TCS-TR-A-11-53

TCS Technical Report

Efficient Algorithms on Sequence Binary Decision Diagrams for Manipulating Sets of Strings

by

Shuhei Denzumi, Ryo Yoshinaka, Shin-ichi Minato, and Hiroki Arimura

Division of Computer Science

Report Series A

April 30, 2011

Hokkaido University
Graduate School of
Information Science and Technology

Email: arim@ist.hokudai.ac.jp

Phone: +81-011-706-7680
Fax: +81-011-706-7680

Efficient Algorithms on Sequence Binary Decision Diagrams for Manipulating Sets of Strings

Shuhei Denzumi[1], Ryo Yoshinaka[2], Shin-ichi Minato[1,2], and Hiroki Arimura[1]

[1)]Graduate School of IST, Hokkaido University, Japan
[2)]ERATO MINATO Discrete Structure Manipulation System Project, JST, Japan
{denzumi,ry,minato,arim}@ist.hokudai.ac.jp

Abstract. We consider sequence binary decision diagrams (sequence BDD or SDD, for short), which are compact representation for manipulating sets of strings, proposed by (Loekito, et al., Knowl. Inf. Syst., 24(2), 235-268, 2009). An SDD resembles to an acyclic DFA in binary form with different reduction rules from one for DFAs. In this paper, we study the power of SDDs for storing and manipulating sets of strings on shared and reduced SDDs. Particularly, we first give the characterization of minimal SDDs as reduced SDDs. Then, we present simple and efficient algorithms for various problems related to reduced and shared SDDs: on-the-fly and off-line minimization, dynamic string set construction, and factor SDD construction. Finally, we run experiments on real data sets that show the efficiency and usefulness of SDDs in large-scale string processing.

1 Introduction

Backgrounds. Recent emergence of massive text and sequence data has increased the importance of string processing technologies. In particular, compact string index data structures for storing, filtering, and manipulating sets of strings [3, 7, 9, 13, 16, 21] have attracted much attention in many applications such as bioinfomratics, natural language processing, event stream processing, and sequence mining. *Sequence binary decision diagrams* (*sequence BDDs* or *SDDs*, for short) are compact representation for manipulating sets of strings, proposed by Loekito, et al. [15]. Roughly speaking, an SDD is a node-labelled DAG in binary form that resembles to an *acyclic DFA* (*ADFA*) [9, 21] but with different reduction rule from one for ADFAs, where equivalent siblings as well as children can be shared. From this difference, minimal SDDs can be slightly smaller than the equivalent minimal ADFAs, and the implementation of minimization and other set manipulation procedures can be much more simplified and made easy, as it will be shown in this paper later. In the paper [15], the framework of shared and reduced SDDs are shown to be useful in string processing applications. However, the theoretical analysis need further investigation.

Main results. In this paper, we study basic properties and efficient algorithms for shared and reduced SDDs as follows. First, we study on-the-fly and

off-line minimization of SDDs. Particularly, we present the characterization of minimal SDDs as reduced SDDs (Theorem 1), the correctness of Getnode procedure for the on-the-fly minimization in SDD, the time complexity of a linear-time off-line minimization procedure Reduce (Corollary 3). Next, we present efficient algorithms for related problems: We show an incremental construction algorithm with insert, delete, and switch operations for an SDD from a string set, and shows its linear-time complexity in dynamic setting (Theorem 4 and Corollary 5). We present a practical algorithm RecFSDD for constructing a factor SDD from an input SDD, which is a simple recursive procedure based on tabling and binary set operation, and ran in almost linear time in our experiments. Finally, we run experiments on real data sets that show the usefulness of SDDs in large-scale string processing.

In summary, the SDD is a compact index structure, which is conceptually simple, having clear semantics, easy to implement, and extensible to various purposes. Moreover, the above results demonstrate that the framework of shared and reduced SDD environment is a practical choice for large-scale string applications utilizing the power of node-sharing, on-the-fly minimization, and built-in set operations, inherited from ADFAs and BDDs (See below for BDDs).

Related works. There have been a number of studies on the use of minimal ADFAs as compacted string indexes [9, 21] and as compacted factor indexes [3, 7, 13, 16, 21]. In the former line of researches, there are a number of works on off-line algorithms for state minimization and Boolean set operations [12]. For example, Daciuk *et al.* [9] presented a linear-time incremental algorithm for constructing a minimal ADFA from a string set. Mohri *et al.* [21] studied the construction of factor automata (FA) and factor transducers from an ADFA and showed a non-trivial upperbound of the size and the construction time for FA. Apart from well-known linear time algorithms [7, 26] for suffix trees (STs) and FAs (or DAWGs) based-on suffix links, there is a line of researches on practical top-down algorithms, such as wotd [11], for constructing factor indexes [11, 21, 24] similar to our RecFSDD. Although they have non-linear time complexity, some of their advantages are reported that they are conceptually simple [11], run fast in practice [11, 21], and have good I/O complexity [24].

On the other hand, logic design community has developed compact storage and manipulation techniques for combinatorial structures [5, 14, 17, 18, 27] in the form of BDDs (*binary decision diagrams*) for Boolean functions, invented by Bryant [5] in 80s, and its variant ZDDs (*zero-suppressed BDDs*) for combinatorial sets, proposed by Minato [18] in 90s. Hence, it is an interesting idea to add new functionality to string indexes based on BDDs and ZDDs. Recently, Loekito, et al. [15] discovered that ZDDs with node-sharing and zero-suppress rules resemble to ADFAs [21] in binary form, when the ordering constraint on 1-children is removed, and then, they proposed the sequence BDDs (SDDs) [15] as an alternative to ASFA. In [15], they presented the basic framework of shared and reduced SDDs and studied efficient algorithms for set operations $\cup$ and $\setminus$, with application to a sequence mining problem, while further study is required

