𝕋ℂ𝕊-𝕋ℝ-𝔸-**10-42**

𝕋ℂ𝕊 Technical Report

Substring Indices Using Sequence BDDs

by

Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato

Division of Computer Science

Report Series A

April 14, 2010

Hokkaido University
Graduate School of
Information Science and Technology

Email:	minato@ist.hokudai.ac.jp	Phone:	+81-011-706-7682
		Fax:	+81-011-706-7682

Building Substring Indices Using Sequence BDDs

Shuhei Denzumi
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

Hiroki Arimura
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

Shin-ichi Minato
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

April 14, 2010

(Abstract) There is a demand for efficient indexed-substring data structures, which can store all substrings of a given text. Suffix trees and *Directed Acyclic Word Graphs* (DAWGs) are examples of substring indices, but they lack operations for manipulating sets of strings. The *Sequence Binary Decision Diagram* (SeqBDD) data structure proposed by E. Loekito, J. Bailey, and J. Pei (KAIS, 2009) is a new type of *Binary Decision Diagram* (BDD), and represents sets of sequences. This study focuses mainly on two issues: (a) compact substring indices based on SeqBDD, called *Suffix Decision Diagrams* (SuffixDDs), which make it possible to represent the set of all substrings efficiently via various operations inherited from *Zero-suppressed Binary Decision Diagrams* (ZDDs), and (b) methods for building SuffixDD, beginning with an empty string and updating iteratively whenever a new letter is read. This paper presents an efficient algorithm for constructing a SuffixDD for a given text, together with a proof of correctness and some notes about BDD families, and discusses why the new data structure appears to have advantages over existing substring indices. An upper bound on the running time is also obtained. It is hoped that presenting this data structure to a wider audience at this time will help to promote useful discussion of the important issues.

1 Introduction

The development of information networks has raised the necessity for data mining to discover usage patterns from massive online data. For a given text T and pattern P, string matching is the problem of determining if P occurs in T. This problem was well known long ago, and is as important today for manipulating large-scale data such as Internet-based data.

Substring indices are efficient data structures for storing all the substrings of a given text. DAWG [2] is such a data structure, representing all the substrings contained in a text of length n. For a pattern of length m, DAWG can solve the problem in $O(m \log |\Sigma| + occ)$ time, where $|\Sigma|$ and occ denote the alphabet size and the number of occurrences of P in T, respectively. We can obtain DAWG in $O(n)$ time by simulating Ukkonen's online algorithm for constructing a suffix tree [12].

In this paper, we present a method for applying SeqBDDs to substring indices. SeqBDDs [7] is a new ZDD-based data structure recently proposed by E. Loekito, J. Bailey, and J. Pei. This data structure is a new type of BDD. BDDs [3] are widely used for representing and manipulating Boolean functions inside a computer, and ZDDs [8, 10], a special type of BDDs, are suitable for handling large-scale sets of combinations. SeqBDDs are string indices based on ZDDs, they are able to enumerate large sets of sequences implicitly, and use a variety of efficient operations. Families of BDDs are compact representations of discrete structures, with rich collections of operations. The extension of SeqBDD to substring indices is an open issue, and we would like to devise an efficient substring index structure based on SeqBDD.

In the remainder of this paper, Chapter 2 formulates a convenient notational framework and discusses BDD families. This treatment is essentially self-contained, assuming no prior knowledge of BDDs. Chapter 3 presents two construction procedures for SuffixDD, one being a simple technique and the other a more efficient implementation, together with some theorems that describe the virtues of the method. Some computational experiments are recorded in Chapter 4, with Chapter 5 concluding the paper and mentioning several future directions for related work.

2 Preliminaries

2.1 Basic string definitions

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 by $w[i \ldots j]$. For convenience, we let $w[i \ldots j] = \epsilon$ for $j < i$. We also denote by $Prefix(w)$ the set of all prefixes of the string $w \in \Sigma^*$, and denote by $Substring(w)$ the set of all substrings of $w \in \Sigma^*$. The reversal of a string $w = w[1 \ldots |w|]$ is defined by $w^R = w[|w|] \cdots w[1]$.

2.2 BDDs

A BDD [3] is a directed-graph representation of a Boolean function, as illustrated in Figure 2(b). It is derived by reducing the binary tree graph representing a recursive *Shannon's expansion*, as shown in Figure 2(a). The following reduction rules yield a BDD, which can efficiently represent the Boolean function (see [4] for details).

- Delete all redundant nodes whose two edges point to the same node (see Figure 1(a)).
- Share all equivalent subgraphs (see Figure 1(b)).

