand

[1] In general, the term BDD includes graphs that do not fix the order of variables or that are not canonical, but we deal only with those graphs that follow the above reduction rules in this paper.

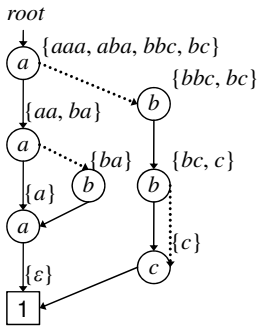

Fig. 4. An example of SeqBDD

BDDs. This algorithm is based on hash-table techniques, and the computation time is almost linear with data size, unless the data overflows the main memory (see [14] for details).

2.3 ZDDs

A ZDD [15, 13] is a special type of BDD for the efficient manipulation of sets of combinations [16].

With a ZDD, we do not delete the nodes whose two edges point to the same node, which was required under the original rule. Instead, we delete all nodes whose 1-edge points directly to the 0-terminal node, and jump through to the 0-edge's destination as shown in Fig. 2 (c).

When no equivalent nodes exist in a ZDD, which is the worst case, the ZDD structure explicitly stores all combinations of all items, in addition to using an explicit linear linked-list data structure. Note that the order of ZDD size never exceeds that of the explicit representation.

It is known that binary operations can be executed efficiently in almost linear computation time and in a single memory space matching the number of nodes of the ZDD. This feature depends on using a technique that avoids redundant computation by using a hash table to store and make available prior computation results.

2.4 SeqBDDs

A SeqBDD is a ZDD that has removed the ordering constraint only for 1-edges with ordered 0-edges, as shown in Fig. 3, and for which a letter is allowed to occur multiple times in a path. SeqBDD semantics are such that a path in a SeqBDD represents a string, for which the nodes are arranged in order of the positions of their respective variables in the string. More specifically, the top node corresponds to the head of the string, and the successive nodes take 1-edges corresponding to the following letters, respectively.

Table 1. Primitive SeqBDD operations

"$\emptyset$"	Returns empty set. (0-terminal node)
"1"	Returns the set of only the empty string. (1-terminal node)
P.top	Returns the letter at the root node of P.
P.onset(x)	Selects the subset of sequences that begin with letter x, and then removes x from the head of each sequence.
P.offset(x)	Selects the subset of sequences that do not begin with letter x.
P.push(x)	Appends x to the head of every sequence in P.
$P \cup Q$	Returns the union set.
$P \cap Q$	Returns the intersection set.
$P \backslash Q$	Returns the difference set (in P but not in Q).
P.count	Counts the number of sequences.

We define the procedure Getnode(x, P_0, P_1), which generates (or makes a reference to) a node with the letter x and two subgraphs P_0, P_1 as 0-child and 1-child respectively. The letter of node P has a lower order (appears earlier in the variable ordering) than the letter of P_0. We denote the total number of descendant nodes of node P, including P itself, by $|P|$. In SeqBDD, a 0-terminal node encodes the empty set $\emptyset$, and a 1-terminal node encodes the set $\{\epsilon\}$ of empty string. The set of sequences that Getnode(x, P_0, P_1) represents the union of the set P_0 represents and the set of sequences appended x to the head of every sequences P_1 represents. Fig. 4 shows an example SeqBDD for a set $S = \{aaa, aba, bbc, bc\}$ of strings on $\Sigma = \{a, b, c\}$, where the set by each node indicates the corresponding set of sequences.. For clarity, we omit the 0-terminal nodes from the illustrations in this paper. (A solid line represents a 1-edge and a dotted line represents a 0-edge). A 1-terminal node is considered an internal node with a bigger letter than all letters in the alphabet.

Table 1 shows most of the primitive operations for SeqBDDs. In these operations, $\emptyset$, 1, *P.top*, and *P.push*(x) are executed in constant time, P.onset(x) and P.offset(x) need linear time to the alphabet size, with the others being almost linear with the size of the graph. We can describe a variety of processing operations on sets of combinations by composition of these primitive operations.