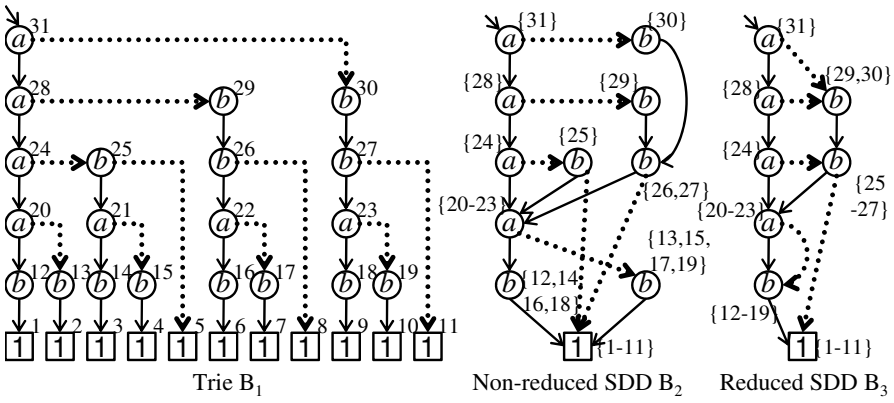

Fig. 1. Examples of three index structures on $\Sigma_1 = \{a, b\}$ for the same string set $S_1 = \{aaaab, aaab, aabab, aabb, aa, abbab, abbb, ab, bbab, bbb, b\}$: a trie B_1 in the leftmost-child right-sibling form (left), a minimal DFA as a non-reduced SDD B_2 (middle), and a reduced SDD B_3 (right). In the figure, solid and dotted arrows indicate the 1- and 0-edge and the numbers attached to a node indicate the equivalent nodes in B_1. The 0-terminal **0** and all edges pointing to **0** are omitted in the figure.

for properties and algorithms of SDDs from the view of automata and algorithm theories.

Organization of this paper. In Sec. 2, we give basic definitions on sequence BDDs. Then, we discuss the on-the-fly minimization in Sec. 3, the dynamic string set construction in Sec. 4, the factor SDD construction in Sec. 5, and show experimental results in Sec. 6. Finally, Sec. 7 concludes this paper.

2 Preliminaries

In this section, we give basic definitions and notations in sets of strings, BDDs, and SDDs according to [14, 15, 18]. For the definitions and results not found here, please consult papers on SDDs [10, 15].

Strings and string sets. Let Σ be an alphabet of symbols with a total order $\preceq_\Sigma$ on Σ. A *string* on Σ is a sequence $s = s[1] \cdots s[n]$ of letters, where $|s| = n$ is the *length* and $s[i] \in \Sigma$ $(1 \leq i \leq n)$. If $s = xyz$ for some $x, y, z \in \Sigma^*$, then we say that x is a *prefix*, y is a *factor*, and z is a *suffix* of s. A *string set* (or a *language*) is any finite $S \subseteq \Sigma^*$. We denote by $|S| = m$ the cardinality, by $||S|| = \sum_{s \in S} |s|$ the total size of S, and by $\mathsf{maxlen}(S) = \max\{\, |s| \mid s \in S \,\}$. For any $x \in \Sigma$, we define $x \cdot S = \{\, xy \mid y \in S \,\}$. For a string set S, we denote by $Suf(S)$ the set of all suffixes of strings in S.

Sequence BDDs. Let dom and op be countable domains of the nodes and the operations, resp. A *sequence binary decision diagram* or a *sequence BDD* (abbreviated as *SDD*[1] here) is a DAG $B = \langle \Sigma, V, \tau, \mathbf{r}, \mathbf{0}, \mathbf{1} \rangle$, with the *node set*

[1] Note that the abbreviation SeqBDD was used to denote sequence BDD in the original paper by *Loekito et al.* [15]. We also note that the abbreviation SDD was also used for the *set decision diagrams* [6] and the *spectral decision diagrams* [25] in logic design community. However, we use the abbreviation for its simplicity if no confusion arises.

$V = V(B) \subseteq \mathsf{dom}$, the *root* $\mathbf{r}$ and two *terminal nodes* $\mathbf{1}$ and $\mathbf{0} \in V$, called the 1- and 0-terminals. The nodes in $V_{\mathrm{N}} = V \backslash \{\mathbf{0}, \mathbf{1}\}$ are called nonterminals. Each node $v \in V_{\mathrm{N}}$ of B is labeled by a symbol $v.lab \in \Sigma$ and has the 1- and 0-children, denoted by $v.1$ and $v.0$, resp., which correspond to the *leftmost-child* and the *right-sibling* in a DAG in *binary form* [1, 14]. Formally, the information is described by a function $\tau : V_{\mathrm{N}} \to \Sigma \times V^2$ that assigns the *node triple* $\tau(v) = \langle v.top, v.0, v.1 \rangle$ to each $v \in V_{\mathrm{N}}$. For a node u, a *path* $\pi \in \{0,1\}^*$ specifies the node $u.\pi$ by $u.\varepsilon = u$ and $u.\pi\alpha = (u.\pi).\alpha$ for every $\alpha = 0, 1$. In our shared SDD environment, the function τ is implemented by a hash table *uniqtable*, and shared by more than one SDDs in common. We define the *size* of B by $|B| = |V_{\mathrm{N}}| = |V| - 2$, the number of non-terminals in B. [2] An SDD B is *deterministic*, i.e., the siblings of each node has no repeated labels and are ordered from left to right by symbol order $\prec_\Sigma$ on their labels. B is *acyclic* if there exists a strict partial order $\succ_V$ on nodes such that for every non-terminal $v \in V_{\mathrm{N}}$, $v \succ_V v.0$ and $v \succ_V v.1$ hold. We assume that any SDD B is *well-defined* meaning that B is both deterministic and acyclic.

Minimal and Reduced SDDs. For any node $v \in V$, we define the language $L_B(v)$ of node v of SDD B: (i) $L_B(\mathbf{0}) = \emptyset$; (ii) $L_B(\mathbf{1}) = \{\varepsilon\}$; (iii) $L_B(v) = L_B(v.0) \cup x \cdot L_B(v.1)$ for any $v \in V_{\mathrm{N}}$ and $x = v.top$. Equivalently, $L_B(v)$ is the set of the strings spelled out by all the paths π from v to the terminal $\mathbf{1}$ obtained by concatenating the labels $v.lab$ of the nodes such that π contains both of v and $v.1$. The *language* of B is defined by the set $L(B) = L_B(\mathbf{r})$. We say that SDD B is *equivalent* to SDD B' if $L(B) = L(B')$. An SDD B is said to be *minimal* if it has the smallest number of nodes among the equivalent SDDs, i.e., $|B| \leq |B'|$ for any SDD B' such that $L(B') = L(B)$. A reduced SDD is a normal form of SDDs. An SDD $B = \langle \Sigma, V, \tau, \mathbf{r}, \mathbf{0}, \mathbf{1} \rangle$ is said to be *reduced* if it satisfies the conditions 1 and 2 below:

1. For any $u, v \in V_{\mathrm{N}}$, $\tau(u) = \tau(v)$ implies $u = v$ (*node-sharing rule*).
2. For any $v \in V_{\mathrm{N}}$, $v.1 \neq \mathbf{0}$ holds (*zero-suppress rule*).