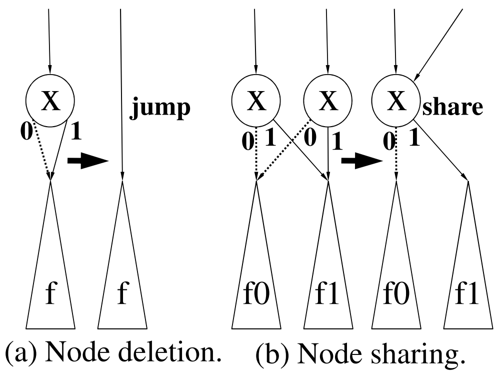


Figure 1: BDD reduction rule

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]

As shown in Figure 3, 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 [3], we can efficiently construct a BDD for the result of a binary logic operation (i.e. AND, OR, XOR), given a pair of operand BDDs. This algorithm is based on hashtable techniques, and the computation time is almost linear with data size, unless the data overflows the main memory (see [9] for details).

2.3 ZDDs

A ZDD [8, 10] is a special type of BDD for the efficient manipulation of sets of combinations [11]. An example is shown in Figure 4.

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 Figure 5.

[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.

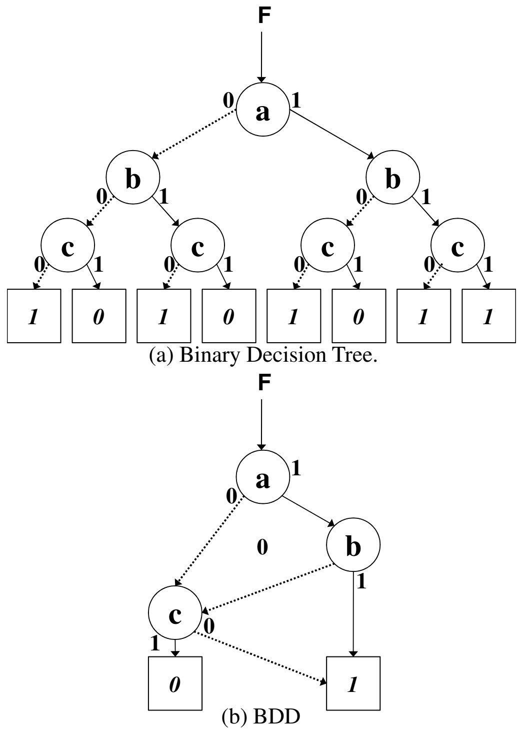

(a) Binary Decision Tree.
(b) BDD

Figure 2: Binary tree and BDD for $F = (a \wedge b) \vee c$

$$F1 = a \wedge \overline{b}$$
$$F2 = a \oplus b$$
$$F3 = \overline{b}$$
$$F4 = a \vee \overline{b}$$

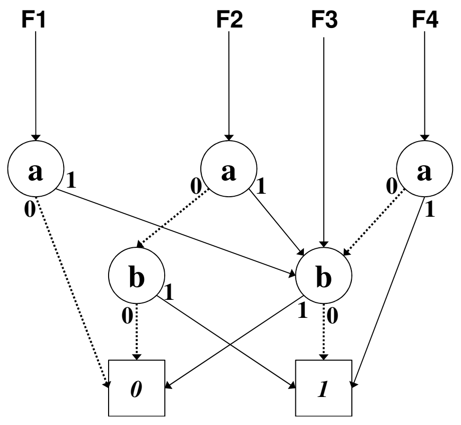


Figure 3: Shared multiple BDDs

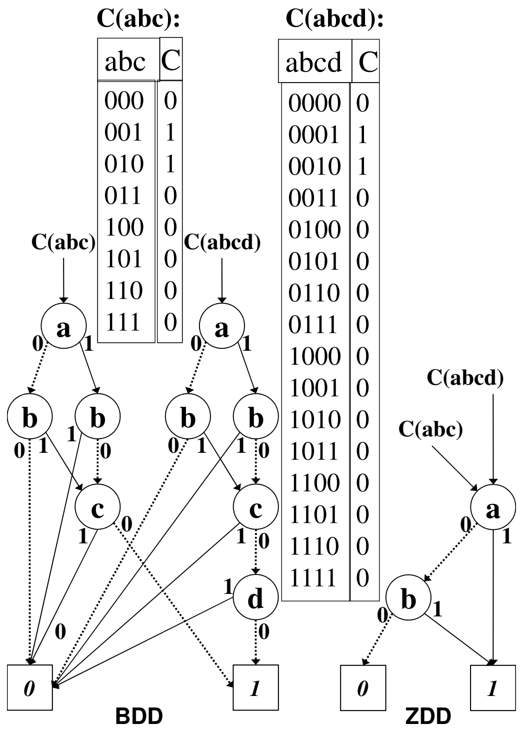
When no equivalent nodes exist in a ZDD, which is the worst case, the ZDD structure explicitly stores all combinations of all items, in addition to using an explicit linear linked-list data structure. An example is shown in Figure 6. Note that the order of ZDD size never exceeds that of the explicit 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.