These are natural extensions to ZDDs, as used in Loekito *et.al*'s paper [12]. The size of the output is at most $|P||Q|$ because this is the number of distinctly different calls of binary operations that can arise. To keep a lid on the computation, we can remember what we have done before by hashing. Previously solved cases thereby become terminal ones. Therefore, the running time will be $O(|P||Q|)$ in the worst possible case. However, the running time is practically linear with the SeqBDD representation for almost all operations.

First, we show the SeqBDDs' union operation $\cup$ in Fig. 5. An example is shown in Fig. 6. Recursions always terminate when a sufficiently simple case arises. The terminal cases are $P \cup \emptyset = P$, $\emptyset \cup Q = Q$, and $P \cup Q = P$ when $P = Q$.

```
Operation: P ∪ Q
Input: P, Q: SeqBDD;
 1: if (P > Q) return Q ∪ P;
 2: if (P = ∅) return P;
 3: if (Q = ∅ or P = Q) return P;
 4: R ← cache["P ∪ Q"];if (R exists) return R;
 5: x ← P.top; y ← Q.top;
 6: P_0 ← P.offset(x); P_1 ← P.onset(x);
 7: Q_0 ← Q.offset(y); Q_1 ← Q.onset(y);
 8: if (x < y) R ← Getnode(x, P_0 ∪ Q, P_1);
 9: if (x > y) R ← Getnode(y, P ∪ Q_0, Q_1);
10: if (x = y) R ← Getnode(x, P_0 ∪ Q_0, P_1 ∪ Q_1);
11: cache["P ∪ Q"] ← R;
12: return R;
```

Fig. 5. Code for "$P \cup Q$"

If P is a 0-terminal, i.e. P is empty, the output comprises all sequences in Q. Similarly, if Q is a 0-terminal, P is returned as output.

If P equals Q, i.e. P and Q comprise the same sequences, then the output also comprises all sequences in P.

If both P and Q are internal nodes, we compare the letters of both x and y. Suppose $x = y$. Then, because x is the smallest letter among the head letters of all strings in P and Q, x should be the letter of the output node. The 1-child of the output node should contain all sequences that begin with x, with the 0-child of the output node containing the remaining sequences. Suppose x is smaller than y. Then x is smaller than the head of all sequences in Q. Therefore, the output node should have letter x, and all sequences that begin with x are present in P_1. The remaining output sequences exist in P_0 and Q. Suppose x is bigger than y. This condition is the opposite of the above condition, since the operation is commutative.

When we construct a substring index in SeqBDD, we can consider a 1-terminal node as a 0-terminal node, since no 0-terminal node appears when computing $\cup$. In this way, the $\cup$ operation can be twice as fast. In addition, we can define the operations $\cap$ and $\setminus$, which compute the intersection and difference sets, respectively, by recursive algorithms. These operations are almost the same as for $\cup$. The only differences are that we change the output in the terminal cases and the inputs of recursive calls. The remaining algorithms proceed exactly as for $\cup$.

3 SuffixDD: Definition and Algorithms

In this section, we introduce the *SuffixDD* index structure, which is a compact substring index based on SeqBDD. Then, we present two construction algorithms

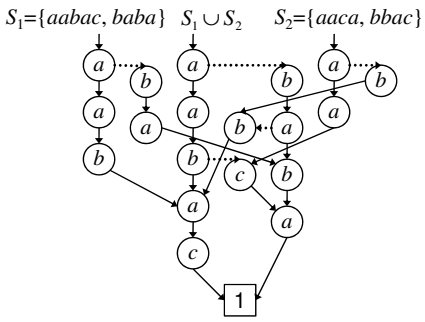

Fig. 6. An example of SeqBDD showing the result of a ∪ operation and its inputs.