The above rules say that no distinct non-terminal nodes have the same triple, and the 1-child of any non-terminal node v is not the 0-terminal.

Example 1. In Fig. 1, we show examples of a trie B_1 (left), a minimal DFA B_2 (middle), and a reduced SDD B_3 (right) for the same string set $S_1 = \{aaaab, aaab, aabab, aabb, aa, abbab, abbb, ab, bbab, bbb, b\}$ on an alphabet $\Sigma = \{a, b\}$ all in the form of SDDs, where the set of numbers attached to each node indicate the equivalent node numbers, and the 0-terminal $\mathbf{0}$ and all edges pointing to $\mathbf{0}$ are omitted for convenience. In the figure, we observe that

– we get the DFA B_2 from the trie B_1 and the minimal SDD B_3 from the DFA B_2 by merging a set of equivalent nodes that have the same subgraph to get a smaller structure. For example, we obtain B_2 from B_1 by merging

[2] In our implementation of an shared and reduced SDD environment described in Sec. 6, $|B| = |V_{\mathrm{N}}|$ is actually the number of entries in a node table since terminals $\mathbf{0}$ and $\mathbf{1}$ have no associated entries in the table.

all 1-terminals into the node labeled [1-11], the nodes $12, 14, 16$, and 18 into the node labeled $[12, 14, 16, 18]$, and so on.
- Similarly, we obtain B_3 from B_2 by merging the nodes labeled $[12, 14, 16, 18]$ and $[13, 15, 17, 19]$ into the node labeled [12-19], the nodes labeled [25] and $[26, 27]$ into the node labeled [25-27], and so on.
- In their sizes, the reduced SDD B_3 is smallest having seven nodes, while the minimal ADFA B_2 and the trie B_1 have 10 and 20 nodes, resp.
- In all structures, the path $\pi_1 = 101101$ spells out the same string $s_1 = abbb$ on the node sequences $\boldsymbol{v}_1 = (\overline{31}, 28, \overline{29}, \overline{26}, 22, \overline{17}, 8)$ in B_1, $\boldsymbol{v}_2 = (\overline{31},\ 28,\ \overline{29},\ \overline{[26, 27]},$ [20-23], $\overline{[13, 15, 17, 19]}$, [1-11]) in B_2, and $\boldsymbol{v}_3 = (\overline{31},\ 28,\ \overline{[29, 30]},\ \overline{[25\text{-}27]}$, [20-23], $\overline{[12\text{-}19]}$, [1-11]) in B_3, where the overlined nodes are the nodes having 1-edge on π emitting a symbol in s_1.
- We also see that B_2 is an SDD representing a minimal DFA, but is not reduced because three pairs of nodes $\langle 29, 30\rangle$, $\langle 25, [26, 27]\rangle$ and $\langle [12, \ldots], [13, \ldots]\rangle$ violate the node-sharing rule.

Relationship between an ADFA and an SDD. The relationship between an ADFA and an SDD is summarized as follows. An ADFA (in binary format) is an SDD without sibling sharing, that is, an SDD satisfying conditions (i) only first children have incoming 1-edges and (ii) each node has at most one incoming 0-edges. For example in Fig. 1, we see that the DFA B_2 satisfies the above conditions, while it is not the case for the SDD B_3. Thus, we know that the size of a minimal SDD B is no more than that of an equivalent minimal ADFA A, i.e., $|B| \leq |A|$. It is shown in [10] that the minimal ADFA A equivalent to an SDD B is at most $|\Sigma|$ times larger than B.

3 Reduction and Minimization algorithms

In this section, we show the equivalence of reduced SDDs and minimal SDDs, and then, present a linear-time reduction algorithm for SDD.

3.1 The minimality of a reduced SDD

Let $L \subseteq \Sigma^*$ is any finite set of strings. We first see that the canonical SDD for L is defined, in a similar way to the minimal DFA in Myhill-Nerode theorem (e.g.,[12]), as follows. The main difference from one for DFA is the operation to derive successors of a state. We define string sets

- $L.1 = \{\, \alpha \in \Sigma^* \mid x\alpha \in M, x = top(L) \,\}$ (onset operation),
- $L.0 = L \setminus (x \cdot L.1)$ (offset operation),

where $top(L)$ is $top(L) = \min_{\leq_\Sigma}\{\, x \in \Sigma \mid x\alpha \in L, \alpha \in \Sigma^* \,\}$, i.e., the first letter of the lexicographically smallest string in L if it exists. For any $L \notin \{\emptyset, \{\varepsilon\}\}$, $top(L)$, $L.1$, and $L.0$ are always defined. Now, we build the canonical SDD B_* for L by taking as its *nodes* the different sets $L.\pi$ for all $\pi \in \{0, 1\}^*$, as the *root* $L = L.\varepsilon$, as the *1-terminal* $\mathbf{1} = \{\varepsilon\}$, as the *0-terminal* $\mathbf{0} = \emptyset$, and with

1- and *0-edges* defined for any $L \neq \mathbf{1}, \mathbf{0}$, respectively, by $(L.\pi).1 = L.(\pi.1)$ and $(L.\pi).0 = L.(\pi.0)$. The nodes $\mathbf{1}$ and $\mathbf{0}$ has no out-going edges. Clearly, the canonical SDD for L is completely determined by L. For any $L \subseteq \Sigma^*$, let us define its *index* by $\mathsf{idx}(L) = |L| + \mathsf{maxlen}(L) \geq 0$.

Lemma 1. *For any $L \subseteq \Sigma^*$, if L is neither $\emptyset$ nor $\{\varepsilon\}$, then for every $\alpha = 0, 1$, $L.\alpha$ is defined, and moreover, $\mathsf{idx}(L) > \mathsf{idx}(L.\alpha)$ holds.*

Proof. The claims are easily shown by induction on $\mathsf{idx}(L)$. □

Lemma 2. *The canonical SDD B_* for any $L \subseteq \Sigma^*$ is well defined.*

Proof. The deteminicity is obvious. Let us define $\mathsf{idx}(L) = |L| + \mathsf{maxlen}(L) \geq 0$. From Lemma 1, we see that there is no arbitrary long chain of nodes $L_1, L_2, \ldots$ in B_* following either 0- or 1-edges such that $\mathsf{idx}(L_1) > \mathsf{idx}(L_2) > \cdots$. This shows the acyclicity of B_*. □

We prepare some technical lemmas on the relationship between canonical SDDs and reduced and non-reduced SDDs.

