Sequence Binary Decision Diagram: Minimization, Relationship to Acyclic Automata, and Complexities of Boolean Set Operations☆

Shuhei Denzumi[a,*], Ryo Yoshinaka[b], Hiroki Arimura[a], Shin-ichi Minato[a,c]

[a] *Graduate School of Information Science and Technology, Hokkaido University, N14, W9, Sapporo 060-0814, Japan*
[b] *Graduate School of Infomatics, Kyoto University, Japan*
[c] *ERATO MINATO Discrete Structure Manipulation System Project, JST, Japan*

Abstract

The manipulation of large sequence data is one of the most important problems in string processing. In this paper, we discuss a new data structure for storing and manipulating sets of strings, called *Sequence Binary Decision Diagrams (sequence BDDs)*, which has recently been introduced by Loekito *et al.* (Knowl. Inf. Syst., 24(2), 235-268, 2009) as a descendant of both acyclic DFAs (ADFAs) and binary decision diagrams (BDDs). Sequence BDDs can compactly represent sets of sequences similarly to minimal ADFAs, and allow efficient set operations inherited from BDDs. We study fundamental properties of sequence BDDs, such as the characterization of minimal sequence BDDs by reduced sequence BDDs, non-trivial relationships between sizes of minimal sequence BDDs and minimal ADFAs, the complexities of minimization, Boolean set operations, and sequence BDD construction. We also show experimental results for real and artificial data sets.

Keywords: Sequence binary decision diagram, persistent data structure, deterministic finite automaton, minimization, Boolean set operation

1. Introduction

1.1. Background

Compact string indexes for storing sets of strings are fundamental data structures in computer science, and have been extensively studied for decades [3, 6, 7, 9, 15, 32]. Examples of compact string indexes include tries [1, 7], finite automata and transducers [9, 17], suffix trees [28], suffix arrays [27], DAWGs [3], and factor automata (FAs) [32]. Because of the rapid increase in the massive amounts of sequence data, such as biological sequences, natural language texts, and event sequences, these compact string indexes have attracted much attention and gained more importance [7, 15]. In such applications, an index is required not only to store sets of strings compactly for *searching*, but also to manipulate efficiently them with various set operations, e.g., *merge (union), intersection*, and *difference*.

Minimal acyclic deterministic finite automata (*minimal ADFAs*) [7, 9, 17] are one of the index structures that fulfill the above requirement based on finite automata theory, and have been used in many sequence processing applications [25, 33]. However, they have a drawback in that the procedures for minimization and various set operations are complicated because of the multiple branching of the underlying directed acyclic graph structure. To overcome this problem, Loekito *et al.* [24] proposed the class of *sequence binary decision diagrams* (*sequence BDDs* in this paper), which is a compact representation of finite sets of strings that allows a variety of operations to be performed for sets of strings.

☆ An earlier version of this paper was presented in part at the Prague Stringology Conference 2011 (PSC'11), Prague, Czech Republic, August 29–31, 2011.

* Corresponding author. Tel: +81-11-706-7678, Fax: +81-11-706-7680

Email addresses: denzumi@ist.hokudai.ac.jp (Shuhei Denzumi), ry@i.kyoto-u.ac.jp (Ryo Yoshinaka), arim@ist.hokudai.ac.jp (Hiroki Arimura), minato@ist.hokudai.ac.jp (Shin-ichi Minato)

A sequence BDD is a vertex-labeled graph structure, which resembles an acyclic DFA in binary form with associated minimization rules for sharing siblings as well as children that are different from those for a minimal DFA. Due to these minimization rules, a sequence BDD can be more compact than an equivalent ADFA. A novel feature of sequence BDDs is their ability to share equivalent subgraphs and the results of similar intermediate computation between different sequence BDDs, which avoids redundant generation of vertices and computation. In addition, sequence BDDs have a collection of manipulation operations that creates a new sequence BDD by combining existing ones in the current environment, which will be useful for implementing various string applications on the top of sequence BDDs.

1.2. Main results

In this paper, we present a theoretical as well as empirical analysis of the fundamental properties and manipulation operations of sequence binary decision diagrams, which have not previously been studied, namely, minimization, relationship to acyclic automata, and computational complexities of set operations. The results are summarized as follows.

Characterization of minimal sequence BDDs and minimization. By introducing the notion of the canonical sequence BDD, we give a characterization of minimal sequence BDDs. A sequence BDD is minimal if and only if it is isomorphic to the canonical sequence BDD of its language. Equivalently, a sequence BDD is minimal if and only if it is reduced w.r.t. the zero-suppress and subgraph-sharing rules. Then, we define our shared sequence BDD environment as a confluently persistent data structure based on tabulation with manipulation operations, and present an on-the-fly minimization procedure, Getvertex, which is the basis for all the algorithms in this paper. Then, we present an off-line minimization algorithm, Reduce, which computes the reduced sequence BDDs from an input sequence BDD in linear-time.

The relationship to acyclic automata. The structure of sequence BDDs apparently resembles that of *acyclic deterministic finite automata* (ADFAs), which are classical models for representing string sets. While a state of an ADFA may have many outgoing edges, a vertex of a sequence BDD always has two outgoing edges, which can be seen as just the "first-child next-sibling" representation of a branching with many edges. Indeed, one can find a straightforward translation from an ADFA to a sequence BDD and vice versa. However, there are subtle differences between these data structures, and in fact a sequence BDD can be even more compact than the corresponding ADFA. We show that the minimal sequence BDD is never larger than the minimal ADFA, but the latter can be $|\Sigma|$ times larger than the former for the same language over alphabet Σ.

The computational complexities of Boolean set operations. We study the complexity of the *binary synthesis* that directly constructs a minimal sequence BDD by combining two reduced sequence BDDs using Boolean set operations. We devise an efficient algorithm $\mathsf{Meld}_\diamond$, which uniformly implements eight Boolean set operations by generalizing the algorithms proposed in [4, 24] in the style of Knuth's *melding* for BDDs [22]. As compared with a straightforward two-stage algorithm for ADFAs using the product followed by minimization, the advantages of the proposed algorithm are its simplicity and efficiency based on on-the-fly minimization. Then, we show an upper bound that the time complexity of $\mathsf{Meld}_\diamond$ is quadratic in the input size, and linear in the non-reduced output size. Moreover, we show a lower bound that $\mathsf{Meld}_\diamond$ in fact requires quadratic time in input size, disproving the conjecture that $\mathsf{Meld}_\diamond$ runs in input-output linear time [24]. We present a practical algorithm, RecFSDD, for constructing a factor index from an input sequence BDD, which is a simple recursive procedure based on tabling and binary set operation.

On-the-fly and off-line construction. Although the Meld algorithm requires quadratic time in general, we show an expected linear-time bound for a special case where one of the arguments is a chain-like sequence BDD for one string. Using this property, we present efficient on-the-fly construction algorithms, AddString and DeleteString, that can add and delete a string of length m in $O(|\Sigma|m)$ expected time and additional space per string, respectively. We also present an off-line construction algorithm for a sorted set L of strings in $O(n)$ time and space in the total size n of L.

Experimental results. Finally, we run experiments on real and artificial data sets. We observe that reduced sequence BDDs are mostly as compact as, and sometimes even more compact than, minimal ADFAs, and that minimization, melding, and construction operations are sufficiently efficient and scalable to manipulate large string sets in a practical setting.

1.3. Related works

From the automata-theoretic viewpoint, sequence BDDs are direct descendants of deterministic finite automata (DFAs). In particular, acyclic DFAs (ADFAs) have been used for representing large vocabularies and sequence data [9, 25, 26, 33, 32]. There has been considerable research on the manipulation of finite automata in automata theory and string algorithms.

On the subject of the minimization of DFAs, Hopcroft [16] presented an $O(n \log n)$ state-minimization algorithm for a given DFA. Daciuk *et al.* [9] presented incremental and off-line algorithms for constructing minimal ADFA, to which Watson [37] added a survey of construction algorithms for minimal automata. Blumer *et al.* [3] and Crochemore [6] presented linear-time algorithms for constructing minimal ADFAs, called *DAWGs* and *factor automaton*, for all suffixes and all factors of a string, respectively. Mohri [32] gave an algorithm for constructing a factor automaton from an ADFA. Denzumi *et al.* [10, 12] presented a simple recursive algorithm that constructs a minimal factor sequence BDD from a given sequence BDD.

On the subject of Boolean set operations, a textbook [17] gave classic examples of a quadratic-time product algorithm for computing the union, intersection, and difference of two DFAs. Loekito *et al.* [24] presented algorithms for directly computing the reduced sequence BDD for the union and difference of two sequence BDDs. Our algorithm $\mathsf{Meld}_\diamond$ described in Sec. 5 generalizes their algorithm for all Boolean set operations. The work most closely related to that presented in this paper is that of Lucchesi *et al.* [26] as well as of Bubenzer [5]. Lucchesi *et al.* [26] introduced *binary automata*, which are essentially the same as sequence BDDs when restricted to acyclic ones, and discuss their minimization procedure. Bubenzer [5] studied a bottom-up minimization of acyclic DFAs using tabulation similar to Reduce described in Sec. 4. However, both papers considered the minimization procedure only, and did not consider other operations such as construction and binary operations.

Sequence BDDs inherit many of their features from *binary decision diagrams* (*BDDs*), which are a compact representation for storing and manipulating Boolean functions. BDDs were developed in the logic design community [4, 22, 29, 38]. In their early years, reduced BDDs were constructed from tree-like circuits through off-line minimization. It became popular to build large BDDs on-the-fly only after the invention of the binary synthesis algorithm by Bryant [4]. On the other hand, a variant of BDDs with zero-suppress and subgraph-sharing rules, called *zero-suppressed BDDs* (*ZDDs*), was proposed by Minato [29] in the 90s for sparse combinatorial sets, and later applied to databases and data mining [31]. Loekito *et al.* [24] discovered that if we remove the ordering constraint on the 1-edges from ZDDs, the resulting variant of ZDDs, which are in fact sequence BDDs, has a similar structure to that of ADFAs in binary form, and is suitable for storing and manipulating sets of strings. This observation led to the invention of sequence BDDs [24].

Sequence BDDs are closely related to the notion of *persistent data structures* [19, 20], which are data structures in which the structure can be changed without destroying the old version so that every version of the structure can be accessed or possibly modified every time. A data structure is called *fully persistent* [19, 20] if all versions can be both accessed and modified by a unary operation, and is called *confluently persistent* [20, 21] if it is fully persistent and all versions can be combined by a binary operation [20]. It is not easy to make a data structure confluently persistent [20], whereas there are general transformation methods for fully persistent ones [19]. In this context, our shared reduced sequence BDD described in Sec. 4 is a confluently persistent data structure for storing sets of strings in compressed form.

1.4. Organization of this paper

In Sec. 2, we introduce a formalization of sequence BDDs. In Sec. 3, we give a characterization of minimal sequence BDDs, and the shared sequence BDD environment. We give a linear-time minimization algorithm in Sec. 4. In Sec. 5, we discuss upper and lower bounds of the time complexities of Boolean set operation. In Sec. 6, we show linear-time construction algorithms. In Sec. 7, we give the size bounds for sequence BDDs and DFAs. In Sec. 8, we show an application of a sequence BDD and give an algorithm that constructs a factor graph from a sequence BDD. In Sec. 9, we show experimental results. In Sec. 10, we present our conclusion.

2. Preliminaries

In this section, we give the definition of sequence binary decision diagrams in the style of Bryant [4], which were originally introduced by Loekito, Pei, and Bailey [24].

Attribute	Terminal	Nonterminal
zero	null	$zero(v)$
one	null	$one(v)$
label	$\top$	$label(v)$
val	$value(v)$	null

Figure 1: The attribute values for a vertex v.

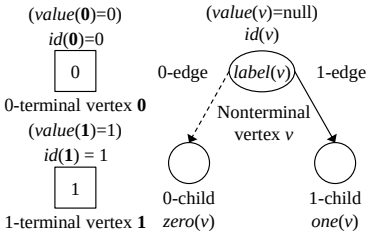

Figure 2: The 0-terminal, 1-terminal and nonterminal vertices.

2.1. Strings and languages

Let $\Sigma = \{a, b, \ldots\}$ be a countable alphabet of *symbols*. We assume that the symbols of Σ are ordered by a precedence $\prec_\Sigma$ such as $a \prec_\Sigma b \prec_\Sigma \cdots$ in a standard way. We also assume the special *null symbol* $\top \notin \Sigma$, which is larger than any symbol in Σ, i.e., $c \prec_\Sigma \top$ for any $c \in \Sigma$. The equality $=_\Sigma$ and the strict total order $\prec_\Sigma$ are defined on $\Sigma \cup \{\top\}$.

Let $s = a_1 \cdots a_n$, $n \geq 0$, be a string over Σ. For every $i = 1, \ldots, n$, we denote by $s[i] = a_i$ the i-th symbol and by $|s| = n$ the *length* of s. The *empty string* of length zero is denoted by ε. We denote by Σ^* the set of all strings of length $n \geq 0$. For strings s and t, we denote the *concatenation* of s and t by $s \cdot t$ or st. If $s = pqr$ for some possibly empty strings p, q, and r, we refer to p, q, and r as a *prefix*, *factor*, and *suffix* of s, respectively.