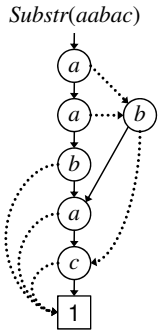

Fig. 7. Structure for a SuffixDD, given an input *aabac*

for a SuffixDD, one construct a SuffixDD from a given input string in $O(|\Sigma|n^3)$ time and the other in $O(|\Sigma|n^2)$ time using only standard ZDD/SeqBDD operations. This gives a great deal of flexibility to our sequence mining framework based on SeqBDD/SuffixDD since every manipulation can be done in this framework.

3.1 SuffixDD

A *Substring index* for a text T is any data structure S that stores the set $Substr(T)$ of all substrings of T, which has the following operations:

Index construction: $SuffixDD(T)$ returns the index S that represents the set of all substrings of $T \in \Sigma^*$.

Membership decision: $S.member(w)$, given a string $p \in \Sigma^*$, returns "yes" if p is a member of S, i.e., is a substring in S, and "no" otherwise.

The *substring index construction problem* is stated as follows: given an input text $T = T[1 \dots n]$, construct the substring index S for T. Let $T = T[1 \dots n] \in \Sigma^*$ be a text of length $n \geq 0$. Then the *SuffixDD* (Suffix Decision Diagram) of text T is a SeqBDD constructed for the set of all substrings $Substr(T)$. Fig. 7 shows an example of SuffixDD for a string $T = aabac$.

It is possible to show that SuffixDD is isomorphic to DAWG (Directed Acyclic Word Graph) [3] with some modifications. From this fact, we can infer that the number of nodes of SuffixDD is $O(n)$ for a text of length $n \geq 0$. The number of nodes of SuffixDD is linear with respect to the length of the text T.

3.2 Naive construction method

Fig. 8 shows a straightforward construction algorithm BuildNaive for SuffixDD. Given a text $T = T[1 \dots n]$, we first build a SeqBDD S_i that represents only the string $T_i = T[1 \dots i]$ at each step. Then we update SuffixDD R_{i-1} by adding all

```
Procedure BuildNaive
Input: Text string T = T[1 ··· n]
 1: P ← ε;  R ← 1;
 2: for (i ← 1, ..., n) {
 3:    P ← P · T[i];
 4:    Build SeqBDD Q for the prefix P = T[1 ... i];
 5:    while (R ≠ (R ∪ Q)) {
 6:       R ← R ∪ Q;
 7:       Q ← Q.onset(Q.top);
 8:    }
 9: } //for
10: return  R;
```

Fig. 8. A naive algorithm for construction of a SuffixDD

```
Procedure BuildFast
Input: Text string T[1 ··· n]
1: S ← 1;
2: R ← 1;
3: for(i ← n, ..., 1){
4:    S ← Getnode(T[i], 1, S);
5:    R ← R ∪ S
6: }
7: return R;
```

Fig. 9. A faster algorithm for constructing a SuffixDD

those suffixes of T_i that are not contained in R_{i-1} at the time, and the result is set as the new SuffixDD R_i. In this algorithm, a SeqBDD must be built at every reading of a new letter, to represent all suffixes of the current text. Since any substring is a suffix of some prefix of T, the correctness of the algorithm is obvious.

For every letter, this algorithm has to build a rectilinear SeqBDD whose length is the same as the text already read, and it computes unions at almost the same time as for the length. Since the computation time of the union $\cup$ is $O(|\Sigma|n)$ in this case, and it is repeated $O(n^2)$ times, we have the following lemma.

Lemma 1. *The running time of* BuildNaive *is bounded by* $O(|\Sigma|n^3)$.