In the past, an approach was reported that applied ZDD to the sequence data-mining field [5]. Figure 7 shows an example of a ZDD representing a set of sequences. However, the research did not demonstrate the performance required for application to practical-scale problems. This was attributed to the fact that the ZDDs were not optimized for sets of sequences. In fact, if we use SeqBDDs, as follows, we can manipulate sets of sequences efficiently.

2.4 SeqBDDs

A SeqBDD is a ZDD that has removed the ordering constraint only for 1-edges with ordered 0-edges, 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. A SeqBDD node $N = node(x, N_1, N_0)$ denotes an internal node labeled by letter X, and N_1 (or N_0) denotes its 1-edge (or 0-edge). The letter of node N has a lower order (appears earlier in the variable ordering) than the letter of N_0. We denote the total number of descendant nodes of node N, including N itself, by $|N|$. Let x be a letter and P and Q be two sets of strings. In SeqBDD, a 0-terminal node



C(abc):

abc	C
000	0
001	1
010	1
011	0
100	0
101	0
110	0
111	0

C(abcd):

abcd	C
0000	0
0001	1
0010	1
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

Figure 4: Example of ZDD effect

encodes the empty set (*emptyset*), and a 1-terminal node encodes the set of empty string (ϵ). Figure 8 shows an example. 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).

We define an operation P.push(x) that appends x to the head of every sequence in P, and an operation $P \cup Q$ that returns the set of sequences occurring in either P or Q.

Definition 1 *A SeqBDD node $N = node(x, N_1, N_0)$ represents a set of sequences S such that $S = S_{\bar{x}} \cup S_x.push(x)$, where S_x is the set of sequences that begin with element x (with the head elements being removed), and $S_{\bar{x}}$ is the set of sequences that do not begin with x. Node N_1 represents set S_x, and node N_0 represents set $S_{\bar{x}}$.*

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, with

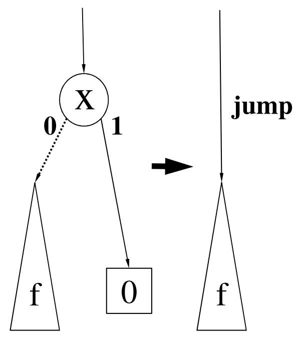


Figure 5: ZDD reduction rule

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 [7]. 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 Figure 10. 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 $P = Q$.

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-edge of the output node should contain all sequences that begin with x, with the 0-edge of the output node containing