A *language* on an alphabet Σ is a set $L \subseteq \Sigma^*$ of strings on Σ. A *finite language* of size $m \geq 0$ is just a finite set $L = \{s_1, \ldots, s_m\}$ of m strings on Σ. A finite language L is referred to as a *string set*. We define the *cardinality* of L by $|L| = m$, the *total length* of L by $\|L\| = \sum_{i=1}^{m} |s_i|$, and the *maximal string length* of L by $\mathsf{maxlen}(L) = \max\{\, |s| \mid s \in L \,\}$. The *concatenation* of languages L and M is defined as $L \cdot M = \{\, st \mid s \in L, t \in M \,\}$. For any $a \in \Sigma$, the notation $a \cdot L = \{a\} \cdot L$ represents the set obtained from L by adding the symbol a to the beginning of each strings in L.

2.2. Graphical representation of a finite language

In this subsection, we give the sequence BDD, introduced by Loekito *et al.* [24], as our graphical representation of a finite language. Then, we show its canonical form. In a directed acyclic graph, a root is a vertex that has no parent. A vertex v in a sequence BDD is represented by a struct with the attributes *id*, *label*, *zero*, *one*, and *value*. We have two types of vertices, called nonterminal and terminal vertices, both of which are represented by the same type of struct, but the attribute values for a vertex v depend on its vertex type, as given in Fig. 1. A graphical explanation of the correspondence between the attribute values and the vertex type is given in Fig. 2.

Definition 1 (Sequence BDD). [24] A *sequence binary decision diagram* (a sequence BDD) is a multi-rooted, directed graph $G = (V, E)$ satisfying the following:

- V is a vertex set containing two types of vertices known as terminal and nonterminal vertices. Each has certain attributes, *id*, *label*, *zero*, *one*, and *value*. The respective attributes are shown in Table 1.
- There are two types of terminal vertices, called *1-terminal* and *0-terminal* vertices, respectively. A sequence BDD may have at most one of each of these vertices. A *terminal vertex* v has as an attribute a value $value(v) \in \{0, 1\}$, indicating whether it is a 1-terminal or a 0-terminal, denoted by $\mathbf{1}$ or $\mathbf{0}$, respectively. A *nonterminal vertex* v has as attributes a symbol $label(v) \in \Sigma$ called the *label*, and two children, $one(v)$ and $zero(v) \in V$, called the 1-child and 0-child. We refer to the pair of corresponding outgoing edges as the 1-edge and 0-edge from v . We define the *attribute triple* for v by $triple(v) = \langle label(v), zero(v), one(v) \rangle$. For distinct vertices u and v, $id(u) \neq id(v)$ holds. A *root* is any vertex with no parent.
- We assume that the graph is *acyclic* in its 1- and 0-edges. That is, there exists some partial order $\prec_V$ on vertices of V such that $v \prec_V zero(v)$ and $v \prec_V one(v)$ for any nonterminal v.
- Furthermore, we assume that the graph must be *ordered* in its 0-edges, that is, for any nonterminal vertex v, if $zero(v)$ is also nonterminal, we must have $label(v) \prec_\Sigma label(zero(v))$, where $\prec_\Sigma$ is the strict total order on symbols of $\Sigma \cup \{\top\}$. The graph is not necessarily ordered in its 1-edges.

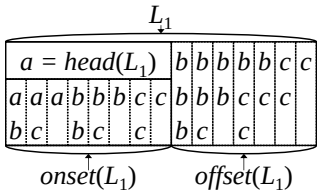

Figure 3: The binary decomposition of a finite language $L_1 = \{ aab, aac, aa, abb, abc, ab, acc, ac, bbb, bbc, bb, bcc, bc, cc, c \}$ in Example 1.

We define the *size* of the graph, denoted by $|G|$, as the number of its nonterminals. By definition, the graph consisting of a single terminal vertex, **0** or **1**, is a sequence BDD of size zero. For any vertex v in a sequence BDD G, the *subgraph rooted by* v is defined as the graph consisting of v and all its descendants. A graph G is called *single-rooted* if it has exactly one root, and *multi-rooted* otherwise. In this paper, we identify a single-rooted sequence BDD and its root node name. Multi-rooted graphs are useful in the shared sequence BDD environment described in Sec. 4.

2.3. The first semantics

For representing a Boolean function, Bryant presented BDD [4] based on the Shannon expansion [35] of a function. Below, we give the first semantics of sequence BDD based on the binary decomposition of a finite language, string counterpart of the Shannon expansion. A sequence BDD can define its language based on the binary decomposition defined below. In the following, we consider only finite languages. A language is *trivial* if it is either $\{\varepsilon\}$ and $\emptyset$, and *nontrivial* otherwise. Clearly, any nontrivial language contains at least one string of length more than zero.

Then, we introduce three operations, *head*, *onset*, and *offset*. Let L be a finite non-trivial language. The *head* of L, denoted by $head(L)$, is the smallest first symbol of a non-empty string in L. Consider all strings in L that start with the symbol $c = head(L)$. The language obtained by removing the first symbol, c, from all these strings is called the *onset language* of L, and is denoted by $onset(L)$. The language consisting of all strings in L that do not start with c is called the *offset language* of L, and is denoted by $offset(L)$.

In summary, we have

$$head(L) = \min\{ c \in \Sigma \mid cs \in L, s \in \Sigma^* \}, \tag{1}$$

$$onset(L) = \{ s \mid cs \in L, c = head(L) \}, \tag{2}$$

$$offset(L) = \{ cs \mid cs \in L, c \neq head(L) \} \cup (\{\epsilon\} \cap L). \tag{3}$$

If L is trivial, the operations *offset* and *onset* are undefined, while $head(L)$ is defined as the largest symbol $\top \notin \Sigma$. For any nontrivial language L, we refer to as the *binary decomposition* of L the decomposition of L given by

$$L = (head(L) \cdot onset(L)) \ \cup \ offset(L). \tag{4}$$

Example 1. Let us consider the finite language $L_1 = \{ aab, aac, aa, abb, abc, ab, acc, ac, bbb, bbc, bb, bcc, bc, cc, c \}$ on the alphabet $\Sigma_1 = \{a, b, c\}$ consisting of 15 strings with a total size of 37 symbols. Fig. 3 illustrates the binary decomposition of finite language $L = L_1$, where $head(L_1) = a$, $onset(L_1)$ contains eight strings, and $offset(L_1)$ contains seven strings.

We show the next lemma, which is essential in the definition of sequence BDDs.

Lemma 1 (Uniqueness of binary decomposition). *(1) Using the binary decomposition in Eq. 4, any nontrivial finite language L on Σ is uniquely decomposable into $c = head(L), L_1 = onset(L)$, and $L_0 = offset(L)$ such that $c \prec_\Sigma head(L_0)$ and $L_1 \neq \emptyset$. (2) Conversely, if $L = (c \cdot L_1) \cup L_0$ for some c, L_1, and L_0 such that $c \prec_\Sigma head(L_0)$ and $L_1 \neq \emptyset$, then $head(L) = c$, $onset(L) = L_1$, and $offset(L) = L_0$.*

Proof. (1) It is easy from definition. (2) From the implication $L_1 \neq \emptyset$, which implies $c \cdot L_1 \neq \emptyset$, we have $head(L) = c$. Since $c \prec_\Sigma head(L_0)$ implies $c \cdot L_1 \cap L_0 = \emptyset$, we have $onset(L) = L_1$ and $offset(L) = L_0$. This shows the lemma. □

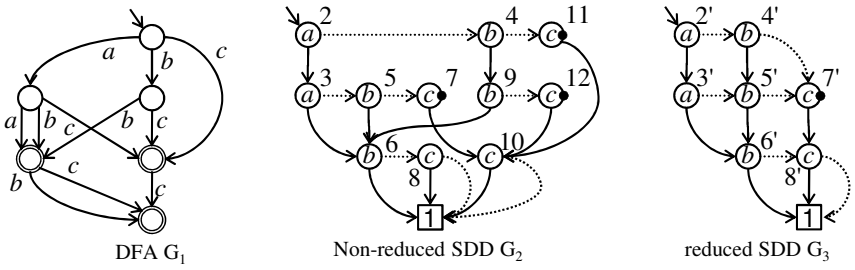

Figure 4: A minimal acyclic DFA G_1 (left), a non-reduced sequence BDD G_2 (middle), and a reduced sequence BDD G_3 (right) on $\Sigma_1 = \{a, b, c\}$ for the same finite language L_1 in Example 1. The numbers at each node indicate vertex id. The 0-terminal **0** is omitted, and and all edges incoming to **0** are indicated by a small black dot.

Clearly, we see that a sequence BDD for a language L simulates the binary decomposition of L by using three attributes, *label*, *one*, and *zero*, of each vertex, which correspond to the head, onset, and offset operations, respectively. Now, we give the first semantics of a sequence BDD.

Definition 2 (The first definition of the language). In a sequence BDD G, a vertex v in G denotes a finite language $L_G(v)$ on Σ defined recursively as:

1. If v is a terminal vertex, $L_G(v)$ is the trivial language defined as: (i) if $value(v) = 1$, $L_G(v) = \{\varepsilon\}$, and (ii) if $value(v) = 0$, $L_G(v) = \emptyset$.
2. If v is a nonterminal vertex, $L_G(v)$ is the finite language $L_G(v) = (label(v) \cdot L_G(one(v))) \cup L_G(zero(v))$.

We write $L(v)$ for $L_G(v)$ if the underlying graph G is clearly understood. Moreover, if G is a sequence BDD with the single root r, we write $L(G)$ for $L_G(r)$. We say that G is a *sequence BDD for* L if $L = L(G)$.

2.4. The second semantics

In its structure, a sequence BDD resembles an ordinary BDD, except that the former is ordered in only 0-edges, whereas the latter is ordered in both 1- and 0-edges. However, the semantics of a sequence BDD is very different from that of an ordinary BDD. A sequence BDD represents a set of strings as finite automata, while an ordinary BDD represents a Boolean function as decision trees. Below, we give the second semantics of a sequence BDD, where we interpret a sequence BDD as an NFA.

Definition 3 (The finite automata for a sequence BDD). Let G be a sequence BDD with vertex set V. We interpret G as the nondeterministic finite automaton (NFA, for short) $A(G) = (Q, \Sigma, T, I, F)$, defined as follows. The state set $Q = V$ consists of all vertices of G, and the set $I \subseteq Q$ of initial states consists of all root vertices of G. The 1-terminal **1** of G is the unique final state in $F = \{\mathbf{1}\}$, and the 0-terminal $\mathbf{0} \in Q - F$ is the unique garbage state of $A(G)$. The transition relation $T \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is defined as follows: for each nonterminal vertex v with label $a \in \Sigma$, we regard its 1-edge outgoing to its 1-child $one(v)$ as the letter transition $v \xrightarrow{a} one(v)$, and its 0-edge to its 0-child $zero(v)$ as the ε-transition $v \xrightarrow{\varepsilon} zero(v)$.

The automaton $A(G)$ above is an NFA with the special form that (i) it is acyclic, and (ii) each state other than the final and garbage states has exactly two transitions, one of which is a letter transition and the other an ε-transition. The acceptance of a string $s \in \Sigma^*$ by $A(G)$ is defined in a similar way to ordinary NFAs (See [17]). The second semantics of a sequence BDD is given by its NFA.

Definition 4 (The second definition of the language). For any vertex v, the *language* $L_G(v)$ *defined by* G *at a vertex* $r \in Q = V$ is the set of all strings over Σ that $A(G)$ accepts. If G is single-rooted, we define $L(G) = L_G(r)$ for the unique root r of G.

Since the automata $A(G)$ is non-deterministic, it seems at second glance that it requires exponential time in n to decide the acceptance of a string s of length n by $A(G)$. Fortunately, however, the next lemma says that the decision can be done by the deterministic procedure Decide as shown in Fig. 5.

procedure Decide(v: a vertex, $s = a_1 \cdots a_n$: a string of length $n \geq 0$ on Σ):

1: **for** $(i \leftarrow 1, \ldots, n)$
2: **while** $(label(v) \prec_\Sigma a_i)$ $v \leftarrow zero(v)$;
3: **if** $(a_i \prec_\Sigma label(v))$ **return** "*Reject*";
4: **else** $v \leftarrow one(v)$;
5: **while** $(value(v) = null)$ $v \leftarrow zero(v)$;
6: **if** $(value(v) = 0)$ **return** "*Reject*";
7: **else return** "*Accept*";

Figure 5: The procedure for deciding the acceptance of a string s by a sequence BDD vertex v.

Lemma 2. *The procedure* Decide *in Fig. 5 decides the acceptance of a given string s at vertex r in $O(|\Sigma| \cdot n)$ time.*

Proof. Due to the ordering rule for 0-edges in a sequence BDD, at each state, the while-loop with label checking correctly finds the branching to the next state. We omit the details. □

Example 2. In Fig. 4, we show examples of acyclic DFA G_1, and sequence BDD G_2 and G_3 for the finite language L_1 in Example 1.

Lemma 3. *The first definition and the second definition of the language $L_G(v)$ defined by a sequence BDD at vertex v are equivalent.*

2.5. Reduced sequence BDDs

Let G be a sequence BDD. We write $G(v)$ for the subgraph rooted by a vertex v. We say that two vertices v and v' of G *coincide in their attributes* if either (i) both are terminal and $value(v) = value(v')$, or (ii) both are nonterminal and $triple(v) = triple(v')$. We give the definition of isomorphism for two sequence BDDs as follows.

Definition 5. Sequence BDDs G and G' are *isomorphic*, denoted by $G \approx G'$, if their exists a one-to-one mapping f from the vertices of G onto the vertices of G' such that (1) f maps the root r of G to the root r' of G', i.e., $f(r) = r'$, and (2) for any node v in G, if $f(v) = v'$ then either (2.i) both v and v' are terminal vertices with $value(v) = value(v')$, or (2.ii) both v and v' are nonterminal vertices with $label(v) = label(v')$, $f(zero(v)) = zero(v')$, and $f(one(v)) = one(v')$.