3.3 Efficient construction method

We now give a faster algorithm BuildFast that constructs SuffixDD in $O(|\Sigma|n^2)$ time by using only standard SeqBDD operations as in the naive algorithm. It

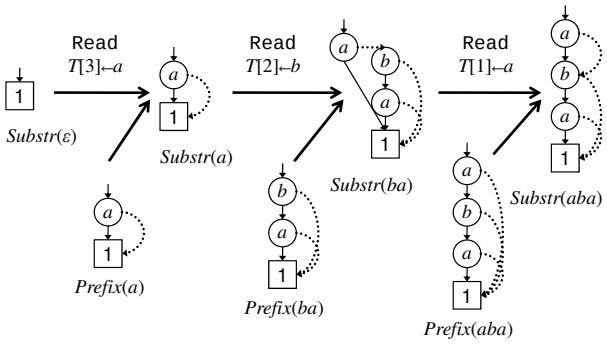

Fig. 10. Processing of *aba* in BuildFast

does not involve redundant bottom-up building or repeating the same union-operation computations.

Fig. 9 shows the algorithm BuildFast. Scanning T from right to left, we maintain and update a SeqBDD S that represents all prefixes of a suffix $T[i \dots n]$ at every state $i = n, \dots, 1$ by appending one node to the old S. To update SuffixDD R, we compute the union of $R \cup S$ once per iteration. In Fig. 10, we show an example of construction for $T = aba$.

Theorem 1. *The algorithm* BuildFast *in Fig. 9 computes the SuffixDD for an input text T of length n in $O(|\Sigma|n^2)$ time.*

Proof. (1) The correctness: We prove by induction on iteration $i = n+1, n, \dots, 1$ that S correctly represents $Substr(T)$. (i) Base case: When $i = n+1$, i.e., before the for loop, the claim immediately holds. We omit the details. (ii) Induction case: Suppose that R is the SuffixDD for the last text $T[i+1 \dots n]$ and that S represents $Prefix(T[i+1 \dots n])$. Then, it is not hard to show that updated S represents $Prefix([i \dots n])$ in line 4. For the set $Substr(T[i \dots n])$, we have the recurrence: $Substr(T[i \dots n]) = Substr(T[i+1 \dots n]) \cup Prefix(T[i \dots n])$. From this recurrence, the computation at line 5 updates R so that it represents $Substr(T[i \dots n])$. (2) Running time: This algorithm makes a new node in $O(1)$ time in one iteration, and computes the union of the existing SuffixDD for $T[i+1 \dots n]$ and the SeqBDD for $Prefix(T[i \dots n])$ in $O(|\Sigma|n)$ time. Since the number of iteration is at most n, the claimed time complexity follows. □

3.4 Extended operations for constrained sequence mining

In this subsection, we give a set of extended operations useful for constrained sequence mining that can be efficiently implemented on SuffixDD. We implemented these operations on our SeqBDD and SuffixDD systems written in the Erlang language [10]. The implementation of some of the operators are similar to techniques in [5, 6] and their details will be described in full paper.

In the followings, T is any input text on Σ, P and Q are any SeqBDDs, $S = \mathit{SuffixDD}(T)$ be the SuffixDD for T, and $N = |P|$ be the number of nodes in P. Let us denote by $p \in \Sigma$ a short string of length m. For an input SuffixDD S, we denote the invocation an operation op of arity $k \geq 0$ with k operands $\alpha_1, \ldots, \alpha_k$ by $T = S.\mathrm{op}(\alpha_1, \ldots, \alpha_k)$, where T is the output SuffixDD/SeqBDD obtained by this operation.

Problem of the membership: $P.\mathrm{member}(p)$, given a string $p \in \Sigma^*$, returns "yes" if p is a member of S, i.e., is a substring in S, and "no" otherwise. The time complexity is $O(|\Sigma|m)$.

Problem of set operations: $P \cup Q$, $P \cap Q$, and $P \backslash Q$ return the union, the intersection, and the difference of P and Q, respectively. The time complexity is $O(|P| \cdot |Q|)$ in the worst case and linear to the output size.

Problem of the count: $P.\mathrm{count}()$ returns the number of all substrings s in S. The time complexity is $O(N)$.

Problem of the matching to substring and subsequence: $P.\mathrm{substr}(p)$ ($P.\mathrm{subseq}(p)$, resp.), given a string $p \in \Sigma^*$, finds all substrings s in S that p is included in s as a contiguous substring (a noncontiguous subsequence, resp.). The time complexity is $O(Nm)$.