Lemma 3. *Any reduced SDD B is isomorphic to the canonical SDD B_* for $L(B)$. More precisely, for any $v \in V_N$, we have: (0) $v.lab = top(L)$. (1) $L_B(v) = \emptyset$ iff $v = \mathbf{0}$. (2) $L_B(v.1) = L_B(v).1 \neq \emptyset$. (3) $L_B(v.0) = L_B(v).0$.*

Proof. Recall that $L_B(v) = x \cdot L_B(v.1) \cup L_B(v.0)$ for any $v \in V_{\mathrm{N}}$. (0) Since B is deterministic and v satisfies zero-suppress rule, it is proved. (1) Immediate from zero-suppress rule. Let v be any non-terminal node v with $L = L_B(v)$. From (2) of Lemma 3, we have $L_B(v.1) \neq \emptyset$. (2) From zero-suppress rule, $L_B(v.1) \neq \emptyset$. Since $x = v.lab$ is the smallest among its sibling, the claim holds. (3) Since B is deterministic, we have $(x \cdot L_B(v.1)) \cap L_B(v.0) = \emptyset$. Thus, the claim follows. □

Lemma 4. *Let B_* and B be any SDDs. If B_* is canonical and $L(B_*) = L(B)$, then for any $u \in V(B_*)$, there is some $v \in V(B)$ such that $L_{B_*}(u) = L_B(v)$.*

Proof. Let $L = L(B_*) = L(B)$. Let $u \in V(B_*)$ be any node of a canonical SDD B_*. Then, we have a path $\pi(u) \in \{0,1\}^*$ in B_* such that $\mathbf{r}.\pi(u) = u$. Since B_* is unique, such a $\pi(u)$ is unique for each u. We show the claim by induction on the length $m = |\pi(u)|$. (Case 1) The case that $|\pi(u)| = 0$ is obvious. (Case 2) The case that $|\pi(u)| > 0$. Then, there exists the unique parent p of u such that $u = p.\alpha$ for some $\alpha \in \{0,1\}$. Because B_* is canonical, p is non-terminal and $L_{B_*}(p)$ is neither $\emptyset$ nor $\{\varepsilon\}$. Clearly, $\pi(p) < \pi(u)$. By inductive hypothesis, there exists some $q \in V(B)$ with $L_{B_*}(p) = L_B(q)$. It is possible that the node q has an empty 0-child. However, we can show that there exists some younger sibling $\tilde{q}$ of q such that $L_B(\tilde{q}) = L_B(q)$ and $\tilde{q}.1 \neq \mathbf{0}$ by following 0-edges from q. Otherwise, we finally reach either $\mathbf{1}$ or $\mathbf{0}$. From this observation, let $\tilde{q}$ be a non-terminal node of B which is a descendant of q and satisfies that $L_B(\tilde{q}) = L_B(q)$ and $\tilde{q}.1 \neq \mathbf{0}$. If $L_{B_*}(p) = L_B(\tilde{q})$ and $\tilde{q}.1 \neq \mathbf{0}$, we have that $L_{B_*}(p.1) = L_B(\tilde{q}.1)$ and $L_{B_*}(p.0) = L_B(\tilde{q}.0)$. Thus, $L_{B_*}(p.\alpha) = L_B(\tilde{q}.\alpha)$ holds, and the lemma is proved. Combining the above arguments, we show the lemma. □

Now, we have the main theorem of this section.

Theorem 1 (characterization of minimal SDDs). *For any SDD B with the language $L = L(B)$, the following (1)–(3) are equivalent each other.*

(1) B is a reduced SDD.
(2) B is a canonical SDD for L up to isomorphism.
(3) B is a minimal SDD.

Proof. [(1) $\Rightarrow$ (2)]: Suppose that B is reduced. Let B_* be the canonical SDD for L. From Lemma 3, if we take $\phi(v) = L_B(v)$, we see that $\phi : V(B) \to V(B_*)$ satisfies that $\phi(\mathbf{1}) = \{\varepsilon\}$, $\phi(\mathbf{0}) = \emptyset$, $\phi(\mathbf{r}) = L$, and for any $v \in V(B)$, $\phi(v.\alpha) = \phi(v).\alpha$ holds for every $\alpha = 1, 0$. From Lemma 3 and the fact that the states of B_* are the different languages, we see that ϕ is actually a bijection between $V(B)$ and $V(B_*)$. This shows the claim (End of (1) $\Rightarrow$ (2)).

[(2) $\Rightarrow$ (3)]: Suppose without loss of generality that B is the canonical SDD B_* for L. Obviously, all nodes of B_* have mutually distinct languages. By contradiction, suppose that there exists some SDD B' such that $L(B) = L(B')$ and $|B'| < |B|$. From Lemma 4, there exists some mapping $\phi : V(B) \to V(B')$ such that $L_B(u) = L_{B'}(\phi(u))$ holds for any $u \in V(B)$. By pigeonhole principle, we see that ϕ must be injective; Otherwise, there exist some distinct pair of nodes, $u_1, u_2 \in V(B)$ $(u_1 \neq u_2)$, such that $\phi(u_1) = \phi(u_2)$, and thus, $L_B(u_1) = L_B(u_2)$; The contradiction is derived. Hence, the claim is proved. (End of (2) $\Rightarrow$ (3)).