We can decide if given two single-rooted sequence BDDs G and G' are isomorphic in the time proportional to the size of the smaller one.

Definition 6. A sequence BDD G is *reduced* if and only if (a) it contains no vertex v such that $one(v)$ is the 0-terminal vertex (*zero-suppress rule*), and (b) it contains no pair of distinct vertices v and v' that coincide in their attributes (*vertex-sharing rule*).

A sequence BDD can be reduced in size without changing its language by eliminating redundant vertices and duplicate subgraphs as in the binary decision diagraphs of Bryant [4]. In the definition, the sequence BDD G can be either multi-rooted or one-rooted. The above pair of reduction rules for sequence BDDs is identical to the reduction rules for the zero-suppressed BDDs of Minato [29]. The next lemma gives another definition of reduced sequence BDD that is equivalent to the above definition.

Lemma 4. *For any sequence BDD G, (A) G is* reduced *if and only if (B) G satisfies the following conditions: (a) it contains no vertex v such that $one(v)$ is the 0-terminal vertex (*zero-suppress rule*), and (b') it contains no pair of distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic (*subgraph-sharing rule*).*

Proof. (A)⇒(B): To contradict, we assume that (A) holds but (B) does not hold. Then, there are some pair of distinct vertices v and v' such that $G(v) \approx G(v')$. We show the next claim.

(*Claim 1*) If $G(v) \approx G(v')$ for some distinct vertices v and v' of G, there are some distinct vertices u and u' of G, respectively, that coincide in their attributes.

(*Proof for Claim 1*) We prove the claim by induction over the size n of the larger of $G(v)$ and $G(v')$. Let v and v' be distinct vertices such that $G(v) \approx G(v')$. Without loss of generality, we assume that $|G(v)| \leq |G(v')| = n$. Base case: Suppose that $n = 0$. Then, $G(v)$ and $G(v')$ contain only v and v', respectively, and both of them are terminals. Clearly we have $value(v) = value(v')$. Induction case: Suppose that $n > 0$ and the induction hypothesis that the claim holds for all sequence BDDs of sizes at most $n - 1$. Since $G(v) \approx G(v')$, both of the roots v and v' are nonterminal because $|G(v')| = n > 0$. Then, there are two cases below: (I) If $triple(v) = triple(v')$, the claim obviously holds. (II) Otherwise, $triple(v) \neq triple(v')$. Since $label(v) = label(v')$, there are two cases below: In the case (II.i) that $zero(v) \neq zero(v')$. Generally, we observe that for any vertices v and v' if $G(v) \approx G(v')$, then $G(zero(v)) \approx G(zero(v'))$ and $G(one(v)) \approx G(one(v'))$. From this observation, it follows from the assumption that $G(zero(v)) \approx G(zero(v'))$. Since, $|G(zero(v))|$ is less than n, it follows from the induction hypothesis that there are some distinct vertices u and u' that coincide in their attributes. This shows the claim. In the case (II.ii) that $one(v) \neq one(v')$, the claim is shown similarly. (*End of Proof for Claim 1*)

From Claim 1 above, the vertices v and v' have distinct descendants that coincide in their attributes. This contradicts condition (A). (B)⇒(A): Assume that condition (A) does not hold. Without loss of generality, there are vertices v and v' that coincide in their attributes. Then, we can show that $G(v) \approx G(v')$, and the claim immediately follows. □

Below, we show some properties of a reduced sequence BDD.

Lemma 5. *Let G be any reduced sequence BDD. For any vertex v of G, we have the following conditions (1)–(2):*

(1) The vertex v is the 0-terminal if and only if $L(v) = \emptyset$.
(2) If v is nonterminal, we have that $head(L(v)) = label(v)$, $onset(L(v)) = L(one(v))$, and $offset(L(v)) = L(zero(v))$ hold.

Proof. Since G is acyclic, Claim (1) can be shown by induction on the composition of v. From Claim (1), if G is reduced, $L(one(v)) \neq \emptyset$. Thus, Claim (2) immediately follows from (2) of Lemma 1. □

3. Characterization of Minimal Sequence BDDs

In this section, we show the equivalence of a reduced sequence BDD and a minimal sequence BDD. Two single-rooted sequence BDDs G and G' with roots v and v', respectively, are *equivalent* to each other, denoted by $G \equiv G'$, if they define the same language.

Definition 7 (Minimality). A single-rooted sequence BDD G is *minimal* if and only if $|G| \leq |G'|$ for any single-rooted sequence BDD G' such that $L(G) = L(G')$.

For sequence BDDs, the reducedness is a syntactic property of the structure, while the minimality is a semantic property of its language.

Example 3. In Fig. 4, again, we observe that reduced sequence BDD G_3 of size 7 is more compact than non-reduced sequence BDD G_2 of size 11 representing the same language L_1. Actually, G_3 is obtained by merging the pairs of the *equivalent vertices* $\langle v, v' \rangle$ which represent the same language, where $\langle 5, 9 \rangle$, $\langle 7, 12 \rangle$, $\langle 6, 10 \rangle$, and $\langle 11, 12 \rangle$ are such pairs of vertices in G_2.

To show the equivalence, we start with some definitions and lemmas in the style of Myhill-Nerode's theorem [17, 34] for minimal DFAs. For a finite language L on Σ, we define the family $\mathcal{SUB}(L) \subseteq 2^{\Sigma^*}$ of finite languages on Σ, called the family of canonical sublanguages, by the following recurrence:

1. If L is trivial, then $\mathcal{SUB}(L) = \{L\}$.
2. If L is non-trivial, then $\mathcal{SUB}(L) = \{L\} \cup \mathcal{SUB}(onset(L)) \cup \mathcal{SUB}(offset(L))$.

In other words, $\mathcal{SUB}(L)$ is the family of languages obtained by iterative applications of binary decomposition starting from L. From the construction, we see that $\mathcal{SUB}(L)$ consists of at most $\|L\|$ unique sublanguages. The family $\mathcal{SUB}(L)$ contains either $\{\varepsilon\}$ or $\emptyset$, but not necessarily both. For example the finite language $L_2 = \{\varepsilon, a\}$ contains $\{\varepsilon\}$ but not $\emptyset$, while the language $L_3 = \emptyset$ is the only finite language containing $\emptyset$ itself. We denote by $\mathcal{SUB}_*(L) = \mathcal{SUB}(L) - \{\{\varepsilon\}, \emptyset\}$ the subfamily consisting only of non-trivial languages in $\mathcal{SUB}(L)$.

Now, we give the canonical sequence BDD of a language as follows.

Definition 8 (Canonical sequence BDD). A single-rooted sequence BDD G for a finite language $L = L(G)$ having the vertex set V is *canonical* if there is a bijection $\phi : V \to \mathcal{SUB}(L)$ satisfying the following conditions:

- For the root r of G, $\phi(r) = L$.
- For terminals, $\phi(\mathbf{1}) = \{\varepsilon\}$ and $\phi(\mathbf{0}) = \emptyset$.
- For any nonterminal vertex $v \in V$, $\phi(v) = (\mathit{label}(v) \cdot \phi(\mathit{one}(v))) \cup \phi(\mathit{zero}(v))$.

Since the family $\mathcal{SUB}(L)$ is well-defined, finite, and of cardinality at most $\|L\|$, we can easily see that a canonical sequence BDD $G_*(L)$ for L is well-defined and has size equal to $|\mathcal{SUB}_*(L)| \leq \|L\|$.

Lemma 6. *If G_* is canonical sequence BDD for L, then $L_{G_*}(v) = \phi(v)$ for any vertex v.*

We can easily see that a canonical sequence BDD G_* for L is unique up to isomorphism, and thus denote it by $G_*(L)$, since the structure of G_* is completely determined by the language L itself through $\mathcal{SUB}(L)$. The next lemma shows a relationship between a reduced sequence BDD and a canonical BDD.

Lemma 7. *Any reduced sequence BDD G is isomorphic to the canonical sequence BDD for its language. That is, $G \approx G_*(L(G))$.*

Proof. This follows immediately from Lemma 5 and Lemma 6. □

From Lemma 7 above, we see that a reduced sequence BDD G and the canonical sequence BDD for its language have the same size. The next lemma is crucial to our main result.

Lemma 8. *Let G be any possibly non-reduced sequence BDD. For any canonical sublanguage $L' \in \mathcal{SUB}(L(G))$, there exists some vertex v of G such that $L(v) = L'$.*

Proof. Let v be the root of G and let $L = L(G) = L(v)$. By induction on the composition of G, we show the claim. Base case: If G consists of only a terminal, say v, then $L(v)$ is a trivial language, and thus we know that the terminal v represents L'. Induction case: We have $L(v) = (\mathit{label}(L) \cdot L(\mathit{one}(v))) \cup L(\mathit{zero}(v))$. There are two cases on $L(\mathit{one}(v))$:

(a) Suppose that $L(\mathit{one}(v)) \neq \emptyset$. From (2) of Lemma 1, we have $L' \in \mathcal{SUB}(L) = \{L\} \cup \mathcal{SUB}(\mathit{onset})(L) \cup \mathcal{SUB}(\mathit{offset}(L)) = \{L\} \cup \mathcal{SUB}(L(\mathit{one}(v))) \cup \mathcal{SUB}(L(\mathit{zero}(v)))$. (Case i) $L' \in \{L\}$: Trivially v represents L'. (Case ii) $L' \in \mathcal{SUB}(L(\mathit{one}(v)))$: From (2) of Lemma 1, $\mathit{onset}(L) = L(\mathit{one}(v))$. If we take the subgraph G_1 rooted at $v_1 = \mathit{one}(v)$, then we can show by induction hypothesis that some vertex in G_1 represents L'. (Case iii) $L' \in \mathcal{SUB}(L(\mathit{zero}(v)))$: By similar argument to (Case ii), we can show that some vertex in G_0 represents L'.

(b) Suppose that $L(\mathit{one}(v)) = \emptyset$. Since $L(v) = L(\mathit{zero}(v))$ in this case, we have $L' \in \mathcal{SUB}(L) = \{L(v)\} \cup \mathcal{SUB}(L(\mathit{zero}(v)))$. By similar argument to (a), we can show the claim. Hence, the result follows. □

Now, we have the main theorem of this section.

Theorem 1 (Characterization of minimal sequence BDDs). *For any single-rooted sequence BDD G with the language $L = L(G)$, the following (1)–(3) are equivalent to each other.*

(1) G is a reduced sequence BDD.
(2) G is a canonical sequence BDD.
(3) G is a minimal sequence BDD.

Table 1: Basic operations on finite languages represented as sequence BDDs G, G_1, and G_2 for languages L, L_1, and L_2, respectively.

Procedure	Result	Time Complexity
0	The 0-terminal vertex **0**	$O(1)$
1	The 1-terminal vertex **1**	$O(1)$
Getvertex(c, v_0, v_1)	A vertex with a given triple	$O(1)$
Reduce(G)	G reduced to canonical form	$O(\|G\|)$
Meld(G_1, G_2)	$L_1 \langle op \rangle L_2$ for $\langle op \rangle \in \{\cup, \cap, -, \oplus\}$	$T_{Meld}(G_1, G_2) = O(\|G_1\|\|G_2\|)$
Concat(G_1, G_2)	$L_1 \cdot L_2$	$O(\|G_1\| \cdot 2^{\|G_2\|})$
MakeString(s)	A sequence BDD for a string s	$O(\|s\|)$
AddString(G, s)	$L \cup \{s\}$	$O(\|\Sigma\| \cdot \|s\|)$
DeleteString(G, s)	$L - \{s\}$	$O(\|\Sigma\| \cdot \|s\|)$
equals(G_1, G_2)	$L(G_1) = L(G_2)$?	$O(1)$
isContained(G_1, G_2)	$L(G_1) \subseteq L(G_2)$?	$O(T_{Meld}(G_1, G_2))$
Print-one(G)	some string of $L(G)$	$O(\mathsf{maxlen}(L(G)))$
Print-all(G)	$L(G)$	$O(\mathsf{maxlen}(L(G))\|L(G)\|)$
Count(G)	$\|L(G)\|$	$O(\|G\|)$

Proof. The proof of (1) $\Rightarrow$ (2): It is immediate from Lemma 7. The proof of (2) $\Rightarrow$ (3) is by contradiction: Suppose that there exists some sequence BDD G' such that $L(G) = L(G')$ and $|G'| < |G|$. Since the vertices of G are mutually distinct languages in $\mathcal{SUB}(L)$, this implies $|G'| < |G| = |\mathcal{SUB}(L)|$. Thus, it follows from Lemma 8 that if $|G'| < |\mathcal{SUB}(L)|$ then $M_1 = L_G(u) = M_2$ for some vertex u in G and for $M_1, M_2 \in \mathcal{SUB}(L)$ such that $M_1 \neq M_2$; Contradiction. The proof of (3) $\Rightarrow$ (1): If G is not reduced, then there exists some vertex that violates either the subgraph-sharing rule or the zero-suppress rule. In either case, we can remove the redundant vertex from G without changing its language $L(G)$. This shows that G is not minimal, and thus, the result is proved. □

4. Operations

4.1. Shared sequence BDDs

We can use a multi-rooted sequence BDD G as a persistent data structure for storing and manipulating a collection of more than one set of strings on an alphabet Σ. In an environment, we can create a new subgraph by combining one or more existing subgraphs in G in an arbitrary way. As an invariant, all subgraphs in G are maintained as minimal.