Problem of minimum and maximum length constraints: $P.\mathrm{longer}(\ell)$, given a nonnegative integer $\ell \geq 0$, called the *minimum length* (the *maximum length*), finds all substrings s in S whose length $|s|$ is larger or equal to ℓ (smaller or equal to ℓ, resp.), i.e., $|s| \geq \ell$ ($|s| \leq \ell$, resp.). The time complexity is $O(N)$.

Problem of the longest and shortest substrings: $P.\mathrm{longest}(\ell)$ ($P.\mathrm{shortest}(\ell)$) find the longest substring (shortest substring, resp.) s in S. The time complexity is $O(N)$.

Problem of frequent substrings: $S.\mathrm{freqstr}(\sigma)$, given a nonnegative integer $\sigma \geq 0$, called the *minimum frequency threshold*, finds all substrings s in $S = \mathit{SuffixDD}(T)$ that appear at least σ times in a text T with an end marker not in Σ. The time complexity is $O(N)$ time.

Most of the operations shown above can be freely combined as long as their argument types, indicated by P, Q, S, p, ℓ, and σ, are compatible. Applying the extended operators shown above, we can solve several sequence mining problems in compressed form by using SuffixDDs as follows, where both of the input and the output are represented in SuffixDDs. Let $\mathit{Gen}(|T|)$ be the time complexity of constructing $\mathit{SuffixDD}(T)$.

The longest repeated substring problem: Given an input string T with a special end marker, find the longest substring in T that appears at least twice in T. This problem can be solved in $O(\mathit{Gen}(|T|) + |T|)$ by the following code:

$P \leftarrow \mathit{SuffixDD}(T)$; $Q \leftarrow P.\mathrm{freqstr}(2)$;
return $Q.\mathrm{longest}()$;

The longest common substring problem: Given a pair T_1 and T_2 of input strings, find the longest string that appears both of T_1 and T_2 as their substrings. This can be solved in $O(Gen(|T_1|+|T_2|)+|T_1|+|T_2|)$ time by the following code.

$$P \leftarrow \mathit{SuffixDD}(T_1);\ Q \leftarrow \mathit{SuffixDD}(T_2);$$
$$\textbf{return}\ (P \cap Q).\mathrm{longest}();$$

This problem can be generalized for any number m of input strings.

The frequent substring problem with maximum length and subsequence constraints: Given a set $S \subseteq \Sigma^*$ of input strings with mutually distinct special end markers, find all substrings s in S that appears at least σ times in S that has length no more than ℓ and contains a, b, c as a subsequence in this order in $O(Gen(||S||) + ||S||)$ time.

$$P \leftarrow \emptyset;\ \textbf{for}\ (s \in S)\ \textbf{do}\ P \leftarrow P \cup \mathit{SuffixDD}(s);$$
$$\textbf{return}\ P.\mathrm{freqstr}(\sigma).\mathrm{subseq}(\text{"}a, b, c\text{"}).\mathrm{shorter}(\ell);$$

The characteristic substring problem: Given a pair S_+ and S_- of sets of *positive* and *negative* strings, respectively, find all *consistent substrings*, that is, those strings that appears every member of S_+ and appears no member of S_-. This problem can be solved in $O(Gen(||S_+|| + ||S_-||) + ||S_+|| + ||S_-||)$ time by the following code.

$$P \leftarrow \emptyset;\ Q \leftarrow \emptyset;$$
$$\textbf{for}\ (s \in S_+)\ \textbf{do}\ P \leftarrow P \cup \mathit{SuffixDD}(s);$$
$$\textbf{for}\ (t \in S_-)\ \textbf{do}\ Q \leftarrow Q \cup \mathit{SuffixDD}(t);$$
$$\textbf{return}\ P \backslash Q;$$

At Experiment 6 in the next section, we will show an example of ad hoc queries on SuffixDD for solving a sequence mining problem.