[(3) $\Rightarrow$ (1)]: We show that if B is not reduced then B is not minimal. Suppose that B is not a reduced SDD. Then, (i) If B violates the node-sharing rule, then there are distinct nodes $u_1, u_2 \in V_{\mathrm{V}}$ $(u_1 \neq u_2)$ such that $\tau(u_1) = \tau(u_2)$. Then, we identify u_1 and u_2. (ii) If B violates the zero-suppress rule, then there is a non-terminal $u \in V_{\mathrm{V}}$ such that $u.1 = \mathbf{0}$, and we replace u with $u.0$. Then, the resulting reduced SDD B' is equivalent to B and has a strictly smaller than B, and thus, B is not minimal. (End of (3) $\Rightarrow$ (1)). Combining the above arguments, the theorem is proved. □

3.2 On-the-fly and off-line minimization

At the top of Fig. 2, we show the node allocation procedure Getnode for on-the-fly minimization of SDDs as well as the hash tables $uniqtable : \Sigma \times \mathsf{dom}^2 \to \mathsf{dom}$ that corresponds to τ for node-sharing, and $cache : \mathsf{op} \times \mathsf{dom}^* \to \mathsf{dom}$ that stores invocation patterns of user-defined operations for avoiding redundant computation. In all results in this paper, we assume that these hash tables are global variables, and a look-up for a hash table takes $O(1)$ time; We have to add additional $O(\log n)$ term if we use balanced binary tree dictionary [1]. In the shared and reduced SDD environment [15] studied here, we use write-only construction, similar to [11], such that any new SDD is constructed by adding a new node on the top of already constructed SDDs using a call of Getnode given existing nodes as its arguments. The next lemma say that on-the-fly construction of SDDs is correct under condition below.

Global variable: $uniqtable, cache$: hash tables for triples and operations.

Proc Getnode(x: letter, P_0, P_1: SDD):

1: **if** ($P_1 = \mathbf{0}$) **return** P_0; /* zero-suppress rule */
2: **else if** (($R \leftarrow uniqtable[\langle x, P_0, P_1\rangle]$) exists) **return** R; /* node-sharing rule */
3: **else**
4: $R \leftarrow$ a new node with $\tau(R) = \langle x, P_0, P_1\rangle$;
5: $uniqtable[\langle x, P_0, P_1\rangle] \leftarrow R$;
6: **return** R;

Algorithm Reduce(P: possibly non-reduced SDD):

1: **if** ($P = \mathbf{1}$ or $P = \mathbf{0}$) **return** P;
2: **else if** (($R \leftarrow cache[$"$Reduce(P)$"$]$) exists) **return** R;
3: **else**
4: $R \leftarrow$ Getnode(x, Reduce($P.0$), Reduce($P.1$));
5: $cache[$"$Reduce(P)$"$] \leftarrow R$;
6: **return** R;

Fig. 2. An algorithm for computing the reduced version of a seqBDD

Lemma 5 (correctness of on-the-fly reduction). *Let B be any reduced SDD and $\langle x, v_0, v_1\rangle \in \Sigma \times \mathsf{dom}^2$ be any triple with $x \prec_\Sigma v_0.top$. Then, if we invoke $v =$ Getnode(x, v_0, v_1) and add the result v to V, then the resulting SDD B' obtained from B is well-defined and reduced, too.*

In Fig. 2, we show our off-line reduction algorithm Reduce using Getnode that computes the reduced version $red(B)$ from an input SDD B.

Theorem 2 (reduction). *The algorithm* Reduce *of Fig. 2 computes a reduced SDD $red(B)$ equivalent to B in $O(n)$ time and space in the input size $n = |B|$.*

Proof. For the equivalence of the language of B and $red(B)$, the node-sharing and zero-suppress rules at Lines 1 and 2 in Getnode does not change the language. Lemma 5 ensures that the resulting SDD B is kept reduced after adding new nodes returned by Getnode. For time complexity, we see that the use of *cache* is sufficient to ensure the linear-time complexity by avoiding the duplicated call of Reduce. □

From Theorem 2 and Theorem 1, we have the complexity of the minimization problem for SDDs using the algorithm Reduce.

Corollary 3 (complexity of off-line minimization) *For any SDD B, the minimal SDD equivalent to B is linear time and space computable in $n = |B|$.*

Global variable: *uniqtable*, *cache*: hash tables for triples and operations.

Algorithm $\mathsf{Meld}_\diamond(P, Q$: possibly non-reduced SDDs):
Output: The reduced SDD for the melding $P \diamond Q$ given $F_\diamond : \{0,1\}^2 \to \{0,1\}$;

```
 1: if (P = 0 or Q = 0 or P = Q)
 2:    if (F_◇[sign(P), sign(Q)] = 0) return 0; /* See text for F_◇. */
 3:    else if P ≠ 0 return P;
 4:    else if Q ≠ 0 return Q;
 5: else if (F_◇[0,1] = F_◇[1,0] and P > Q) return Meld_◇(Q, P);
 6: else if ((R ← cache["Meld_◇(P,Q)"]) exists) return R;
 7: else
 8:    x ← P.lab; y ← Q.lab;
 9:    if (x ≺_Σ y) R ← Getnode(x, Meld_◇(P.0, Q), Meld_◇(P.1, 0));
10:    else if (x ≻_Σ y) R ← Getnode(y, Meld_◇(P, Q.0), Meld_◇(0, Q.1));
11:    else if (x = y) R ← Getnode(x, Meld_◇(P.0, Q.0), Meld_◇(P.1, Q.1));
12:    cache["Meld_◇(P,Q)"] ← R;
13:    return R;
```

Fig. 3. An algorithm $\mathsf{Meld}_\diamond$ for built-in binary set operations $\diamond \in \{\cup, \cap, \setminus, \oplus\}$.

4 A linear-time dynamic construction algorithm

This section studies the dynamic construction problem for SDDs in the shared and reduced SDD environment. First, we prepare built-in binary operations for SDDs. Let $\diamond \in \{\cup, \cap, \setminus, \oplus\}$ be the names of set operations $\diamond : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$. We define $F_\diamond : \{0,1\}^2 \to \{0,1\}$ by $F_\cup[x,y] = x \vee y$, $F_\cap[x,y] = x \wedge y$, $F_\setminus[x,y] = x \wedge \neg y$, and $F_\oplus[x,y] = x \oplus y$ (exclusive-or), In Fig. 3, we give the algorithm $\mathsf{Meld}_\diamond$ that computes the reduced SDD R for the set $L(R) = L(P) \diamond L(Q)$ of two SDDs given $F_\diamond$ [5, 10, 14, 15], where $\mathsf{sign}(P)$ is 0 if $P = \emptyset$ and 1 otherwise. The $\mathsf{Meld}_\diamond(P, Q)$ runs in $O(|P| \cdot |Q|)$ and also in $O(|R|)$ in the worst case. See the paper [10] for the details.