A *shared sequence BDD environment* is a 3-tuple $\mathcal{E} = (G, uniqtable, cache)$ consisting of a multi-rooted sequence BDD G with a vertex set V and two hash tables *uniqtable* and *cache*, explained below. If two tables are clearly understood from context, we identify $\mathcal{E}$ and the underlying graph G by omitting tables.

The first table *uniqtable*, called the *unique vertex table*, assigns a nonterminal vertex $v = uniqtable(c, v_0, v_1)$ of G to a given triple $\tau = \langle c, v_0, v_1 \rangle$ of a symbol and a pair of vertices in G. This table is maintained such that it is a function from all triples τ to the nonterminal vertex v in G such that $triple(v) = \tau$. If such a node does not exist, *uniqtable* returns *null*.

The second table *cache*, called the *operation cache*, is used for a user to memorize the invocation pattern "$op(x_1, \ldots, x_k)$" of a user-defined operation *op* and the associated return value $u = op(v_1, \ldots, v_k)$, where each v_i, $i = 1, \ldots, n$ is either a symbol or an existing vertex in G.

We assume that the above hash tables *uniqtable* and *cache* are global variables in $\mathcal{E}$, and initialized to the empty tables when $\mathcal{E}$ is initialized.

4.2. Operations

We view a symbolic manipulation program as executing a sequence of commands that build up representations of languages and that determine various properties about them. For example, suppose we wish to construct the representation of the language computed by a data mining program. At this point we can test various properties of

Global variable: *uniqtable*: hash table for attribute triples.

Proc Getvertex(c: symbol, v_0, v_1: vertices in V):

1: **if** ($v_1 = \mathbf{0}$) **return** v_0; /* *zero-suppress rule* */
2: **else if** (($v \leftarrow$ *uniqtable*$[\langle c, v_0, v_1\rangle]) \neq$ *null*) **return** v; /* *subgraph-sharing rule* */
3: **else if** ($c \prec_\Sigma$ *label*(v_0))
4: Create a fresh nonterminal v in V;
5: $\tau(v) \leftarrow \langle c, v_0, v_1\rangle$;
6: *uniqtable*$[\langle c, v_0, v_1\rangle] \leftarrow v$;
7: **return** v;
8: **else error** "*Wrong argument. Not 0-ordered!*";

Figure 6: The procedure Getvertex for constructing a vertex with a given triple as attribute, where a hash table *uniqtable* from triples of a symbol and a pair of vertices to vertices is assumed.

the language, such as whether it equals $\emptyset$ or $\{\varepsilon\}$, or whether it equals the language denoted by some other expression (equivalence). We can also ask for information about the strings in the language, such as to list some member, to list all member, to test some string for membership, etc.

In this section we will present algorithms to perform basic operations on sets of strings represented as sequence BDD as summarized in Table 1. These few basic operations can be combined to perform a wide variety of operations on sets of strings. In the table, the language L is represented by a reduced sequence BDD G containing $|G|$ vertices. Similarly the languages L_1 and L_2 are represented by reduced sequence BDDs G_1 and G_2 respectively, containing $|G_1|$ and $|G_2|$ vertices. Our algorithms utilize techniques commonly used in BDD and ZDD algorithms such as ordered traversal, table look-up and vertex encoding. As the table shows, most of the algorithms have time complexity proportional to the size of the sequence BDDs being manipulated. Hence, as long as the languages of interest can be represented by reasonably small sequence BDD, our algorithms are quite efficient.

4.3. On-the-fly minimization algorithm

In the construction of a finite language in our shared sequence BDD environment, we construct a new subgraph from an existing graph G by incrementally adding new vertices, each having specific attributes. During this process, it is crucial to ensure that the constructed subgraph is always reduced. We denote by $\mathbf{1}$ (or $\mathbf{0}$, resp.) the smallest sequence BDD consisting of the 1-terminal (or the 0-terminal, resp.) only.

In Sec. 2.3, we saw that any finite language other than $\{\varepsilon\}$ and $\emptyset$ can be decomposed by binary decomposition as

$$L = (\mathit{head}(L) \cdot \mathit{onset}(L)) \ \cup \ \mathit{offset}(L).$$

Conversely, whenever $\mathit{onset}(L) \neq \emptyset$ holds, we can reconstruct L from its components $\mathit{head}(L)$, $\mathit{onset}(L)$, and $\mathit{offset}(L)$.

In Fig. 6, we show the procedure Getvertex that can be used to construct incrementally a sequence BDD. It assumes that a sequence BDD, G, is in place, and receives a tuple $\langle c, v_0, v_1\rangle$ which consists of a symbol and two vertices in G. If $v_1 = \mathbf{0}$, it returns v_0. If G is already reduced this ensures consistency with the zero-suppress rule, so that G remains reduced. If a vertex, v, such that $\tau(v) = \langle c, v_0, v_1\rangle$, is already in G, then Getvertex returns v, thereby complying with the subgraph-sharing rule and thus ensuring that if G is reduced, it remains reduced. Alternatively, it inserts and returns a fresh non-terminal, v with attributes $\langle c, v_0, v_1\rangle$ into G, provided that $c \prec_\Sigma \mathit{label}(v_0)$. However, if $c \prec_\Sigma \mathit{label}(v_0)$ is false, an error message is issued. The next lemma follows directly:

Lemma 9. *Let G be a reduced sequence BDD, and let $\tau = \langle c, v_1, v_0\rangle$ be a tuple consisting of a symbol in Σ and two existing vertices in G, such that $c \prec_\Sigma \mathit{label}(v_0)$. If $v =$ Getvertex(c, v_1, v_0) then $G' = G \cup \{v\}$ is a reduced sequence BDD such that $L_{G'}(v) = (c \cdot L_G(v_1)) \cup L_G(v_0)$.*

If we use ordinary hash tables, then Getvertex can be implemented in $O(1)$ expected time and $O(N)$ words of space for storing N vertices. If we use a dynamic dictionary (Beame and Fich [2]), then Getvertex can be implemented in $O((\log\log N)^2)/\log\log\log N)$ worst case time and $O(N)$ words of space.

Global variable: *cache*: hash table for operations.

Algorithm Reduce(v : vertex of a sequence BDD):
Output: The root u of the minimal sequence BDD for $L(v)$;

1: **if** ($v = \mathbf{1}$ or $v = \mathbf{0}$) **return** v;
2: **else if** (($u \leftarrow$ *cache*["*Reduce*(v)"]) exists) **return** u;
3: **else** $u \leftarrow$ Getvertex(*label*(v), Reduce(*zero*(v)), Reduce(*one*(v)));
4: *cache*["*Reduce*(v)"] $\leftarrow u$;
5: **return** u;

Figure 7: An off-line minimization algorithm for a sequence BDD.

The above lemma indicates that we can construct any reduced sequence BDD from an existing one. As long as a user uses *Getvertex* to incrementally insert vertices, the minimality of constructed subgraphs is ensured. As another advantage, a shared sequence BDD environment allows a user to access any version of a sequence BDD at any point due to the purely functional, write-only nature of update. Hence, we can say that it is a fully persistent data structure [20] for sets of variable-length strings.

4.4. Linear-time minimization algorithm

Using the algorithm Getvertex in the previous subsection, we show in Fig. 7 an efficient off-line minimization algorithm Reduce for sequence BDDs. Starting from the root of an input sequence BDD P, the algorithm Reduce recursively computes the reduced sequence BDD $P*$ equivalent to P in a top-down manner. The time complexity of the algorithm is linear in $|P|$ since it visits $O(|P|)$ vertices making expected constant-time look-up to hash tables. Thus, we have the following theorem.

Theorem 2 (Linear-time reduction). *The algorithm* Reduce *of Fig. 7 computes the reduced version of an input sequence BDD of size N rooted at vertex v in $O(n)$ expected time and space in the input size $n = |G(v)|$ words of space if we use hash table, and in $O((\log\log N)^2)/\log\log\log N)$ worst case time and $O(N)$ words of space if we use dynamic dictionary of (Beame and Fich [2]).*

Corollary 3 (Complexity of off-line minimization). *In the best case, minimization problem for sequence BDDs is solvable in linear expected time. In the worst case, it is solvable in $O(N(\log\log N)^2)/\log\log\log N)$ time using linear words of space, where N is the size of a sequence BDD. The run time is therefore independent of $|\Sigma|$.*

5. Input- and Output-Sensitive Time-bounds for Boolean Set Operations

In this section, we study the time complexity of Boolean set operations for sequence BDDs. We present an efficient algorithm $\mathsf{Meld}_\diamond$ for eight Boolean set operations $\diamond$ by generalizing the algorithm in [24] based on a recursive synthesis algorithm Apply for Boolean operations by Bryant [4] in the style of Knuth [22].

5.1. The melding algorithm for Boolean set operations

We define the melding operations as follows. Let $\mathsf{op}_{meld} = \{\cup, \cap, \setminus, /, \oplus, \emptyset, LHS, RHS\} \subseteq \mathsf{op}$ be a set of operations. For all operations $\diamond \in \mathsf{op}_{meld}$, the *terminal operation table* $F_\diamond$ is the Boolean function $F_\diamond : \{0,1\}^2 \to \{0,1\}$ such that $F_\diamond[0,0] = 0$, which is defined as: $F_\cup[x,y] = x \vee y$, $F_\cap[x,y] = x \wedge y$, $F_\setminus[x,y] = x \wedge \neg y$, $F_/[x,y] = \neg x \wedge y$, $F_\oplus[x,y] = x \oplus y$ (exclusive-or), $F_\emptyset[x,y] = 0$, $F_{LHS}[x,y] = x$, $F_{RHS}[x,y] = y$, where $x, y \in \{0,1\}$. For any sequence BDD P, we define sign(P) to be 0 if $P = \mathbf{0}$, and 1 otherwise. For binary set operations for finite languages, $F[0,0]$ must be 0. Since $F[0,0] = 1$ means that the strings that are not included in both inputs will be contained by the output, the output language becomes infinite.

Global variable: *cache*: hash table for operations.

Algorithm $\mathsf{Meld}_\diamond(u, v$: vertices):
Output: The reduced sequence BDD for $L(u) \diamond L(v)$, where $\diamond$ is the Boolean set operation specified by a given terminal operation table $F_\diamond : \{0,1\}^2 \to \{0,1\}$;

```
1:  if (u = 0 or v = 0 or u = v)
2:      if (F_⋄[sign(u), sign(v)] = 0) return 0; /* See Sec. 5.1 for F_⋄. */
3:      else if u ≠ 0 return u;
4:      else if v ≠ 0 return v;
5:  else if ((w ← cache["Meld_⋄(u, v)"]) exists) return w;
6:  else
7:      if (label(u) ≺_Σ label(v)) w ← Getvertex(label(u), Meld_⋄(zero(u), v), Meld_⋄(one(u), 0));
8:      else if (label(v) ≺_Σ label(u)) w ← Getvertex(label(v), Meld_⋄(u, zero(v)), Meld_⋄(0, one(v)));
9:      else if (label(u) =_Σ label(v)) w ← Getvertex(label(u), Meld_⋄(zero(u), zero(v)), Meld_⋄(one(u), one(v)));
10:     cache["Meld_⋄(u, v)"] ← w;
11:     return w;
```

Figure 8: An algorithm $\mathsf{Meld}_\diamond$ for Boolean string set operations, where $\diamond \in \{\cup, \cap, \oplus, \setminus, /, \emptyset, LHS, RHS\}$ and function $F_\diamond$ are defined in Sec. 5.1.

Definition 9 (Melding operations). For every $\diamond \in \mathsf{op}_{meld}$, the *melding* operation, also denoted by $\diamond$, is the binary operation $\diamond : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$ on string sets defined by

$$L_1 \diamond L_2 = \{\, x \in \Sigma^* \mid F_\diamond([x \in L_1], [x \in L_2]) = 1 \,\},$$

where $L_1, L_2 \subseteq \Sigma^*$ are any string sets on Σ, and $[prop]$ is the indicator function for a proposition $prop$, which returns 1 if $prop$ is true and 0 otherwise.

In Fig. 8, we give the algorithm $\mathsf{Meld}_\diamond(P, Q)$ for computing the reduced sequence BDD R such that $L(R) = L(P) \diamond L(Q)$ from reduced sequence BDDs P and Q, assuming a terminal operation table $F_\diamond$. Although the trivial operations are $\emptyset$, LHS and RHS can be computed in constant time without $\mathsf{Meld}_\diamond$; however, these are also uniformly described as $\mathsf{Meld}_\diamond$. From a similar discussion in [22] and by induction on top-left decomposition, we can show the correctness of $\mathsf{Meld}_\diamond$.

In the following complexity analysis of $\mathsf{Meld}_\diamond$, we assume that $\mathsf{Getvertex}$ takes at most the expected constant-time in upper bounds and requires at least worst-case constant-time per operation in lower bounds, as usual.

5.2. Input-Sensitive Complexity of Binary Synthesis

First, we start with input-sensitive analysis of the time complexity of $\mathsf{Meld}_\diamond$ of Fig. 8. A key is to estimate the number of recursive calls under tabulation. Let us denote by $\mathsf{Meld}^0_\diamond$ and $\mathsf{Meld}^1_\diamond$ the first and second parts of $\mathsf{Meld}_\diamond$, i.e., the top level if-clause and else-clause for Lines 1 to 5 and Lines 6 to 11, respectively. Let $\#op$ denote the number of times that procedure op is called during the computation of $\mathsf{Meld}_\diamond(P, Q)$. We assume that $|\mathbf{0}| = |\mathbf{1}| = 1$ for convenience.