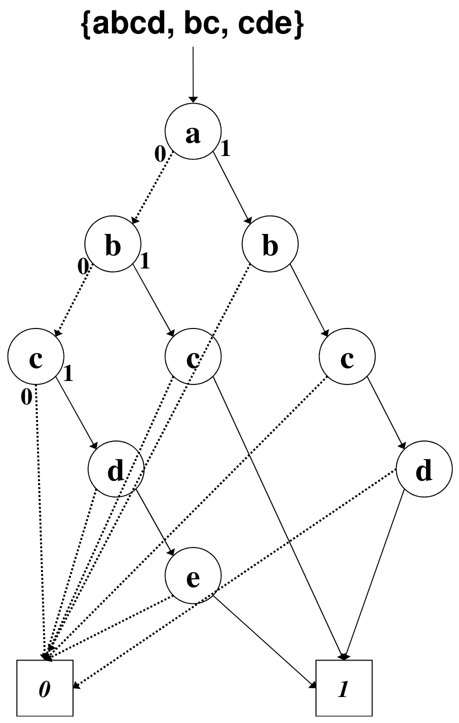


Figure 6: Explicit representation by ZDD

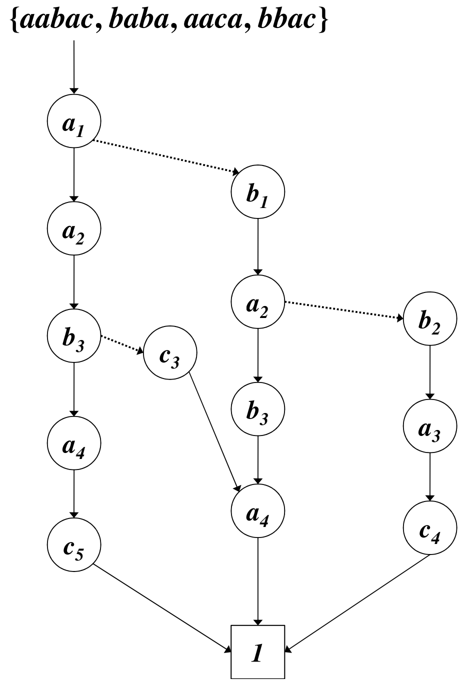


Figure 7: ZDD representation of a set of sequences $aabac, baba, aaca, bbac$

the remaining sequences. Suppose x is smaller[2] 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 procedures 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 $\cup$.

[2]A 1-terminal node is considered an internal node with a bigger letter than all letters in the alphabet.

2.5 Online substring-index construction problem

A *Substring index* for a text T is a data structure that stores the set of all substrings of T and has the following operations:

- Index construction $P \leftarrow \text{create}(T)$: Return the index that represents the set of all substrings of $T \in \Sigma^*$.
- Membership determination $P.\text{member}(w)$: For a given string $w \in \Sigma^*$, determine whether "$w \in \mathit{Substring}(T)$" is true.

For a given online text $T = T[1 \dots n]$, the *online substring-index construction problem* is expressed as follows. For each $i = 1, \dots, n$, construct the substring index S_i of $T_i = T[1..i]$, where S_i is built incrementally from S_{i-1} and $T[i]$.

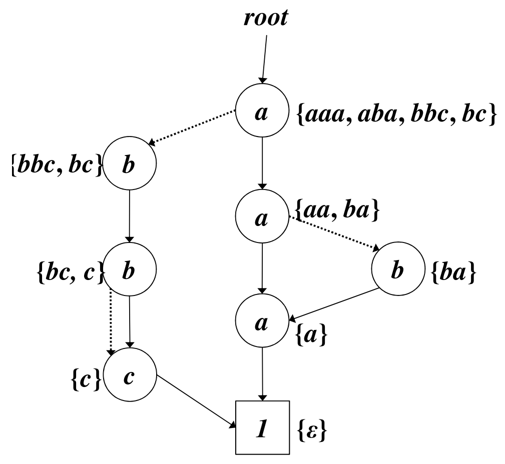


Figure 8: An example of SeqBDD, and the sets of sequences that each node represents

3 Algorithms

In this section, we introduce *SuffixDD*, which is a substring index based on SeqBDD, and give an efficient construction algorithm.

3.1 SuffixDD

Let $T \in \Sigma^*$ be a text of length $n \geq 0$. Then the *SuffixDD* of text T is a SeqBDD constructed for the set of all substrings $Substring(T)$. Figure 11 shows an example of SuffixDD for *aabac*. The number of nodes of SuffixDD is linear with respect to the length of the text T. It is possible to show that SuffixDD is isomorphic to DAWG. From this fact, we can infer that the number of nodes of SuffixDD is $O(n)$ for a text of length $n \geq 0$.

3.2 Naive construction method

Figure 12 shows a straightforward construction algorithm for SuffixDD. We first build a SeqBDD R_i that represents only the string $T_i = T[1 \dots i]$ at each step. Then we update SuffixDD by adding all those suffixes of T_i that are not contained in S_i at the time, and the result is set as the new SuffixDD. In this algorithm, a SeqBDD must be built at every reading of a new letter, to represent all suffixes of the current text. An example of this construction process is shown in Figure 13.

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.

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. Let n be the length of the text. The computation time of $\cup$ is $O(|\Sigma|n)$ in this case, and it is repeated $O(n^2)$ times. These considerations lead to the following lemma.

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

3.3 Efficient construction method

Using the simple algorithm, we build a SeqBDD of length n and add new suffixes one by one whenever a new letter is read. We now give an efficient algorithm for SuffixDD construction. It does not involve bottom-up building or repeating the same union-operation computations. The idea is to construct a SuffixDD comprising all reversed substrings, maintaining a SeqBDD that represents all