For any SDD B with $S = L(B)$ and any string s, $\mathsf{add}(B, s)$, $\mathsf{delete}(B, s)$, and $\mathsf{switch}(B, s)$, resp., build and return the reduced SDD B' for the string sets $S \cup \{s\}$, $S \setminus \{s\}$, and $S \oplus \{s\} = (S \setminus \{s\}) \cup (\{s\} \setminus S)$. The *dynamic construction problem* for SDDs is the problem of constructing a sequence of SDDs $(B_i)_{i=0}^k$ ($k \geq 0$) as follows. The inputs are an SDD $B_0 = \mathbf{0}$, an empty hash table $\tau = \mathit{uniqtable}$, and a sequence $s_1, \ldots, s_k \in \Sigma^*$ of $k \geq 1$ strings. For every $i \geq 1$, we apply one of the operations $\alpha \in \{\mathsf{add}, \mathsf{delete}, \mathsf{switch}\}$ to compute $B_i = \alpha(B_{i-1}, s_i)$. In Fig. 4, we present the algorithms for these operations. Given a string s, MakeString, the algorithm first builds a reduced SDD for s in linear-time, and then apply α using algorithm $\mathsf{Meld}_\diamond$ in linear-time without touching all but $O(n)$ nodes though $\mathsf{Meld}_\diamond$ requires quadratic time in general [10]. This claim is formally verified as follows.

Global variable: *uniqtable*, *cache*: hash tables for triples and operations.

Proc MakeString($s \in \Sigma^*$: string):
Output: The reduced SDD Q for a string s;
$Q = \mathbf{1}$; **for** $(j = |s|, \ldots, 1)$ **do** $Q \leftarrow \mathsf{Getnode}(s[j], \mathbf{0}, Q)$; **return** Q;

Algorithm add(B, s) $\equiv$ $\mathsf{Meld}_\cup(B, \mathsf{MakeString}(s))$;
Algorithm delete(B, s) $\equiv$ $\mathsf{Meld}_\setminus(B, \mathsf{MakeString}(s))$;
Algorithm switch(B, s) $\equiv$ $\mathsf{Meld}_\oplus(B, \mathsf{MakeString}(s))$;

Fig. 4. The algorithm MakeString for constructing the reduced SDD Q for s, and the algorithms add, delete, and switch for dynamically adding, deleting, and switching a single string s to an SDD B.

Theorem 4 (dynamic construction). *For any SDD B and any string s, Algorithms* add*,* delete*, and* switch *of Fig. 4 correctly work, take $O(|s|)$ time, and moreover, add at most $O(|\Sigma| \cdot |s|)$ nodes to and delete no nodes from the input SDD B.*

Proof. We give a proof sketch for add. Other cases are proved similarly. The correctness is obvious from that of $\mathsf{Meld}_\cup$ and MakeString. Given a string s, MakeString builds a chain SDD Q in $O(|\Sigma| \cdot m)$ time. During the computation of $\mathsf{Meld}_\cup(B, Q)$ in Fig. 3, we have $v.0 = \mathbf{0}$ for any node v in Q. Thus, we see that Lines 9, 10, and 11 become equivalent invocations

- (i) $\mathsf{Getnode}(x, \mathsf{Meld}_\diamond(P.0, Q), P.1)$,
- (ii) $\mathsf{Getnode}(y, P, Q.1)$, and
- (iii) $\mathsf{Getnode}(x, P.0, \mathsf{Meld}_\diamond(P.1, Q.1))$,

resp., after constant steps. Calls (i) and (iii) are used for traversing 0- and 1-edges, resp., and (ii) for inserting the rest of Q. From this observation, we can show that $\mathsf{add}(B, Q)$ makes $O(|\Sigma| \cdot |Q|)$ calls of $\mathsf{Meld}_\cup$, adds $O(|\Sigma| \cdot |Q|)$ nodes, and takes $O(|\Sigma| \cdot |Q|)$ time. □

Corollary 5 (linear-time dynamic construction) *For any input sequence of strings $(s_i)_{i=0}^k$ $(k \geq 0)$, the dynamic construction problem for SDDs is solvable in $O(|\Sigma| \cdot n)$ total time and adds $O(n)$ nodes, where $n = \sum_{i=1}^k |s_i|$ is the input size.*

Proof. Immediate from Theorem 4. □

Corollary 5 generalizes the linear-time complexity of the incremental string set construction problem for ADFA by [9]. Interestingly, we observe that that add precisely simulates the algorithm of [9] in write-only manner. In the case that only the final SDD is needed, we can reclaim memory in $O(n)$ time by discarding intermediate SDDs with garbage collection (GC) [14, 15, 18].

Global variable: $uniqtable, cache$: hash tables.

Proc RecFSDD(P: SDD):

2: **if** $P = \mathbf{0}$ **or** $P = \mathbf{1}$ **return** P;
3: **else if** ($F \leftarrow cache[\text{“}RecFSDD(P)\text{”}]$ exists) **return** F;
4: **else**
5: $Pre(P.1) \leftarrow$ SDD for all prefixes of strings in $P.1$;
6: $F \leftarrow \mathsf{Meld}_\cup(Getnode(x, \mathbf{1}, Pre(P.1)),$
$\mathsf{Meld}_\cup(\mathsf{RecFSDD}(P.0), \mathsf{RecFSDD}(P.1))$;
7: $cache[\text{“}RecFSDD(P)\text{”}] \leftarrow F$;
8: **return** F;

Fig. 5. A recursive procedure RecFSDD for constructing the factor SDD of an input SDD.

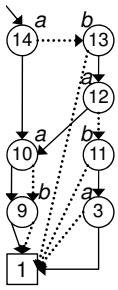

Fig. 6. An example of an FSDD for $S_2 = \{baab\}$.

5 A practical algorithm for factor SDD construction

The *factor SDD* (*FSDD*), denoted by $FSDD(B)$, for a set $S \subseteq \Sigma^*$ is the reduced SDD for Fact(S) of all factors of strings in S. The FSDD for an SDD B is simply the FSDD for $L(B)$. For example, Fig. 6 shows an examle of FSDD for $S_2 = \{baab\}$, which stores the set Fact(S_2) $= \{\varepsilon, a, aa, aab, ab, b, ba, baa, baab\}$ of all 9 strings with totally 17 letters. Since it is well-known that the size of factor automata $FA(S)$ for a string set S is linear in $||S||$ [3, 7], the size of $FSDD(B)$ is also linear in $|B|$. (See [10] for the tight upperbound.)