Theorem 4 (Input complexity of melding). *Let $\diamond \in \mathsf{op}_{meld}$. For reduced sequence BDDs P and Q, the algorithm $\mathsf{Meld}_\diamond$ of Fig. 8 computes the reduced sequence BDD R such that $L(R) = L(P) \diamond L(Q)$ in $O(|P| \cdot |Q|)$ expected time and space.*

Proof. Consider the computation of $\mathsf{Meld}_\diamond(P, Q)$. Since the arguments P' and Q' of any subroutine call $\mathsf{Meld}_\diamond(P', Q')$, resp., are subgraphs of P and Q, the number of *distinct* calls for $\mathsf{Meld}_\diamond(P, Q)$ is at most $|P| \cdot |Q|$ (*Claim 1*). It also follows that *cache* has $O(|P| \cdot |Q|)$ entries. Since the table-lookup with *cache* at Line 5 eliminates duplicated calls, the $\mathsf{Meld}^1_\diamond$ can be executed at most once for each (P', Q'), and thus, we have $\#\mathsf{Meld}^1_\diamond \le |P| \cdot |Q|$ (*Claim 2*). We observe that $\mathsf{Meld}_\diamond$ is called either (i) at the top-level or (ii) within $\mathsf{Meld}^1_\diamond$. Since exactly one of Line 7, 8, and 9 is executed in $\mathsf{Meld}^1_\diamond$, which contains at most two calls for $\mathsf{Meld}_\diamond$, we have $\#\mathsf{Meld}_\diamond \le 2 \cdot \#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 2 and 3,

we have that $\#\mathsf{Meld}_\diamond \le 2\cdot|P|\cdot|Q|+1 = O(|P|\cdot|Q|)$. If each call of $\mathsf{Meld}_\diamond$ takes $O(1)$ time, then the time complexity is $O(|P|\cdot|Q|)$. On the other hand, each $\mathsf{Meld}_\diamond(P',Q')$ makes exactly one call for Getvertex by adding a new node. Thus, the algorithm adds at most $|R| \le \#\mathsf{Getvertex} \le \#\mathsf{Meld}_\diamond = O(|P|\cdot|Q|)$ nodes. Since the number of *cache*-entries is $O(|P|\cdot|Q|)$ and the function stack has depth no more than $\#\mathsf{Meld}_\diamond$, the space complexity is $O(|P|\cdot|Q|)$. □

From the proof of the above theorem, we have the following corollary.

Corollary 5. *For any melding operation $\diamond \in O$, the reduced output size $|R|$ is bounded from above by $O(|P|\cdot|Q|)$.*

5.3. Pseudo Output Sensitive Complexity of Binary Synthesis

Next, we present an output-sensitive analysis of the time complexity of the melding in the style of Wegener [38], who analyzed the time complexity of Boolean operations for BDDs based on the size of non-reduced BDDs. Let R be the reduced sequence BDD such that $L(R) = L(P) \diamond L(Q)$. Then, we define R^* as the (possibly non-reduced) sequence BDD for $L(P) \diamond L(Q)$ computed as a non-reduced output by $\mathsf{Meld}_\diamond$ equipped with the modification of Getvertex by removing Line 1 and 2 of Fig. 6 for zero-suppress and subgraph-sharing rules. We call this modified version $\mathsf{Getvertex}^*$ By construction, $L(R^*) = L(R)$ holds. Clearly, the non-reduced output size $|R^*|$ is bounded from above by $O(|P|\cdot|Q|)$.

Theorem 6 (Output-sensitive complexity w.r.t. non-reduced output). *The reduced sequence BDD for the reduced sequence BDD R such that $L(R) = L(P) \diamond L(Q)$ can be computed in $O(|R^*|)$ expected time and space by the algorithm $\mathsf{Meld}_\diamond$ in Fig. 8, where R^* is the possibly non-reduced sequence BDD for $L(P) \diamond L(Q)$ defined above.*

Proof. Consider the computation of $\mathsf{Meld}_\diamond$ of Fig. 8 equipped with $\mathsf{Getvertex}^*$. Since each call of $\mathsf{Getvertex}^*$ increases the output size by at least one, we have $\#\mathsf{Getvertex}^* \le |R^*|$ (*Claim 4*). Since exactly one of Line 7, 8, and 9 is executed in $\mathsf{Meld}^1_\diamond$ and it contains at least one call for Getvertex, we have $\#\mathsf{Meld}^1_\diamond \le \#\mathsf{Getvertex}^*$ (*Claim 5*). From the proof for Theorem 4, we have $\#\mathsf{Meld}_\diamond \le 2\cdot\#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 3, 4, and 5 above, we now have $\#\mathsf{Meld}_\diamond \le 2\cdot\#\mathsf{Meld}^1_\diamond + 1 \le 2\cdot\#\mathsf{Getvertex}^* + 1 \le 2\cdot|R^*| + 1 = O(|R^*|)$. and thus, we have the time complexity $O(|R^*|)$. Since *uniqtable* and *cache* contain at most $\#\mathsf{Getvertex}^*$ and $\#\mathsf{Meld}_\diamond$ entries, resp., the space complexity follows from an argument similar to the proof for Theorem 4. □

5.4. A Lower Bound for the Time Complexity of Binary Synthesis

In the BDD community, there has been a strong belief that the quadratic input-sensitive complexities of the binary synthesis procedures for variants of BDDs, including BDDs and ZDDs, are output-linear time for most input instances, and there has been no super-linear lower bound for its time complexity. Recently, Yoshinaka et al. [39] showed that this conjecture is not true for BDDs and ZDDs. They constructed an infinite sequence of input BDDs that demonstrates the quadratic lower bound for the time complexity of the melding for BDDs and ZDDs. Based on their discussion, below we show that the quadratic input-sensitive complexity of the melding in terms of input size is optimal in reality.

Theorem 7. *Let $\diamond$ be any melding operations. The algorithm $\mathsf{Meld}_\diamond$ of Fig. 8 requires $\Omega(|P|\cdot|Q|)$ time and space regardless of the output size, where P and Q are the input sequence BDDs.*

Proof. Our example that the binary synthesis takes $O(|P|\cdot|Q|)$ time to compute $R = P \diamond Q$, where $|R|$ is linear in $|P|+|Q|$, is just a straightforward translation of that in [39]. The theorem can be shown in a way similar to that in [39]. Here, we give a rough sketch of the proof. Let $\Sigma = \{0,1\}$. For a fixed positive integer n, we define

$$M = \{\, a_1 b_1 \dots a_n b_n c_1 \dots c_m \in \{0,1\}^{2n+m} \mid a_{\beta(c_1\dots c_m)} = 1 \,\},$$
$$L = \{\, a_1 b_1 \dots a_n b_n c_1 \dots c_m \in \{0,1\}^{2n+m} \mid b_{\beta(c_1\dots c_m)} = 1 \,\},$$

where $m = \lceil \log n \rceil$ and

$$\beta(c_1 \dots c_m) = \begin{cases} 1 + \sum_{k=1}^{m} 2^{k-1} c_k & \text{if } \sum_{k=1}^{m} 2^{k-1} c_k < n, \\ 1 & \text{otherwise.} \end{cases}$$

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

Algorithm AddString(v, s) $\equiv$ Meld$_\cup(v,$ MakeString$(s))$;

Algorithm DeleteString(v, s) $\equiv$ Meld$_\setminus(v,$ MakeString$(s))$;

Procedure MakeString($s \in \Sigma^*$: string):
Output: The root of the reduced sequence BDD for a string s in linear form;
  if $(s = \varepsilon)$ **return 1**;
  else return Getvertex$(s[1], \mathbf{0},$ MakeString$(s[2] \cdots s[|s|]))$;

Figure 9: The algorithms AddString and DeleteString that compute the reduced sequence BDD formed by dynamically adding and deleting a single string s to/from a given sequence BDD P, where the subprocedure MakeString constructs the reduced sequence BDD in linear form representing s.

We have

$$M \diamond L = \{\, a_1 b_1 \dots a_n b_n c_1 \dots c_m \in \{0,1\}^{2n+m} \mid F_\diamond[a_{\beta(c_1 \dots c_m)}, b_{\beta(c_1 \dots c_m)}] = 1 \,\}.$$

Let P and Q be the reduced sequence BDD for M and L, resp.

We first show that $|P|, |Q|, |R| = O(2^n)$. It is easy to see that every vertex in P and Q represents a set of strings of a fixed length, since all strings in M and L have the same length $2n+m$. We define the *level* of a vertex to be $2n+m-k$ if the vertex represents a set of strings of length k. Since the membership of $a_1 b_1 \dots a_n b_n c_1 \dots c_m$ in M does not depend on any of b_i, it is not hard to see that there are at most $O(2^k)$ vertices of level $2k$ for $0 \le k < n$. The number of vertices of level $2k+1$ is at most twice as big as that of level $2k$. On the other hand, since there are at most 2^{2^k} distinct sets of strings of length k, there are at most $|\Sigma| \cdot 2^{2^k}$ vertices of level $2n+m-k$ for $0 \le k \le m = \lceil \log n \rceil$. All in all, $|P| = O(2^n)$. Similarly $|Q| = O(2^n)$. It is easy to see that, for any $a_i, b_i, a'_i, b'_i \in \{0,1\}$ such that $F_\diamond[a_i, b_i] = F_\diamond[a'_i, b'_i]$, we have $x_1 a_i b_i x_2 \in M \diamond L$ iff $x_1 a'_i b'_i x_2 \in M \diamond L$ for any $x_1 \in \{0,1\}^{2k}$, $x_2 \in \{0,1\}^{2n+m-k-2}$ with $k < n$. Hence, we have $|R| = O(2^n)$ by a discussion similar to that for $|P|, |Q| = O(2^n)$.

Second, we show that #Meld$_\diamond \ge 2^{2n}$. For $x \in \{0,1\}^{2n}$, let P_x denote the vertex such that $L(P_x) = \{\, x' \mid xx' \in M \,\}$. In fact, P has such a vertex for each x. Similarly, let Q_x be such that $L(Q_x) = \{\, x' \mid xx' \in L \,\}$. By definition, Meld$_\diamond(P_x, Q_x)$ is called for each x. Moreover, $P_{a_1 \dots b_n} \ne P_{a'_1 \dots b'_n}$ whenever $a_i \ne a'_i$ for some i and $Q_{a_1 \dots b_n} \ne Q_{a'_1 \dots b'_n}$ whenever $b_i \ne b'_i$ for some i. Therefore, for distinct $x, x' \in \{0,1\}^{2n}$, the $\langle P_x, Q_x \rangle$ and $\langle P_{x'}, Q_{x'} \rangle$ are distinct. This means #Meld$_\diamond \ge 2^{2n}$. □

6. Linear-time On-the-Fly and Off-line Construction

In Fig. 9, using algorithm Meld$_\diamond$, we present efficient on-the-fly construction algorithms, AddString(P, s) and DeleteString(P, s), for string sets, which compute the reduced sequence BDD formed by dynamically adding and deleting a single string s of length m to a given sequence BDD P using the procedure MakeString. Although the upperbound of the running time of Meld$_\diamond$ is quadratic by Theorem 4, we have expected linear-time complexity of AddString as follows.

Theorem 8 (Linear-time dynamic string set construction). *For any reduced sequence BDD P of size n on alphabet Σ and any string $s \in \Sigma^*$ of length m,* AddString *(*DeleteString*, resp.) of Fig. 9 computes the reduced sequence BDD R such that $L(R) = L(P) \cup \{s\}$ ($L(R) = L(P) \setminus \{s\}$, resp.) in $O(|\Sigma|\, m)$ expected time and $O(|\Sigma|\, m)$ additional space.*

Proof. The correctness is obvious from those of MakeString and Meld$_\cup$. We see that the chain-like sequence BDD Q can be built in $O(|m|)$ time by MakeString, and Q satisfies $Q'.0 = \mathbf{0}$ for every nonterminal Q'. We can show that, starting from the roots, Meld$_\cup$ follows the path in P whose labels spell a prefix of string s by making $O(|\Sigma| \cdot |Q|)$ calls of Meld$_\cup$ and adding $O(|\Sigma| \cdot |Q|)$ vertices, where the $|\Sigma|$ factor is used for searching the sibling with a specified label. The whole process takes $O(|\Sigma| \cdot |Q|)$ time, and the result follows. □

From the above theorem, if we apply a sequence of insert and delete operations to a sequence BDD, then the total computation time is proportional to the total length of the input strings. Daciuk *et al.* [9] presented an on-the-fly construction algorithm with an insert for ADFA. Interestingly, we see that our algorithm performs the same

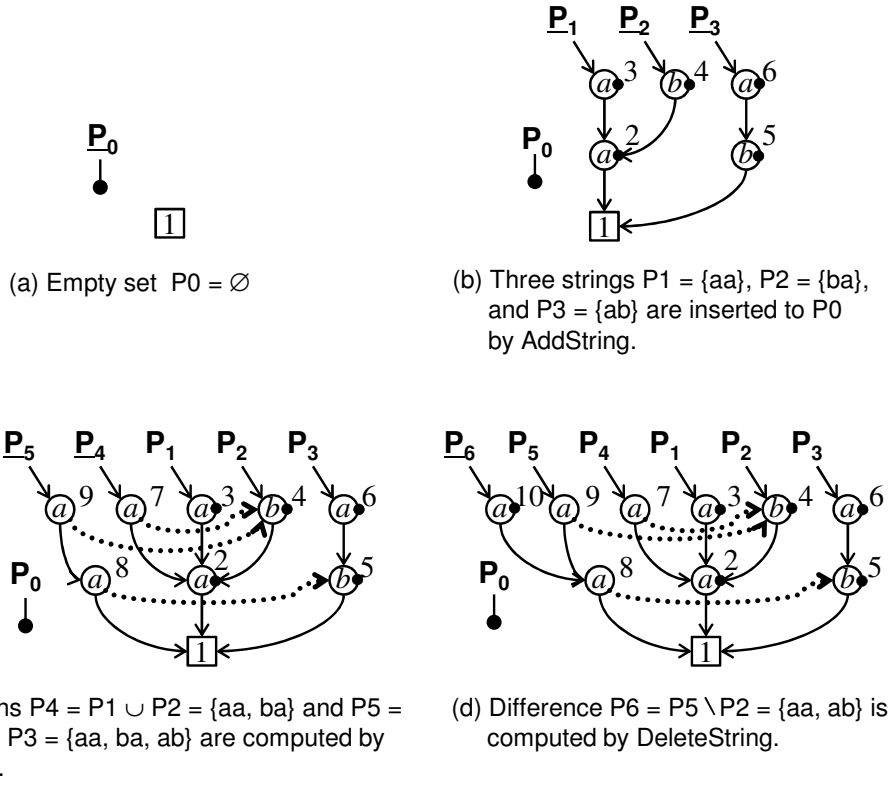

Figure 10: An execution example of AddString, DeleteString, and $\mathsf{Meld}_\cup$ in a shared sequence BDD environment $\mathcal{E}$, where the computation proceeds in write-only manner. In the figure, each circle stands for a vertex, and an associated symbol and a number indicate its vertex label and vertex id, respectively. The terminal **1** is indicated by a square with 1, and **0** is omitted. A small black dot indicates a pointer to **0**.

computation as their algorithm in [9, Algorithm 2], simulating the prefix sharing and subgraph sharing phases of their algorithm by $\mathsf{Meld}_\cup$ and MakeString, respectively, although our algorithm appears to be conceptually much simpler than theirs.