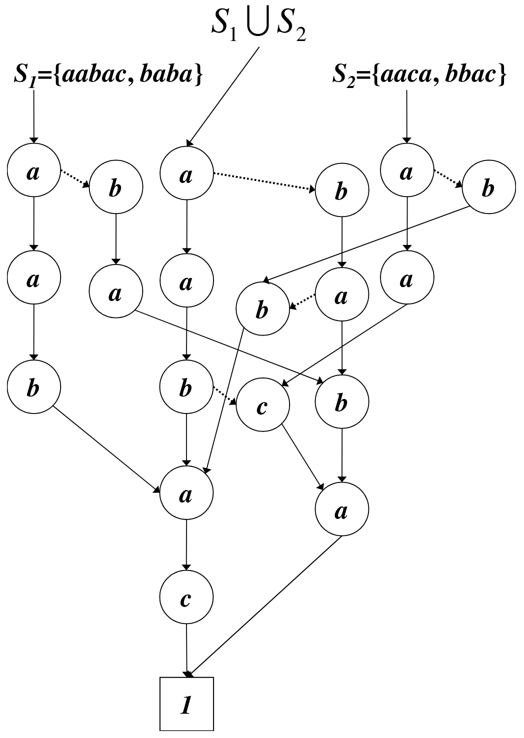


Figure 9: An example of SeqBDD showing the result of a ∪ operation and its inputs.

```
Procedure $P \cup Q$
Input: $P$, $Q$: SeqBDD;
 1: case $P$ = 0-terminal node : return $Q$
 2: case $Q$ = 0-terminal node : return $P$
 3: case $P = Q$ : return $P$
 4: If there is a $P \cup Q$ result, then return it.
 5: $x \leftarrow P$.top; $y \leftarrow Q$.top
 6: $P_1 \leftarrow P$.onset($x$); $P_0 \leftarrow P$.offset($x$)
 7: $Q_1 \leftarrow Q$.onset($y$); $Q_0 \leftarrow Q$.offset($y$)
 8: case $x = y$ : $z \leftarrow x$; $R_1 \leftarrow P_1 \cup Q_1$; $R_0 \leftarrow$
    $P_0 \cup Q_0$
 9: case $x < y$ : $z \leftarrow x$; $R_1 \leftarrow P_1$; $R_0 \leftarrow P_0 \cup Q$
10: case $x > y$ : $z \leftarrow y$; $R_1 \leftarrow Q_1$; $R_0 \leftarrow P \cup Q_0$
11: return $R_1$.push($z$) $\cup R_0$
    %It means making $node(z, R_1, R_0)$ and return it.
```

Figure 10: Code for constructing $P \cup Q$

prefixes of the reversed text, and then updating it by just one union operation.

The procedure can be expressed as shown in Figure 14. We maintain and update a SeqBDD R that represents all reversed suffixes by appending one node to the old R. To update SuffixDD S, we compute the union of $S \cup R$ once per iteration. Let T and P be the given text and pattern. Therefore, the SeqBDD represents $Substring(T^R)$ in this algorithm. An example is shown in Figure 15, for $T = aba$. For this SuffixDD, the string-matching problem is to determine whether or not $P^R \in Substring(T^R)$.

Let n denote the length of the text. This algorithm makes a new node ($O(1)$) in one iteration, and computes the union with the existing SuffixDD. Note that updating SuffixDD takes $O(|\Sigma|n)$ time.

Theorem 1 BuildFast *can be executed in time* $O(|\Sigma|n^2)$.

(1) The correctness of procedure BuildFast

Proof

We prove that the variable S comprises $Substring(T)$ at each iteration. Let n be the text length.
(i) If $n = 0$:

We have $T = \epsilon$, and, because the substrings of the empty string are also only empty strings, then the 1-terminal node represents it. Therefore, variable S is the SuffixDD of the text at the first iteration because BuildFast starts with S = 1-terminal node. It follows that R represents $Prefix(T^R)$ =1-terminal node, for the same reason.
(ii) Suppose that S is the SuffixDD of the text $T[1 \ldots i]^R$ when $n = i (i > 0)$, and R represents $Prefix(T[1 \ldots i]^R)$.

R is updated to represent $Prefix([1 \ldots i+1]^R)$ at the time that the $i+1$th letter is read in line 5. The SuffixDD of $T[1 \ldots i+1]^R]$ is obtained by the union in line 6:

$$Substring(T[1 \ldots i]^R) \cup Prefix(T[1 \ldots i+1]^R)$$

$$= Substring(T[1 \ldots i+1]^R)$$

The variable S is shown to represent

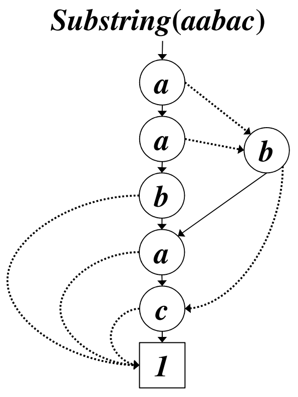


Figure 11: Structure for a SuffixDD, given an input *aabac*

$Substring(T[1 \dots i+1]^R)$ by the above equality. The proof is therefore complete by induction.