Since $|B|$ can be exponentially smaller than $||L(B)||$, efficient construction of FSDD from a given SDD is an interesting problem. In Fig. 5, we presents a simple recursive algorithm RecFSDD that computes $FSDD(B)$ from an input SDD B using $\mathsf{Meld}_\cup$ in write-only manner, where $Pre(P.1)$ is can be computed by redirecting all 0-edges pointing to $\mathbf{0}$ by those to $\mathbf{1}$ in $P.1$.

Theorem 6. *The algorithm* RecFSDD *of Fig. 5 computes $FSDD(B)$ from an input SDD B in $O(n^3)$ time and space, where $n = |B|$.*

Computation of RecFSDD seems similar to wotd algorithm [11] and other top-down algorithms [21, 24]. Actually, we observed that applications of Pre and $\mathsf{Meld}_\cup$, resp., correspond to ε-edge attachment and determinization in [21]. Experiments in Sec. 6showed that RecFSDD ran in almost linear time on some real data sets.

6 Experimental Results

Data and methods. In Table 1, we show the summary of our data sets, where BibleAll and BibleBi are sets of all sentences and all word bi-grams drawn from an English text `bible.txt`, and Ecoli is a single DNA string in `ecoli.txt` in

Table 1. Outline of data sets

Data	Size (byte)	#line	#Uniq line	Ave line len (byte)	$\|\Sigma\|$
BibleAll	4,047,392	30,383	30,129	133.2	62
BibleBi	7,793,268	767,854	154,479	10.1	27
Ecoli	4,638,690	1	1.0	4,638,690.0	4

Table 2. Comparison of algorithms for string set construction

Data	Input (Text)		Output (SDD)	Time (sec)		
	Size (byte)	Uniq ratio	Size (node)	ConstTRIE	ConstINC	ConstOTF
BibleAll	4,047,392	0.99	3,099,401	13.5	16.3	2.0
BibleBi	7,793,268	0.20	101,288	4503.2	4024.3	5.9

Table 3. Comparison of factor indexes and their construction algorithms on BibleAll data set consisting of long strings with few duplicates

Data BibleAll	Algorithm				
	STRIE	NaïveFSDD	RecFSDD	ST	SA
Size (Kilo node/cell)	14,284	6,110	6,110	6,093	4,047
Memory (Kilo byte)	185,687	183,300	183,300	109,680	36,427
Time (sec)	1,142.1	215.3	46.3	4.7	1.4

Table 4. Comparison of factor indexes and their construction algorithms on BibleBi data set consisting of short strings with many duplicates

Data BibleBi	Algorithm				
	STRIE	NaiveFSDD	RecFSDD	ST	SA
Size (Kilo node/cell)	1,315	309	309	8,251	7,793
Memory (Kilo byte)	17,093	9,270	9,270	148,520	70,139
Time (sec)	7,297.6	29.8	8.4	7.8	7.2

Canterbury corpus[3]. We implemented our shared and reduced SDD environment on the top of the *SAPPORO BDD package* [19] for BDDs and ZDDs written in C and C++, where each node is encoded in a 32-bit integer and a node triple occupies approximately 50 to 55 bytes in average including hash entries in *uniqtable*. We also used another implementation of SDD environment in functional language *Erlang*. Experiments were run on a PC (Intel Core i7, 2.67 GHz, 3.25 GB memory, Windows XP SP3). About 1.5 GB of memory was allocated to the SDD environment in maximum. For string set construction in Section 4, we implemented the following algorithms: an off-line algorithm ConstTRIE using reduction of a trie, an incremental algorithm ConstINC that applies Reduce every time adding a string, the proposed dynamic construction algorithm ConstOTF using on-the-fly minimization. For factor index construction in Sec. 5, we implemented the following algorithms: a suffix trie construction algorithm STRIE in binary form, two FSDD construction algorithms NaïveFSDD and the proposed RecFSDD, where the former uses direct set construction by ConstOTF from Fact(S), construction algorithms ST for suffix trees [16] and SA for suffix

[3] `http://corpus.canterbury.ac.nz/resources/`

arrays [2]. The running time for RecFSDD does not include the construction time for an input SDD.

Exp 1: String set construction. First, Fig. 7, Fig. 8, and Table 2 show the results. From Fig. 7, we saw that a minimal SDD was 5 to 8 percent smaller than the equivalent minimal ADFA in binary format. From Fig. 8, we saw that the proposed ConstOTF runs in linear-time as expected and was expected to be scalable. From the comparison of algorithms in Table 2, we saw that the proposed ConstOTF was much faster than other algorithms.

Exp 2: Factor index construction. From Figures 9 and 10, we observed that the memory usage and the running time of our algorithm RecFSDD are almost linear in the input size. Although the current theoretical upper bound of running time is cubic, this empirical result indicates practical efficiency of RecFSDD algorithm. From Tables 3 and 4, RecFSDD algorithm was faster than NaïveFSDD and STRIE algorithms on the set BibleAll of English sentences, and was competitive to ST and SA, and much faster than NaïveFSDD and STRIE on the set BibleBi of many short, duplicated strings.

For binary set operations, our preliminary experiments showed that it took less than seconds to take set operations $\cup$, $\cap$, and $\setminus$ of two SDDs with around three to four thousands of nodes each, which are relatively smaller than the running time for set construction and factor SDD construction. Overall, we conclude that the shared and reduced SDD environment with the above algorithms is a practical choice for storing string sets in large-scale string applications especially when we need flexible manipulation of them.

7 Conclusion

In this paper, we studied *sequence BDDs* (*SDDs*) and the associated algorithms for manipulating sets of strings. Based on the framework of shared and reduced SDDs, we show that the minimality of reduced sequence BDDs, and presented efficient algorithms for on-the-fly and off-line minimization, dynamic construction, and factor index construction from SDDs. demonstrating the power of combination of on-the-fly minimization and graph-based string indexes. Analysis of the factor SDD (FSDD) construction algorithm, development of external memory algorithms for FSDD, and real applications of SDDs are interesting future problems. It is another future problem to add to SDDs new functionalities of, e.g., detecting positions [4] and counting frequency [20].