Example 4. Fig. 10 shows an execution example of AddString, DeleteString, and $\mathsf{Meld}_\cup$ in a shared sequence BDD environment $\mathcal{E}$ on $\Sigma = \{a, b\}$. (a) First, we set $P_0 = \mathbf{0}$. (b) By inserting three strings $s_1 = aa$, $s_2 = ba$, and $s_3 = ab$ by $P_i \leftarrow \mathsf{AddString}(P_0, s_i)$ for $i = 1, 2, 3$, we have string sets $L(P_1) = \{aa\}$, $L(P_2) = \{ba\}$, $L(P_3) = \{ab\}$, where aa and ba share a common suffix a in $\mathcal{E}$. (c) By executing $P_4 \leftarrow \mathsf{Meld}_\cup(P_1, P_2)$ and $P_5 \leftarrow \mathsf{Meld}_\cup(P_4, P_3)$, we obtain $L(P_4) = \{aa, ba\}$ and $L(P_5) = \{aa, ba, ab\}$. For P_4, vertex 7 of P_4 is created by copying vertex 3 for $\{aa\}$, and its 0-edge is directed to vertex 4 for $\{ba\}$. (d) Finally, by deleting s_2 from $L(P_5)$ with $P_6 \leftarrow \mathsf{Meld}_\setminus(P_5, P_2)$, we have P_6 for $L(P_6) = \{aa, ab\}$. The above operations are performed in write-only manner without changing any existing vertices.

Next, we present an off-line construction algorithm, Construct, for a sorted string set, which is faster than successive applications of AddString by a factor of $|\Sigma|$. From the proof of Theorem 8, we can show that if s is the lexicographically largest in the string set $L(P)$ of sequence BDD P, $\mathsf{Meld}_\cup$ visits and creates only $O(|s|)$ vertices in P. Now, let us define Construct(L) as the off-line construction algorithm that first sorts strings in L in lexicographic-order and then insert these strings one by one to a sequence BDD $P = \mathbf{0}$ from largest to smallest.

Corollary 9 (Linear-time off-line construction). *For any string set $L \subseteq \Sigma^*$, the algorithm* Construct(L) *computes the reduced sequence BDD for L in $O(n)$ expected time and space, where $n = \|L\|$ and the running time is independent of $|\Sigma|$.*

7. Space-Bounds for Sequence Binary Decision Diagrams and Acyclic Automata

7.1. Finite Automata

We presume a basic knowledge of the automata theory. For a comprehensive introduction to it, see [17, 34] for example. A (partial) *DFA* is represented by a quintuple $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, where Σ is the input alphabet, Γ is the *state set*, δ is the *partial transition function* from $\Gamma \times \Sigma$ to Γ, $q_0 \in \Gamma$ is the *initial state*, and $F \subseteq \Gamma$ is the set of *acceptance states*. The partial function δ can be regarded as a subset $\delta \subseteq \Gamma \times \Sigma \times \Gamma$. We define the *size* of a DFA A, denoted by $|A|$, as the number of labeled edges in A, i.e., $|A| = |\delta|$.

The set of strings that lead the automaton A from a state q to an acceptance state is denoted by $L_A(q)$. The *language* $L(A)$ accepted by A is $L_A(q_0)$. We say that ADFAs A and A' are *equivalent* if $L(A) = L(A')$. A *minimal DFA* has no state q such that $L_A(q) = \emptyset$ and no distinct states q' and q'' such that $L_A(q') = L_A(q'')$. Since we are concerned with finite languages, all DFAs discussed in this section are *acyclic* DFAs (ADFA, for short).

In this section, single-rooted sequence BDDs $B = (V, E)$ are denoted by the tuple $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ for comparison with ADFAs. For convenience, we continue to use the notation $zero(v)$ by $v.lab$, $v.1$, and $v.0$. The sets of nonterminals are denoted by V_N. $\tau : V_N \to \Sigma \times V^2$ is the function that assigns to each $v \in V_N$ the triple $\langle label(v), zero(v), one(v) \rangle$. $\mathbf{r}$ is the root vertex of B. The structure of sequence BDDs apparently resembles that of ADFAs. There is a straightforward translation from an ADFA to a sequence BDD and vice versa. However, we should note subtle differences between these formalisms. In fact, sequence BDDs can be more compact. This section discusses their relationship in detail. Recall that the sizes of ADFA $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$ and sequence BDD $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ are defined as $|A| = |\delta|$ and $|B| = |V_{\mathrm{N}}|$, respectively, where $V_{\mathrm{N}} = V \backslash \{\mathbf{0}, \mathbf{1}\}$. We will compare the total description size, because edges are labeled by a symbol in ADFA, but vertices are labeled in sequence BDD. In what follows, we write $\prec$ for $\prec_\Sigma$ if Σ is clearly understood.

7.2. From ADFAs to sequence BDDs

We first give a straightforward translation from an ADFA to an equivalent sequence BDD, which may be non-reduced, and compare their size.

Theorem 10. *For any ADFA A, there is an equivalent sequence BDD B such that $|B| \leq |A|$. Moreover, for every positive integer $n \geq 1$, there is an ADFA A that admits no equivalent sequence BDD B such that $|B| < |A| = n$.*

Proof. For an ADFA $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, we construct an equivalent sequence BDD $\mathbf{B}(A) = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ as follows. Let $\deg(q) = |\{\, a \in \Sigma \mid \delta(q, a) \text{ is defined} \,\}|$. The set of vertices is given by $V = \{\mathbf{0}, \mathbf{1}\} \cup \{\, [q, i] \mid q \in \Gamma \text{ and } 1 \leq i \leq \deg(q) \,\}$. For each $q \in \Gamma$ with $\deg(q) = k \geq 1$, let $a_1, \ldots, a_k \in \Sigma$ and $q_1, \ldots, q_k \in \Gamma$ be such that $a_1 \prec a_2 \prec \cdots \prec a_k$ and $\delta(q, a_i) = q_i$ for $i = 1, \ldots, k$. Define τ as $\tau([q, i]) = \langle a_i, [q, i+1], \widehat{q_i} \rangle$ if $1 \leq i < k$, $\tau([q, i]) = \langle a_k, \mathbf{1}, \widehat{q_k} \rangle$ if $i = k$ and $q \in F$, and $\tau([q, i]) = \langle a_k, \mathbf{0}, \widehat{q_k} \rangle$ if $i = k$ and $q \notin F$, where $\widehat{q'} = [q', 1]$ if $\deg(q') > 0$, $\widehat{q'} = \mathbf{1}$ if $\deg(q') = 0$ and $q' \in F$, and $\widehat{q'} = \mathbf{0}$ if $\deg(q') = 0$ and $q' \notin F$. The root $\mathbf{r}$ of $\mathbf{B}(A)$ is $\widehat{q_0}$. It is easy to see that $L_A(q) = L_{\mathbf{B}(A)}(\hat{q})$ for all $q \in \Gamma$. We note that the above construction can be accomplished in linear time in $|A|$. The first claim can be verified by the above construction of $\mathbf{B}(A)$. The second claim is established by observing that for any $n \geq 0$, the minimal ADFA, A, that accepts the singleton $\{a^n\}$ has size $|A| = n$, while the reduced sequence BDD for this language, B, is such that $|B| \leq n$. □

We remark that $\mathbf{B}(A)$ in the proof is not necessarily reduced for a minimal ADFA A.

Example 5. Let us compare the minimal ADFA A and the constructed sequence BDD $\mathbf{B}(A)$ for the set $\{ab, b\}$ with $a \prec b$:

Transition rules of A	Corresponding vertices of $\mathbf{B}(A)$
$\delta(q_0, a) = q_1$	$\tau([q_0, 1]) = \langle a, [q_0, 2], [q_1, 1] \rangle$
$\delta(q_0, b) = q_2$	$\tau([q_0, 2]) = \langle b, \mathbf{0}, \mathbf{1} \rangle$
$\delta(q_1, b) = q_2$	$\tau([q_1, 1]) = \langle b, \mathbf{0}, \mathbf{1} \rangle$

$\mathbf{B}(A)$ is not reduced, since $\tau([q_0, 2]) = \tau([q_1, 1])$ for $[q_0, 2] \neq [q_1, 1]$.

In this example, A has two distinct b-labeled edges incident on q_2. These correspond to vertices $[q_0, 2]$ $[q_1, 1]$ in $\mathbf{B}(A)$. The latter two vertices can be merged in a reduced sequence BDD derived from $\mathbf{B}(A)$. This shows that, for the some languages, a reduced sequence BDD can be more compact than the minimal ADFA. We next discuss by how much a sequence BDD can be smaller than an ADFA defining the same language.

7.3. From sequence BDDs to ADFAs

Let a sequence BDD B be given. We construct an ADFA $\mathbf{A}(B)$ such that $L(\mathbf{A}(B)) = L(B)$. We assume that $\mathbf{r} \neq \mathbf{0}$. Otherwise, the translation is trivial.

A key idea of our translation from B to $\mathbf{A}(B)$ is that each state of $\tilde{v}$ of $\mathbf{A}(B)$ corresponds to a vertex v in B that is a 1-child of some other vertex or the root vertex, and all outgoing edges of $\tilde{v}$ correspond to the set of all descendant of v. Formally, for a given sequence BDD $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ and for each nonterminal $v \in V_{\mathrm{N}}$ of B, let $\widetilde{v} = [v_1, \ldots, v_k] \in \Gamma$ be the vertex of $\mathbf{A}(B)$ that corresponds to the sequence $v_1, \ldots, v_k \in V$ of the descendant of v, where (i) $v_1 = v$, (ii) $\tau(v_i) = \langle a_i, v_{i+1}, u_i \rangle$ for every $1 \leq i < k$, and (iii) v_k is $\mathbf{0}$ or $\mathbf{1}$. That is, $\tilde{v}$ is the sequence of vertices that can be obtained by traversing only 0-edges from v to a terminal. Note that $\tilde{\mathbf{1}}$ is $[\mathbf{1}]$, and that Γ does not include $[\mathbf{0}]$ because $\mathbf{0}$ is not pointed by any 1-edges. The $[\mathbf{1}] \in \Gamma$ is the acceptance state without any outgoing edges. Then, the ADFA $\mathbf{A}(B) = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$ is given by

- $\Gamma = \{\widetilde{\mathbf{r}}\} \cup \{\, \widetilde{v_1} \mid v_1 \neq \mathbf{0}$ is the 1-child of some $v \in V_{\mathrm{N}} \,\}$,
- $\delta(\widetilde{v}, a_i) = \widetilde{u_i}$ if $\widetilde{v} = [v_1, \ldots, v_k]$ and $\tau(v_i) = \langle a_i, v_{i+1}, u_i \rangle$,
- $q_0 = \widetilde{\mathbf{r}}$, and
- $F = \{\, \widetilde{v} \in \Gamma \mid \widetilde{v} = [v_1, \ldots, v_k]$ and $v_k = \mathbf{1} \,\}$.

It is easy to see that $L_B(v) = L_{\mathbf{A}(B)}(\widetilde{v})$ for all $\widetilde{v} \in \Gamma$. This implies that if B is reduced, $\mathbf{A}(B)$ is minimal. Contrary to the translation from an ADFA into a sequence BDD, this construction takes $O(|\Sigma| \cdot |B|)$ time. In fact, this is optimal. The next theorem says that a reduced sequence BDD can be about $|\Sigma|$ times more compact than the equivalent minimal ADFA.