4 Experiments

4.1 Setting

We implemented the proposed algorithm in *Erlang* language [10], which is a functional and concurrent programming language, and then performed experiments using real and artificial data. We used the *Erlang Term Storage table* for storing the nodes of the BDDs. The computer was a 2.67 GHz Corei7 PC running Windows XP SP3, with a 3.25 GB main memory, of which about 1.5 GB was allocated to Erlang.

For data sets, in Section 4.2 and 4.4, we used a data set consisting of six English text files from paper1 to paper6 taken from Calgary corpus [2]. In Section 4.3, we used artificial data, in which all letters occurred with equal likelihood, as the test data. The text strings were generated randomly by uniform distribution on Σ. The alphabet size $|\Sigma|$ was either 4 or 128, depending on the experiment.

[2] `http://corpus.canterbury.ac.nz/resources/calgary.tar.gz`

Table 2. Results of Experiment 1

File	File size (B)	SuffixDD size	#Substrings	#letters	Time (ms)
paper1	53,161	102,025	1.41×10^9	2.50×10^{13}	25,323
paper2	82,199	157,398	3.38×10^9	9.26×10^{13}	43,391
paper3	46,526	89,941	1.08×10^9	1.68×10^{13}	22,344
paper4	13,286	26,078	88,196,012	3.91×10^{11}	4,443
paper5	11,954	23,243	71,392,689	2.85×10^{11}	4,297
paper6	38,105	73,989	725,674,256	9.22×10^{12}	16,261
Sum	245,231	472,674	6.76×10^9	1.44×10^{14}	–
Union	–	470,534	6.76×10^9	1.44×10^{14}	2,079
Intersection	–	2,397	5,280	24,409	521

4.2 The compactness of SuffixDD index

Experiment 1: Table 2 shows the results on the sizes of input and output SuffixDDs for the union and the intersection operations. The first seven rows and the last two rows, respectively, correspond to the input SuffixDD for text files and the output SeqBDD for the results of the operations $\cup$ and $\cap$. From this result, we can see that the SuffixDD can efficiently store and manipulate a huge number of substrings on main memory. For instance, for the size of SuffixDD for paper1 is around 50KB, at most twice in the number of input letters, while the number and the total size of all substrings stored are 1.41 mega strings and 25 tera bytes, respectively, which are too huge to explicitly stored and manipulated. Fig. 11 shows the running time of set operations $\cup, \cap$, and $\setminus$ for SuffixDDs, which looks almost linear to the input sizes.

4.3 The performance of SuffixDD construction

Both SuffixDD construction algorithms, BuildNaive and BuildFast, were implemented for comparison. The experiments were performed to test the data structure of SuffixDD, using each random text of length n with a default Σ of $\{A, B, C, D\}$. The following results were obtained.

Experiment 2: Fig. 13 shows the running time for BuildFast. Fig. 12 shows the running time for both methods. These results suggest that the naive algorithm runs in $O(n^2)$ time and the efficient algorithm runs in $O(n)$ time. These results do not contradict Lemma 1 or Theorem 1, running in better time than predicted. The efficient method built SuffixDD faster than the $O(n^2)$ running time derived from analysis of the algorithm. The memorization technique worked well, which was probably the reason for the overall running time being better than expected.

Experiment 3: Fig. 14 and Fig. 15 show the construction time and the size of the SuffixDD by BuildFast, respectively, for two random texts for which $|\Sigma| = 4$

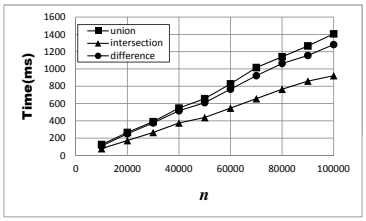

Fig. 11. Experiment 1: Computation times for set operations on two SuffixDDs

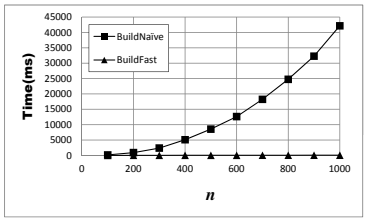

Fig. 12. Experiment 2: Running time of BuildNaive and BuildFast against input size