Acknowledgements

They would like to thank Takashi Horiyama, Takeru Inoue, Jun Kawahara, Takuya Kida, Toshiki Saitoh, Yasuyuki Shirai, Kana Shimizu, Yasuo Tabei, Koji Tsuda, Takeaki Uno, and Thomas Zeugmann for their discussions and valuable comments. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, and MEXT/ JSPS Global COE Program, "*Center for Next-Generation Information Technology based on Knowledge Discovery and Knowledge Federation*," FY2007–2011.

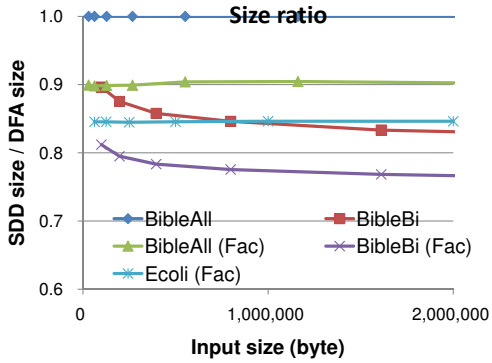

Fig. 7. Ratio between the sizes SDDs and DFAs in binary format

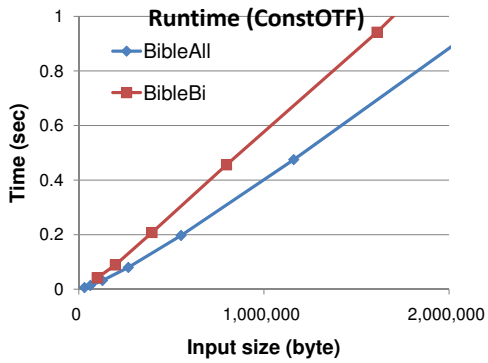

Fig. 8. Time of the incremental construction algorithm ConstOTF for SDDs

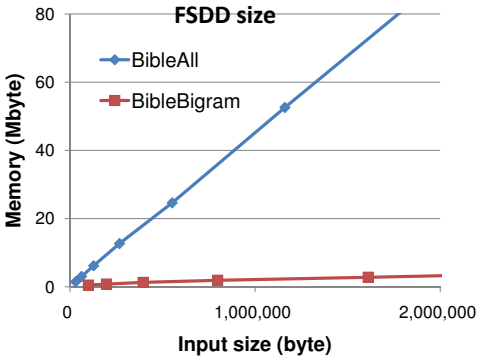

Fig. 9. The memory usage of FSDDs

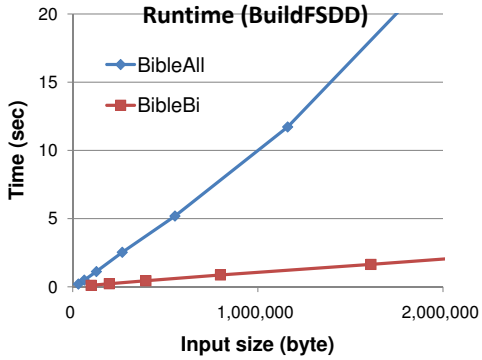

Fig. 10. Time of the construction algorithm RecFSDD for FSDDs

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. J. L. Bentley, R. Sedgewick, Fast Algorithms for Sorting and Searching Strings, *Proc. SODA'97*, SIAM, 360–369, 1997.
3. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.I.: The smallest automaton recognizing the subwords of a text, *TCS*, 40, 31–55, 1985.
4. A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis, *J. ACM*, 34(3), 578–595, 1987.
5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation, *IEEE. Trans. Comput.*, C-35(8), 677–691, 1986.
6. J. M. Couvreur and Y. Thierry-Mieg, Hierarchical decision diagrams to exploit model structure, *Proc. FORTE 2005*, LNCS 3731, 443-457, 2005.
7. M., Crochemore, Transducers and repetitions, *TCS*, 45(1), 63–86, 1986.
8. M. Crochemore, C. Hancart, T. Lecroq, *Algorithms on strings*, Cambridge University Press, 2007.
9. J. Daciuk, S. Mihov, B.W. Watson, R.E. Watson, Incremental construction of minimal acyclic finite-state automata, *Computational Linguistics*, 26(1), 2000.
10. S. Denzumi, R. Yoshinaka, H. Arimura, S. Minato, Notes on Sequence Binary Decision Diagrams and Acyclic Automata, *manuscript*, Hokkaido University, April 2011.

11. R. Giegerich, S. Kurtz, J. Stoye, Efficient implementation of lazy suffix trees *Softw. Pract. Exper.*, 33, 1035–1049, 2003.
12. J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Formal Language Theory, 2nd edition*, Addison-Wesley, 2001.
13. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa, Construction of the CDAWG for a Trie, *Proc. Prague Stringology Conf.(PSC 2001)*, 37-48, 2001.
14. D.E. Knuth, *The Art of Computer Programming, vo.4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley, 2009.
15. E. Loekito, J. Bailey, J. Pei, A Binary decision diagram based approach for mining frequent subsequences, *Knowl. Inf. Syst.*, 24(2), 235-268, 2009.
16. E. M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM*, 23, 262–272, 1976.
17. S. Minato, *Binary decision diagrams and applications for VLSI CAD*, Kluwer Academic, 1996.
18. S. Minato, Zero-suppressed BDDs and their applications, *International Journal on Software Tools for Technology Transfer*, 3(2), 156-170, Springer, 2001.
19. S. Minato, SAPPORO BDD package, Division of Computer Science, Hokkaido University, unreleased, 2011.
20. Shin-ichi Minato, Hiroki Arimura, Efficient method of combinatorial item set analysis based on zero-suppressed BDDs, *Proc. WIRI'05*, IEEE, 4-11, 2005.
21. M. Mohri, P. Moreno, E. Weinstein, Factor automata of automata and applications, *Proc. CIAA 2007*, 168-179, 2007.
22. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences.* Cambridge, 2002.
23. D. Perrin, Finite Automata, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen (Eds.), Elsevier and MIT Press, 1–57, 1990.
24. Y. Tian, S. Tata, R. A. Hankins, J. M. Patel, Practical methods for constructing suffix trees, *VDLB J.*, 14(3), 281–299, 2005.
25. M. Thornton and R. Drechsler, Spectral decision diagrams using graph transformations, *Proc. DATE'01*, IEEE, 713-719, 2001.
26. E. Ukkonen, On-line construction of suffix-trees, *Algorithmica*, 14(3), 249-260, 1995.
27. I. Wegener, *Branching programs and binary decision diagrams: theory and applications*, SIAM, 2000.