(2) Running time of algorithm BuildFast

The length of a straight SeqBDD R is i when the text length is i. Therefore, $S \cup R$ terminates after i recursions on a 1-edge in the worst case. The number of times we traverse 0-edges is at most $|\Sigma|$. Therefore, one $\cup$ will take $O(|\Sigma|i)$ for each letter:

$$\Sigma_{i=1}^{n} O(|\Sigma|i) = O(|\Sigma|n^2)$$

The BuildFast algorithm can therefore be performed in $O(|\Sigma|n^2)$ for a text whose length is n, and we have proved Theorem 1.

4 Experiments

We implemented the proposed algorithm in the concurrent programming language Erlang, and performed experiments using artificial data. We used an 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. We used artificial data, in which all letters occurred with equal likelihood, as the test data. The text strings were generated randomly. The alphabet size was either 4 or 128, depending on the experiment.

```
Procedure BuildNaive
Input: Text string $T$
 1: $n \leftarrow 1$
 2: $S \leftarrow$ 1-terminal node
 3: while(){
 4:   $T[n] \leftarrow$ reads a new letter
 5:   $n++$
 6:   Build SeqBDD $R$ that represents $T[1 \dots n]$.
 7:   do{
 8:     $OldS \leftarrow S$
 9:     $S \leftarrow R \cup OldS$
10:     $R \leftarrow R$.onset($R$.top)
11:   }while($S \neq OldS$)
12: }
```

Figure 12: Naive procedure for construction of a SuffixDD

4.1 Experiments and results

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 empirical results were obtained.

Experiment 1:

Figure 16 shows the running time for BuildFast. Figure 17 shows the running time for both methods. Figure 18 is a logarithmic version of Figure 17.

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 2: Figure 19 shows the running time of BuildFast with two random texts, for which $|\Sigma| = 4$ and $|\Sigma| = 128$, respectively.

In both cases, running times seem to be proportional to n. The $O(n^2)$ running time is derived from analysis of the algorithm. The SuffixDDs for $|\Sigma| = 4$ were computed nearly six times faster than the SuffixDDs for $|\Sigma| = 128$.

Experiment 3: Figure 20 shows the number of nodes in SuffixDDs for $|\Sigma| = 4$ and $|\Sigma| = 128$.

The size of the SuffixDD is linearly related to the length of the text. The SuffixDDs for the larger alphabet were slightly smaller, but the reason is not clear. For example, the number of nodes for alice29.txt, which comprises 152,089 bytes, is 288,759.

Experiment 4: Figure 21 shows the computation times for constructing SeqBDDs that represent the union, intersection, and difference of two SuffixDDs.

Experiment 5: 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. Figures 22 and 23 show the running time and the number of nodes for the SuffixDDs of S, T, and U.

Both the running time and the size are smaller for the texts with stronger bias. The SuffixDDs for T were built 20 times faster than for the other texts, and the number of nodes was $n + 1$, i.e., the smallest size.

5 Conclusion

In this paper, we have discussed SeqBDD, proposed by Loekito *et al.*, and have presented an algorithm that constructs SuffixDD using only primitive SeqBDD operations. We described experiments showing that our proposed algorithm is faster than the simple method, and have found that our method leads to a thousand-fold increase in program speed. The results of our study suggest that SuffixDD can store all substring sets effectively. Therefore, SuffixDD appears to be a valid method.

In future work, additional static and dynamic statistics should be gathered, and a more sophisticated study comparing SuffixDD with existing substring indices would be desirable. SuffixDD can be applied effectively to other string problems, and it would appear to be useful to develop applications for string searching based on SeqBDD. It would be interesting to extend SuffixDD to allow mismatching. It is clear that there is considerable work yet to be done on SuffixDD.

References

[1] S.B. Akers, "Binary decision diagrams," IEEE Trans. Comput., Vol. C-27, No. 6, pp. 509–516, 1978.

[2] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell, "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.

[3] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Trans. Comput., Vol. C-35, No. 8, pp. 677–691, 1986.

[4] D.E. Knuth, "The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques and Binary Decision Diagrams," Addison-Wesley, 2009.

[5] R. Kurai, S. Minato, and T. Zeugmann, "N-Gram analysis based on zero-suppressed BDDs," In T. Washio, et al. editors, "*New Frontiers in Artificial Intelligence*, *Joint JSAI 2006 Workshop, Post-Proceedings*", LNAI 4384, pp. 289–300, Springer, Feb. 2007.