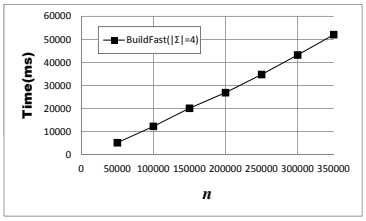

Fig. 13. Experiment 2: Running time of BuildFast against input size

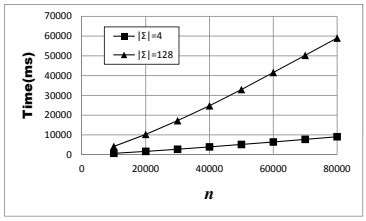

Fig. 14. Experiment 3: Running time of BuildFast for $|\Sigma| = 4$ and $|\Sigma| = 128$

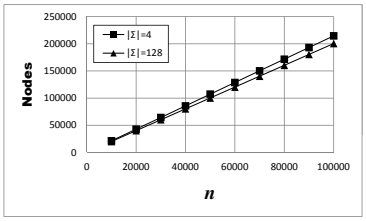

Fig. 15. Experiment 3: Size of SuffixDDs, for $|\Sigma| = 4$ and $|\Sigma| = 128$

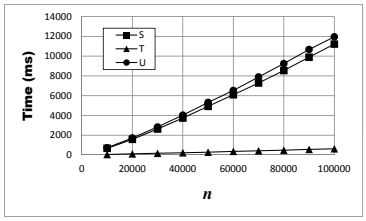

Fig. 16. Experiment 4: The running time of BuildFast, for texts with biased probabilities

and $|\Sigma| = 128$. In Fig. 14, running times seem to be proportional to n in both cases, while the algorithm with $|\Sigma| = 4$ were nearly six times faster than that for $|\Sigma| = 128$ due to the long traverse of 0-edges in the large alphabet case. In Fig. 15, the size of the SuffixDD seems $O(n)$ in the text size as indicated by theory. The effect of the alphabet size on the SuffixDD size seems small from the experiment. .

Experiment 4: We generated three random texts, namely S, T, and U. Text S had $P(\mathrm{A}) = 0.4$ and $P(\mathrm{B}) = P(\mathrm{C}) = P(\mathrm{D}) = 0.2$, where $P(x)$ is the probability that letter x is chosen. Text T had $P(A) = 1$. Text U had equally likely probabilities. Fig. 16 shows the running time for the SuffixDDs of S, T, and U. From the figure, we can see that both the running time and the size are smaller

```
Eshell V5.7.3  (abort with ^G)
1> seqbdd:set_table().                      ==> ok
2> S1 = sdd:suffixdd(seq:read("paper1")).  ==>  4379855
3> S2 = sdd:suffixdd(seq:read("paper2")).  ==>  11555546
4> S3 = sdd:suffixdd(seq:read("paper3")).  ==>  15431702
5> S4 = sdd:suffixdd(seq:read("paper4")).  ==>  16299568
6> S5 = sdd:suffixdd(seq:read("paper5")).  ==>  17134036
7> S6 = sdd:suffixdd(seq:read("paper6")).  ==>  20018804
8> I = sdd:intersect(sdd:intersect(S1,S2),S3).  ==>  20042241
9> U = sdd:union(sdd:union(S4,S5),S6).     ==>  20089690
10> D = sdd:difference(I, U).              ==>  20094751
11> sdd:longest(D).  ==>
   "\n.sp2\n.ce4\nDepartment of Computer Science\nThe University
   of Calgary\n2500 University Drive NW\nCalgary, Canada T2N 1N4
   \n.sp2\n."
```

Table 3. A demonstration of series of ad hoc queries on an implementation of SuffixDDs in the Erlang language. The above sequence of queries finds the longest sequence common to a set of positive sequences (S1, S2, and S3), but appearing in none of a set of negative sequences (S4, S5, and S6), where a number and a string to the right of the arrow `==>` indicate a node-id and the longest string as return values.

for the text with stronger bias. The SuffixDDs for T were built 20 times faster for biased text than balanced texts.