Theorem 11. *For any sequence BDD B, one can construct in $O(|\Sigma||B|)$ time the equivalent minimal ADFA $A = \mathbf{A}(B)$ such that (1) $|\Gamma| \leq |B| + 1$ for the state set Γ of A, and (2) $|A|$ satisfies the inequality: $|A| \leq |B|(|B| + 1)/2$ if $|B| \leq |\Sigma|$, and $|A| \leq |\Sigma|(2|B| - |\Sigma| + 1)/2$ if $|B| > |\Sigma|$. Moreover, there is a sequence BDD B that admits no equivalent ADFA A for which the strict inequality holds.*

Proof. Let $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ and $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$. (1) The first claim, $|\Gamma| \leq |B| + 1 = |V_{\mathrm{N}}| + 1$, clearly holds by the conversion. (2) In order to establish the second part of the theorem, we give a variant of the construction of $\mathbf{A}(B)$. We define $\mathbf{C}(B)$ from B by replacing the definition of Γ and F in $\mathbf{A}(B)$ with $\Gamma' = \{\, \widetilde{v} \mid v \in V - \{\mathbf{0}\} \,\}$ and $F' = \{\, \widetilde{v} \in \Gamma' \mid \widetilde{v} = [v_1, \ldots, v_k]$ and $v_k = \mathbf{1} \,\}$, respectively. For $\mathbf{C}(B) = \langle \Sigma, \Gamma', \delta, q_0, F' \rangle$, we prove the inequality by induction on $|B|$. Clearly $\mathbf{A}(B)$ is not bigger than $\mathbf{C}(B)$, and thus this claim implies the theorem. In the following discussion, we ignore the root of B and the initial state of $\mathbf{C}(B)$, because they do not affect the discussion of their description size. For $|B| = 1$, it is easy to see that the claim holds. Suppose that $|B| > 1$. Let B' be the sequence BDD obtained from B by deleting an arbitrary vertex $v \in V_{\mathrm{N}}$ that has no incoming edge. There are two cases. (2.1) If $|B| \leq |\Sigma|$, we have $|\mathbf{C}(B')| \leq (|B| - 1)|B|/2$ by the induction hypothesis. By definition, $\mathbf{C}(B)$ can be obtained from $\mathbf{C}(B')$ by adding one state $\widetilde{v}$ and at most $|B| = |V_{\mathrm{N}}|$ outgoing edges from it. Hence $|\mathbf{C}(B)| \leq |\mathbf{C}(B')| + |B| \leq (|B| - 1)|B|/2 + |B| = |B|(|B| + 1)/2$. (2.2) If $|B| > |\Sigma|$, we have $|\mathbf{C}(B')| \leq |\Sigma|(2|B| - |\Sigma| - 1)/2$ by the induction hypothesis. By definition, $\mathbf{C}(B)$ can be obtained from $\mathbf{C}(B')$ by adding one state $\widetilde{v}$ and at most $|\Sigma|$ outgoing edges from it. Hence, $|\mathbf{C}(B)| \leq |\mathbf{C}(B')| + |\Sigma| \leq |\Sigma|(2|B| - |\Sigma| - 1)/2 + |\Sigma| = |\Sigma|(2|B| - |\Sigma| + 1)/2$. We have proven the inequality. In order to see that the above bound is tight, consider the reduced sequence BDD and the minimal ADFA for the language $L_n = \{\, a_0^k a_{i_1} \ldots a_{i_j} \mid 0 \leq k \leq n - |\Sigma|,\ 0 \leq j \leq \min\{m, n\},\ 1 \leq i_1 < \cdots < i_j \leq m \,\}$ over $\Sigma = \{a_0, \ldots, a_m\}$ with $a_0 \prec a_1 \prec \cdots \prec a_m$. □

For BDDs [4] and ZDDs [29], it is well-known that the choice of ordering $\prec$ on Σ, called *variable ordering*, affects the size of the resulting BDD [4, 22, 29]. In fact, in the proof of Theorem 11, if we take an ordering $\prec'$ on Σ such that $a_m \prec' \cdots \prec' a_1 \prec' a_0$, we have $|A| = |B'|$ for the reduced sequence BDD B' for L_n. Hence, the choice of ordering $\prec$ may affect the size of a sequence BDD. On the other hand, the choices of $\prec$ change the size of sequence BDDs by at most a factor of $|\Sigma|$.

Corollary 1. *For an order π on Σ and a finite language L over Σ, let B_L^π be the reduced sequence BDD for L that respects the order π over Σ. For any ordering π and ρ on Σ, we have $|B_L^\pi| \leq |\Sigma||B_L^\rho|$.*

Proof. Let A be the minimal automaton for L. By Theorems 10 and 11, $|B^\pi| \leq |A| \leq |\Sigma||B^\rho|$. Hence $|B^\pi| \leq |\Sigma||B^\rho|$. □

Through the translation techniques presented above between ADFAs and sequence BDDs and by Theorems 10 and 11, known results on the size of minimal ADFAs can be translated into those on sequence BDDs. A special case is where the set of all factors of a string is concerned. Let $Fact(w) = \{ y \in \Sigma^* \mid w = xyz \text{ for some } x, z \in \Sigma^* \}$ be the set of all factors of a string w. The following theorem is presented in the [3, 6].

Theorem 12 (Blumer et al. [3], Crochemore [6]). *For $w \in \Sigma^*$, let A be the minimal ADFA for $Fact(w)$ with state set Γ. Then, $|\Gamma| \leq 2|w| - 2$ and $|A| \leq 3|w| - 4$ hold.*

The *factor sequence BDD* for a string $w \in \Sigma^*$ is the reduced sequence BDD for all factors in $Fact(w)$, and denoted by $FSDD(w)$. From Theorems 12 and 10, we have the following corollary.

Corollary 2. *For any string $w \in \Sigma^*$, $|w| \leq |FSDD(w)| \leq 3|w| - 4$.*

For $w = cb^n a$ with $a \prec b \prec c$, $|FSDD(w)| = 3|w| - 4$. For an ordering π on Σ, let $FSDD(w)^\pi$ be the corresponding factor sequence BDD for w.

Corollary 3. *For any $w \in \Sigma^*$ and any orderings π and ρ on Σ, $|FSDD(w)^\pi| \leq |FSDD(w)^\rho| + |w| - 1$. Moreover, there are some w, π and ρ for which the equality holds.*

Proof. Let $B^\pi_w = FSDD(w)^\pi$, and A_w be the minimal automaton for $Fact(w)$ with state set Γ_w. We have $|B^\pi_w| \leq |A_w|$ and $|\Gamma_w| \leq |B^\rho_w| + 1$ by Theorems 10 and 11, respectively. Blumer et al. [3, Lemma 1.6] showed that $|A_w| \leq |\Gamma_w| + |w| - 2$. Hence, $|B^\pi_w| \leq |A_w| \leq |\Gamma_w| + |w| - 2 \leq |B^\rho_w| + |w| - 1$. In fact, for $w = a^n b$, $\pi = \langle b \prec a \rangle$, $\rho = \langle a \prec b \rangle$, we have $|B^\pi_w| = 2n - 1$ and $|B^\rho_w| = n - 1$. □

8. Application to Factor Graph Construction from a Graph

To demonstrate the power of sequence BDDs, in this section, we present an application to real string problems represented in sequence BDDs. The problem is constructing a sequence BDD for the language $\{a_i \cdots a_j \mid a_1 \cdots a_n \in L(G), 1 \leq i \leq j \leq n\} \cup \{\epsilon\}$, which is a factor graph, directly from a given sequence BDD G. In this application, sequence BDDs are quite useful and efficient since they allow a simple but practical solution for manipulating large collections of strings in compressed form.

8.1. A practical algorithm for factor sequence BDD construction

The *factor sequence BDD* (*FSDD*) for a finite language $L \subseteq \Sigma^*$ is the reduced sequence BDD for $Fact(L)$ consisting of all factors of strings in L. The FSDD of a sequence BDD G is simply the FSDD for $L(G)$, and is denoted by $FSDD(G)$. Since it is well-known that the size of factor automata for a finite language L is linear in $\|L\|$ [3, 8], the size of $FSDD(G)$ is also linear in $\|L(G)\|$. (See [13] for the tight upperbound for languages that contain only one string.)

Since $|G|$ can be exponentially smaller than $\|L(G)\|$, efficient construction of FSDD from a given sequence BDD is an interesting problem. In Fig. 8.1, we present a simple recursive algorithm, RecFSDD, which computes $FSDD(G)$ from an input sequence BDD G using $\mathsf{Meld}_\cup$ in write-only manner, $\mathsf{Pref}(v)$ is a function that transforms the sequence BDD rooted in v to a modified sequence BDD in which all 0-edges that point to $\mathbf{0}$ are redirected to point to $\mathbf{1}$.

Theorem 13. *The algorithm* RecFSDD *of Fig. 8.1 computes $FSDD(G)$ from an input sequence BDD G in $O(\Sigma m \|L(G)\|)$ time and space, where $m = \max\{|s| \mid s \in L(G)\}$.*

The computation of RecFSDD seems similar to that of the wotd algorithm [14] and other top-down algorithms [32, 36]. In fact, we observed that applications of Pref and $\mathsf{Meld}_\cup$, resp., correspond to the ε-edge attachment and determinization in [32]. Experiments in Sec. 9 showed that RecFSDD ran in almost linear time to the input size $|G|$ on some real data sets.

Global variable: *cache*: hash table for operations.

Proc RecFSDD(v: vertex):

2: **if** ($v = \mathbf{0}$ **or** $v = \mathbf{1}$) **return** v;
3: **else if** ($u \leftarrow$ *cache*["*RecFSDD*(v)"] exists) **return** u;
4: **else** *Pref*(*one*(v)) $\leftarrow$ the sequence BDD for all prefixes of strings in L(*one*(v));
5: $u \leftarrow \mathsf{Meld}_\cup(\mathsf{RecFSDD}(zero(v)), \mathsf{RecFSDD}(one(v)))$;
6: $u \leftarrow \mathsf{Meld}_\cup(u, \mathsf{Getvertex}(label(v), \mathbf{1}, Pref(one(v))))$;
7: *cache*["*RecFSDD*(v)"] $\leftarrow u$;
8: **return** u;

Figure 11: A recursive procedure RecFSDD that constructs the factor sequence BDD of an input sequence BDD vertex v.

Table 2: Outline of data sets.

Dataset							SDD				
name	#letter (byte)	#line	#unique line	line length (byte)		|Σ|	#nodes	#branch			depth
				ave.	max			total	ave	max	
Bible	4,017,009	30,383	30,129	132.212	529	62	3,209,439	3,209,568	1.009	44	601
BibleBi	7,025,414	767,854	154,479	9.149	30	27	168772	208829	2.398	25	90
Ecoli150	4,638,690	30,925	30,910	149.998	150	4	4,212,706	4,212,705	1.007	4	175
Random4	4,500,606	300,000	298,629	15.002	35	4	617,812	696,961	1.647	4	54
Random128	4,500,606	300,000	300,000	15.002	35	128	3,292,878	3,292,957	1.1	128	296
FibSuf(18)	22,885,995	6,766	6,766	3,382.500	6,765	2	6,774	6,782	1.003	2	6773

9. Experiments

In this section, we show the results of experiments on real and artificial data sets that were performed to evaluate the efficiency of the sequence BDDs and related manipulation algorithms presented in the previous sections.

9.1. Experimental setting

In the experiments, we used the following data sets, whose characteristics are shown in Table 2. As real data sets, we used Bible and Ecoli, respectively, a collection of English texts and a single genome sequence obtained from the Canterbury corpus[1]. From these data sets, we obtained the following derived data sets: BibleBi is the set of all word bi-grams drawn from Bible, Ecoli150 is the set of substrings drawn from Ecoli by cutting the whole sequence at every 150-th letter, and BibleFac, BibleBiFac, and EcoliFac are the sets of all factors of the strings in Bible, BibleBi, and Ecoli, respectively. As artificial data sets, we used the following: For $k = 4$ and 128, Random$k$ is the set of random strings uniformly generated on an alphabet of k letters, where the length of each string is determined by Poisson distribution of mean 15; for $m \geq 0$, FibSuf(m) is the set of all suffixes of the m-th Fibonacci word f_m on $\{a, b\}$, where $f_0 = a$, $f_1 = ab$, and $f_m = f_{m-1}f_{m-2}$ ($m \geq 2$). We made subsets of these data sets by randomly taking ℓ lines, varying $\ell = 10, 30, 100, 3000, \ldots$ for Bible, Ecoli, and Random$k$, and $m = 0, \ldots, 18$ for FibSuf.

We implemented two versions of a shared sequence BDD environment including the algorithms Getvertex and Reduce described in Sec. 4, Meld described in Sec. 5, and an off-line construction algorithm using AddString described in Sec. 6. The first version[2] [11] is implemented in a functional language, Erlang [23] and used in Exp. 1. The second version is implemented on the top of the SAPPORO BDD package [30] in C/C++ languages and used in Exp. 2, Exp. 3, and Exp. 4. The environment was Cygwin and gcc/g++ on a PC with CPU of Intel Core i7 2.67 GHz and 3.25 GB of RAM running on Windows XP.

[1] `http://corpus.canterbury.ac.nz/resources/`
[2] `https://github.com/shu-den/seqbdd`

9.2. Results

Exp. 1. In Fig. 12, we show the ratios between the size of reduced sequence BDDs and the size of ADFAs for the same subset in Bible, BibleBi, BibleFac, BibleBiFac, and EcoliFac. For all data sets except Bible, whose ratio is almost 1 at all points, we see that a minimal sequence BDD was 10 to 22 % more succinct than the equivalent minimal ADFA. The extent of savings for data sets consisting of shorter strings, namely, BibleBi, and BibleBiFac, was smaller than for other data sets with longer strings.