[6] C.Y. Lee, "Representation of switching circuits by binary-decision programs," Bell Syst. Tech. J., Vol. 38, pp. 985–999, 1959.

[7] E. Loekito, J. Bailey, and J. Pei, "A binary decision diagram based approach for mining frequent subsequences," Knowledge and Information Systems, doi:10.1007/s10115-009-0252-9, 2009.

[8] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," Proc. 30th ACM/IEEE Design Automation Conf. (DAC-93), pp. 272–277, 1993.

[9] S. Minato, "Binary Decision Diagrams and Applications for VLSI CAD," Kluwer Academic Publishers, November 1996.

[10] S. Minato, "Zero-suppressed BDDs and their applications," Int. J. Software Tools for Technology Transfer (STTT), Vol. 3, No. 2, pp. 156–170, Springer, May 2001.

[11] S. Minato and H. Arimura "Frequent pattern mining and knowledge indexing based on Zero-suppressed BDDSs," The 5th international workshop on knowledge discovery in inductive databases (KDID'2006), pp. 83–94.

[12] E. Ukkonen "Constructing suffix trees on-line in linear time," IFIP'92, pp. 484–492, 1992.

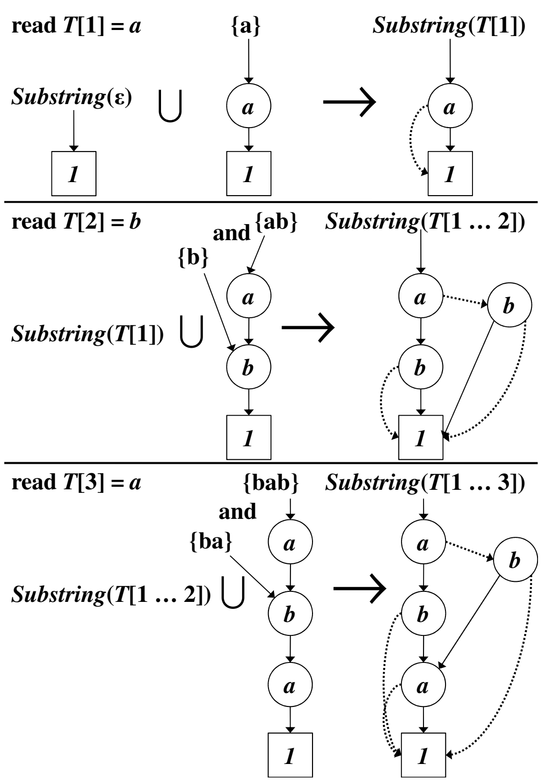


Figure 13: Processing of *aba* in BuildNaive

```
Procedure BuildFast
Input: Text string $T$
1: $R \leftarrow$ 1-terminal node
2: $S \leftarrow$ 1-terminal node
3: while(){
4:     $x \leftarrow$ read a letter
5:     $R \leftarrow R$.push($x$) $\cup$ 1-terminal node
       %make $node(x, S,$ 1-terminal node), and
       let it be the new $S$
6:     $S \leftarrow S \cup R$
7: }
```