4.4 Demonstration of ad hoc queries with a SuffixDD

Experiment 6: Ad hoc queries In Fig. 3, we show an example execution of a series of ad hoc queries that is given by a combination of basic operations given in the previous section.

5 Conclusion

In this paper, we proposed a suffix index, called the SuffixDD for storing a huge number of substrings, while allowing a rich collection of manipulation operations. We then presented an efficient algorithm for constructing SuffixDD using only primitive SeqBDD operations. Experimental results show the efficiency of our construction algorithm and the usefulness of our sequence mining framework based on SuffixDD. These results show that SuffixDD is a powerful framework for data mining and knowledge discovery from large sequence data.

References

1. Abouelhoda, M. I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms, vol.2 no.1, pp.53-86 (2004)
2. Akers, S.B.: Binary decision diagrams. IEEE. Trans. Comput., vol.C-27, no.6, pp.509–516 (1978)
3. Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnell, R.: Linear size finite automata for the set of all subwords of a word: an outline of results, Bull. Europ. Assoc. Theoret. Comput. Sci., 21, pp.12–20 (1983)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation, IEEE. Trans. Comput., vol.C-35, no.8, pp.677–691 (1986)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, Cambridge (2007)
6. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge (1997)
7. Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications, Proc. CPM'01, LNCS 2089, Springer, pp.181–192 (2001)
8. Knuth, D.E.: The Art of Computer Programming, vol.4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams, Addison-Wesley (2009)
9. Kurai, R., Minato, S., Zeugmann, T.: N-Gram analysis based on Zero-suppressed BDDs, New Frontiers in Artificial Intelligence, LNCS 4384, pp.289–300. (2006)
10. Larson, J.: Erlang for Concurrent Programming, Communications of the ACM, vol.52, no.3, pp.48–56, ACM, (2009)
11. Lee, S. D., De Raedt, L.: An efficient algorithm for mining string databases under constraints, KDID2004, LNCS 3377, pp.108–129 (2004)
12. Loekito, E., Bailey, J., Pei, J.: A Binary decision diagram based approach for mining frequent subsequences, Knowl. Inf. Syst., (2009)
13. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems, Proc. 30th ACM/IEEE Design Automation Conf., pp.272–277 (1993)
14. Minato, S.: Binary decision diagrams and applications for VLSI CAD, Kluwer Academic Publishers (1996)
15. Minato, S.: Zero-suppressed BDDs and their applications, Int. J. Software Tools for Technology Transfer (STTT), vol.3, no.2, pp.156–170, Springer (2001)
16. Minato, S., Arimura, H.: Frequent pattern mining and knowledge indexing based on Zero-suppressed BDDs, 5th International Workshop on Knowledge Discovery in Inductive Databases (KDID2006). LNCS 4747, pp.152–169. Springer (2006)
17. Sagot, M.F.: Spelling approximate repeated or common motifs using a suffix tree, Proc. Third Latin American Symposium on Theoretical Informatics, LNCS 1380, pp.374–390, Springer (1998)
18. Shimozono, S., Arimura, H., Arikawa, S.: Efficient discovery of optimal word-association patterns in large text databases, New Generation Computing, vol.18, pp.49–60, 2000.
19. Takeda, T., Matsumoto, T., Fukuda, T., Nanri, I.: Discovering characteristic expressions from literary works: a new text analysis method beyond n-gram statistics and KWIC, Proc. DS 2000, LNCS 1967, pp.112–126. Springer (2000)
20. Ukkonen, E.: On-line construction of suffix trees, Algorithmica, vol.14, no.3, pp.249–260 (1995)