Exp. 2. In Fig. 13, we show the computation time of Reduce for a non-reduced sequence BDD on Bible, Ecoli150, Random4, Random128, and FibSuf , where the non-reduced sequence BDD is built as a tree-shaped sequence BDD, i.e., a *trie* [1], by Construct without the subgraph-sharing rule. We observed that the running time was linear in the input size showing its scalability for large data. The running time for FibSuf is smallest because of the small size of the reduced sequence BDD for FibSuf.

Exp. 3. We measured the running time of Meld$_\diamond$ for different operations or data sets, where a pair of reduced sequence BDDs are constructed as inputs from two halves of a data set. Fig. 14 shows the running time on Bible for union, intersection, and difference, while Fig. 15 shows the running time for the union on Bible, Ecoli150, Random4, and Random128. In both cases, we observed that Meld$_\diamond$ ran in almost input linear time, although its theoretical time complexity is quadratic. As the output size was smaller, Meld$_\diamond$ ran faster. For other settings, we obtained similar results.

Exp. 4. In Figs. 16 and 17, we show the time of a construction algorithm using AddString for the reduced sequence BDD for data sets Bible, Ecoli150, Random4, Random128, and FibSuf, where Fig. 17 is the enlarged view of Fig. 16. We observed that the running time was almost linear for each data set, while the time for Random128 on an alphabet of 128 letters took 5 to 20 times longer than that for the other data sets on smaller alphabets. This implies that alphabet size is an important factor. A comparison of the algorithm to Construct for sorted sets is a future problem.

From the above results, we conclude that reduced sequence BDDs are mostly more compact than minimal ADFAs, and the manipulation operations for sequence BDDs are sufficiently efficient and scalable to handle large string sets in practice.

10. Conclusion

In this paper, we considered the class of sequence binary decision diagrams (sequence BDDs) proposed by Loekito *et al.* [24], and studied fundamental properties and computational problems on sequence BDDs: minimization, relationship to acyclic automata, and the complexities of Boolean set operations and construction in the shared sequence BDD environment. We also presented experimental results, which show the efficiency of the sequence BDDs and proposed algorithms for a wide range of data.

On Boolean set operations, we showed where the Meld$_\diamond$ has quadratic time complexity in general, while it runs in input linear time if one of its arguments is a chain-like sequence BDD. Therefore, it is interesting to study special cases that Meld$_\diamond$ has input linear time complexity. In particular, it is an interesting future problem to study the complexities of direct construction of a *factor sequence BDD* from an input sequence BDD as in factor automata of an automaton [18, 32]. Since our shared sequence BDD environment provides dynamic manipulation of string sets in compressed form, it will be interesting to study the dynamic versions of sequence analysis problems, such as the maximal repeat problem and the consistent string problem [15], on sequence BDDs.

Acknowledgments

The authors are grateful to anonymous reviewers of this paper and anonymous referees of PSC'11 of the earlier version of this paper for many useful comments and suggestions, which have improved the quality of this paper. 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, MEXT/ JSPS Global COE Program, FY2007–2011, and ERATO MINATO Discrete Structure Manipulation System Project, JST.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[2] P. Beame, F.E. Fich, Optimal bounds for the predecessor problem and related problems, Journal of Computer and System Sciences 65 (2002) 38 – 72.

[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J.I. Seiferas, The smallest automaton recognizing the subwords of a text, Theoretical Computer Science 40 (1985) 31–55.

[4] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers 35 (1986) 677–691.

[5] J. Bubenzer, Minimization of acyclic DFAs, in: Proceedings of the Prague Stringology Conference 2011 (PSC'11), Czech Technical University in Prague, 2011, pp. 132–146.

[6] M. Crochemore, Transducers and repetitions, Theoretical Computer Science 45 (1986) 63–86.

[7] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on strings, Cambridge University Press, 2007.

[8] M. Crochemore, W. Rytter, Jewels of stringology, World Scientific, 2002.

[9] J. Daciuk, S. Mihov, B.W. Watson, R. Watson, Incremental construction of minimal acyclic finite state automata, Computational Linguistics 26 (2000) 3–16.

[10] S. Denzumi, H. Arimura, S. Minato, Substring Indices Based on Sequence BDDs, Technical Report, TCS Technical Report Series A, TCS-TR-A-10-40, Division of Computer Science, Hokkaido University, 2010.

[11] S. Denzumi, H. Arimura, S.i. Minato, Implementation of sequence bdds in erlang, in: Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Erlang '11, ACM, New York, NY, USA, 2011, pp. 90–91.

[12] S. Denzumi, R. Yoshinaka, H. Arimura, S. Minato, Efficient algorithms on sequence binary decision diagrams for manipulating sets of strings, Technical Report, TCS Technical Report Series A, TCS-TR-A-11-53, Division of Computer Science, Hokkaido University, 2011.

[13] S. Denzumi, R. Yoshinaka, H. Arimura, S. ichi Minato, Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations, in: J. Holub, J. Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2011 (PSC'11), Czech Technical University in Prague, pp. 147–161.

[14] R. Giegerich, S. Kurtz, J. Stoye, Efficient implementation of lazy suffix trees, Software: Practice and Experience 33 (2003) 1035–1049.

[15] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[16] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, The Theory of Machines and computation (1971) 189–196.

[17] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 3rd. edition, 2006.

[18] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, Construction of the CDAWG for a trie, in: Proceedings of the Prague Stringology Conference 2001 (PSC'01), Czech Technical University, 2001, pp. 37–48.

[19] J. R. Driscoll, N. Sarnak, D. Sleator, R. Tarjan, Making data structures persistent, J. of Computer and System Science 38 (1989) 38–86.

[20] H. Kaplan, Persistent data structures, in: D. Mehta, S. Sahni (Eds.), Handbook on Data Structures and Applications, CRC Press, 2005.

[21] H. Kaplan, C. Okasaki, R.E. Tarjan, Simple confluently persistent catenable lists, SIAM J. on Computing 30 (2000) 965–977.

[22] D.E. Knuth, The Art of Computer Programming, volume 4, fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams, Addison-Wesley, 2009.

[23] J. Larson, Erlang for concurrent programming, Commun. ACM 52 (2009) 48–56.

[24] E. Loekito, J. Bailey, J. Pei, A binary decision diagram based approach for mining frequent subsequences, Knowledge and Information Systems 24 (2010) 235–268.

[25] C.L. Lucchesi, T. Kowaltowski, Applications of finite automata representing large vocabularies, Software: Practice and Experience 23 (1993) 15–30.

[26] C.L. Lucchesi, T. Kowaltowski, J. Stolfi, Minimization of binary finite automata, Journal of the Brazilian Computer Society 1 (1995) 5–11.

[27] U. Manber, E.W. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Computing 22 (1993) 935–948.

[28] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.

[29] S. Minato, Zero-suppressed BDDs and their applications, Software Tools for Technology Transfer 3 (2001) 156–170.

[30] S. Minato, SAPPORO BDD package, Division of Computer Science, Hokkaido University, 2011. Unreleased.

[31] S. Minato, H. Arimura, Frequent pattern mining and knowledge indexing based on zero-suppressed BDDs, in: Proceedings of the 5th International Workshop on Knowledge Discovery in Inductive Databases (KDID2006), LNCS 4747, Springer, 2006, pp. 152–169.

[32] M. Mohri, P. Moreno, E. Weinstein, Factor automata of automata and applications, in: Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA'07), LNCS 4783, Springer, 2007, pp. 168–179.

[33] M. Mohri, M. Riley, R. Sproat, Algorithms for speech recognition and language processing, in: Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics, Morgan Kaufmann/ACL, 1996, pp. 231–238.

[34] D. Perrin, Finite automata, in: J. van Leuwen (Ed.), Handbook of Theoretical Computer Science,, volume B. Formal Models and Semantics, Elsevier, 1990, pp. 1–57.

[35] C.E. Shannon, A symbolic analysis of relay and switching circuits, Electrical Engineering 57 (1938) 713–723.

[36] Y. Tian, S. Tata, R.A. Hankins, J.M. Patel, Practical methods for constructing suffix trees, VLDB J. 14 (2005) 281–299.

[37] B.W. Watson, A taxonomy of finite automata construction algorithms, Technical Report, Computing Science Report 93/43, Department of Computing Science, Eindhoven University of Technology, 1993.

[38] I. Wegener, Branching Programs and Binary Decision Diagrams – Theory and Applications, SIAM Monographs on Discrete Mathematics and Applications, SIAM, 2000.

[39] R. Yoshinaka, J. Kawahara, S. Denzumi, H. Arimura, S. Minato, Counter examples to the conjecture on the complexity of BDD binary operations, Technical Report, TCS Technical Report Series A, TCS-TR-A-11-52, Division of Computer Science, Hokkaido University, 2011.

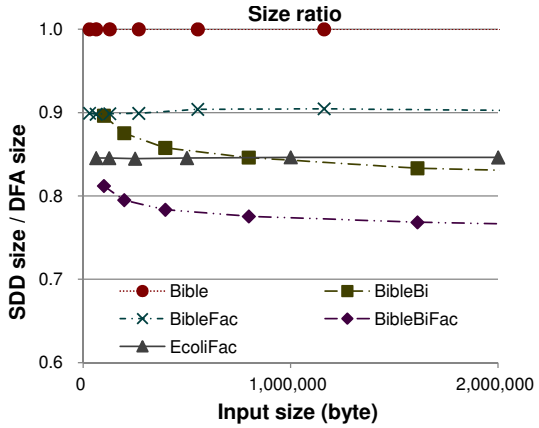

Figure 12: Exp 1. Ratio between the sizes of minimum sequence BDDs and ADFAs.

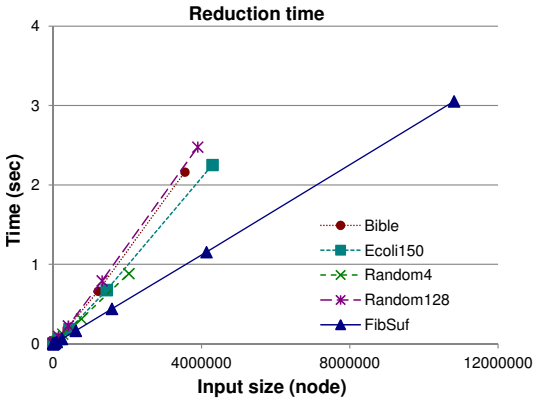

Figure 13: Exp 2. Computation time of Reduce with increasing sequence BDD size.

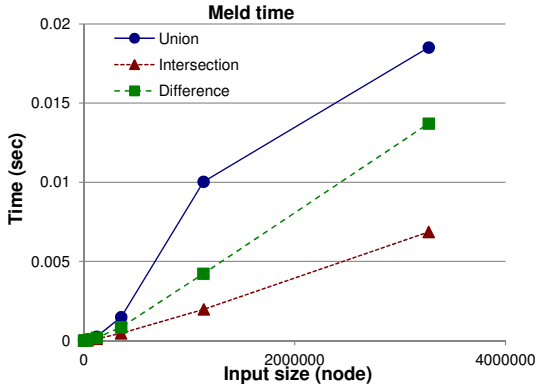

Figure 14: Exp 3. Computation time of $\mathsf{Meld}_\diamond$ for $\diamond \in \{\cup, \cap, \setminus\}$ with increasing sum of input sequence BDD sizes.

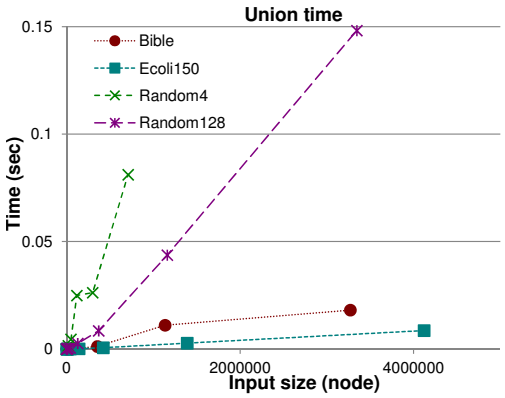

Figure 15: Exp 3. Computation time of $\mathsf{Meld}_\cup$ with increasing sum of input sequence BDD sizes.

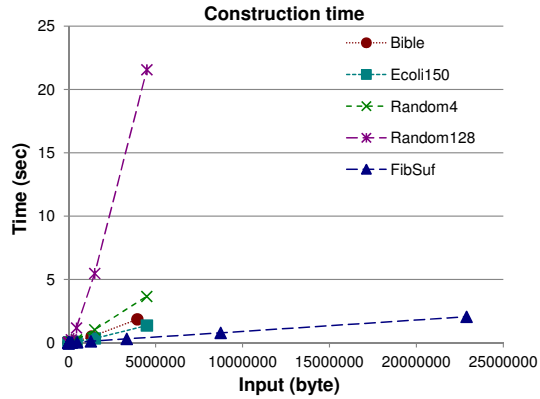

Figure 16: Exp 4. Computation time of Construct with increasing size of a data set.

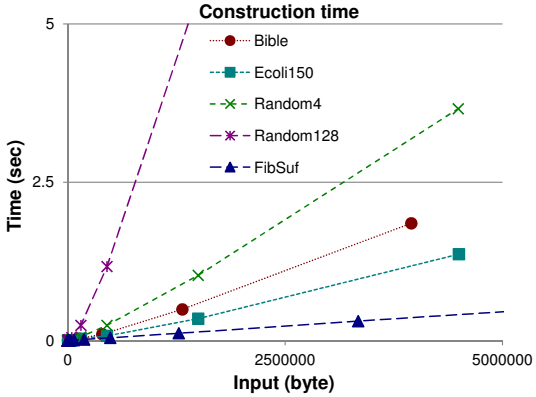

Figure 17: Exp 4. Computation time of Construct with increasing size of a data set (enlarged view).