Figure 14: Efficient code for constructing a SuffixDD

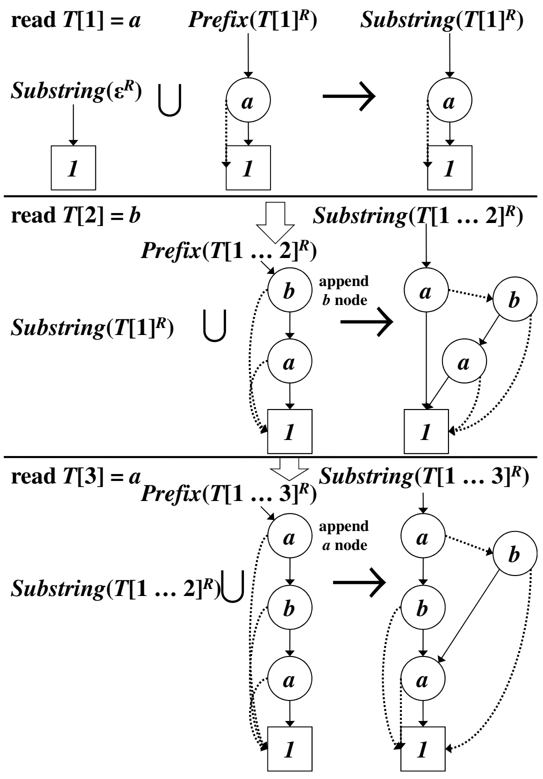


Figure 15: Processing of *aba* in BuildFast

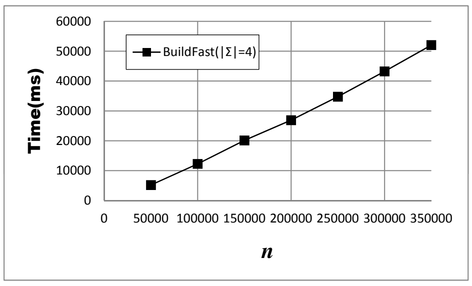


Figure 16: Experiment 1: Empirical running time, for BuildFast

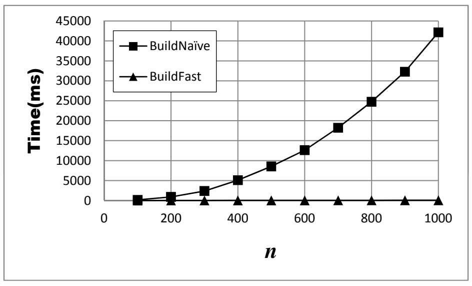


Figure 17: Experiment 1: Running time to construct a SuffixDD, for BuildNaive and BuildFast

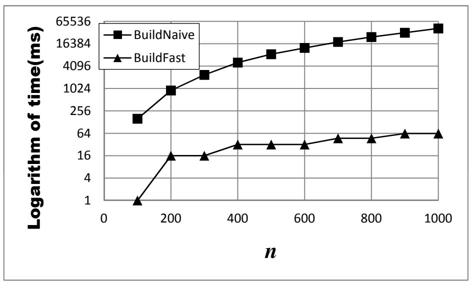


Figure 18: Experiment 1: Logarithm of the running time of Figure 17

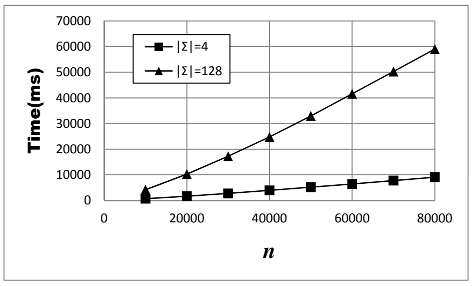


Figure 19: Experiment 2: Empirical running time of BuildFast, for $|\Sigma| = 4$ and $|\Sigma| = 128$

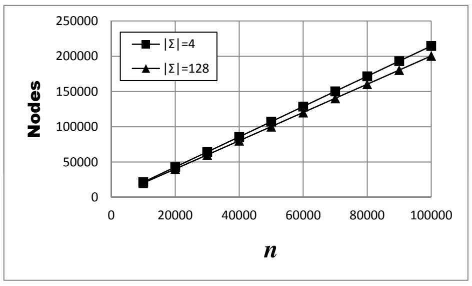


Figure 20: Experiment 3: Empirical size of SuffixDDs, for $|\Sigma| = 4$ and $|\Sigma| = 128$

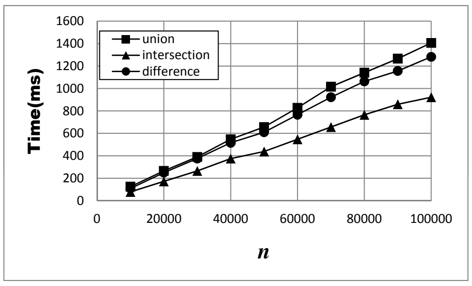


Figure 21: Experiment 4: Empirical computation times for binary set operations on two SuffixDDs

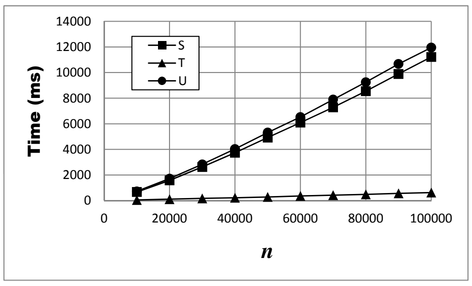


Figure 22: Experiment 5: The running time of BuildFast, for texts with biased probabilities

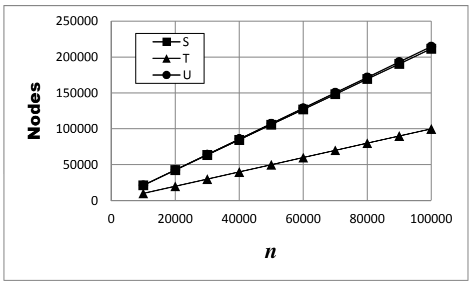


Figure 23: The number of nodes of SuffixDD, for texts with biased probabilities