Studies on Decision Diagrams for Efficient Manipulation of Sets and Strings

Shuhei Denzumi

February, 2015

Division of Computer Science

Graduate School of Information Science and Technology
Hokkaido University

Abstract

In many real-life problems, we are often faced with manipulating discrete structures. Manipulation of large discrete structures is one of the most important problems in computer science. For this purpose, a family of data structures called *decision diagrams* is used. The origin of the decision diagrams is *binary decision diagram (BDD)* proposed by Bryant in 1980s. BDD is a data structure to represent and manipulate Boolean functions efficiently. As a variant of BDD, Minato proposed *zero-suppressed binary decision diagram (ZDD)*. ZDD is a data structure for manipulating families of sets. In 2010, another descendant of BDD, *sequence binary decision diagram (sequence BDD)*, is proposed by Loekito et al. This decision diagram represents sets of strings and allow computing string set operations, too.

In this thesis, we study sequence BDD and ZDD. First, we show 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 Acyclic Deterministic Finite Automata, the complexities of minimization, Boolean set operations, and sequence BDD construction. Secondly, we also define complete inverted files based on sequence BDD for directed acyclic graphs (DAGs). A complete inverted file is an abstract data type that provides various functions for text retrieval. We propose new complete inverted files called SeqBDD-FPs for both texts and DAGs. We also present algorithms to construct them and to retrieve occurrence information from them. Thirdly, we pointed out the problem that is to build index for families of sets in order to store them compactly and allow fast searching. Then, we introduce DenseZDD, a compressed index for static ZDDs to solve a problem that current techniques for storing ZDDs require a huge amount of memory and membership operations are slow. Our technique not only indexes set families compactly but also executes fast membership operations. We also propose a hybrid method of DenseZDD and ordinary ZDDs to allow for dynamic indices.

Content

Chapter 1

Introduction

Discrete structures are fundamental concepts in the field of computer science and discrete mathematics. They include graph theory, combinatorial theory, probablistic theory, stringology, logic, and so on. It is an important technique to represent various types of discrete structures compactly on computers, and processing them efficiently. Such techniques do not only have the theoretical meaning but also have the practical meaning. There are demands to solve a wide range of problems such as equality checking, model analysis, optimization. Therefore, improvement of compact representation of discrete structures and efficient manipulation them will give huge impact on modern society.

One of the most fundamental models among discrete structures is Boolean functions. Binary decision diagram (BDD) is a graph representation of Boolean functions invented by Bryant [6]. BDD has been widely studied especially in VLSI logic design and verification. In recent years, BDD is commonly used in various fields such as constraint sutisfaction problem, data mining in large-scale bussiness data, and genomic analysis. Nowadays, BDDs and its family have been recognized as an important data structure to manipulate discrete structures. Using BDD, we can represent Boolean functions in compact and canonical form. We can obtain a BDD by reducing a binary tree for a given Boolean function. In ordinary BDDs, equivalent subgraphs in multiple BDDs can be shared automatically. Hence, we can store multiple Boolean functions in compressed form. In addition, we can compute the BDD for the result of binary Boolean operations of two BDDs directly by using **Apply** algorithm, proposed by Bryant. This algorithm can compute without expanding the given BDDs. For many practical data, the **Apply** algorithm runs in time almost linear to the input and output BDD size. The most important and superior characteristic of BDDs is the ability to efficiently process compressed

data without expanding them. We can construct desired BDD by using this **Apply** operation repeatedly.

Originally, BDD is a data structure to represent Boolean functions but BDD also can represent families of sets. However, BDD is not the best way to represent familes of sets especially for sparse data sets that frequently occur in real application. Since, linked-list representation of family of sets can be smaller than BDD. BDDs and its families are recognized as an important data structure to manipulate discrete structures. There is a variant of a BDD, Zero-suppressed BDD (ZDD) that is specialized for manipulating families of sets. ZDD is proposed by Minato in twenty years ago [38], and recognized as a most imoportant variation of BDD nowadays. ZDD is used to many real-life problems. ZDD has a different reduction rule from BDD's to represent sparse families of sets. For example, ZDD may be 100 times smaller than BDD for the same family of sets when the number of items is huge and probability of occurence of each item is less than 1 %. Such cases often arise in real-life applications. We can also use apply algorithms to compute the resultant ZDD from two given ZDDs. However, the current ZDD technique has some problem. ZDDs requires large amount of memory in comparison to more compact data structures called succinct data structures. In addition, set searching on ZDD is slow. Its time complexity is not linear to the size of query. It depends on the total number of variables.

Sets of strings is one of fundamental discrete structures to represent many types of data such as text documents, genome information, event sequences. Ordinary ZDD cannot represent sets of strings because ordering of symbols or multiple occurences of symbols are not distinguished in the viewpoint of sets. Loekito et al. proposed sequence BDD (SeqBDD) in 2010 [31]. sequence BDD is almost the same data structure as ZDD, but its variable ordering rule is modified to handle strings [14]. The basic operations of sequence BDD are very similar to those of ZDD. Basic set operations such as union, intersection, difference, and symmetric difference are implemented by almost the same algorithms of ZDD's. Sequence BDD is efficient representation especially for sets of strings having varuous length of strings. Since sequence BDD is a new data structure, its basic properties have not been studied. It is important to understand the relationship between sequence BDD and existing string indices such as acyclic deterministic finite automata. The time and space complexity of sequence BDD operations is required to

improve the research on sequence BDD.

In this paper, we discuss data structures for storing and manipulating sets of strings, sequence BDD and ZDD.

In Chapter 3, 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.

In Chapter 4, we study fundamental properties of sequence BDDs. In particular, we first present non-trivial relationships between sizes of minimum sequence BDDs and minimal ADFAs. We then analyze the complexities of algorithms for Boolean set operations, called the binary synthesis. Finally, we show experimental results to confirm the results of the theoretical analysis on real data sets.

In Chapter 5, we introduces DenseZDD, a compressed index for static ZDDs. Our technique not only indexes set families compactly but also executes fast member membership operations. We also propose a hybrid method of DenseZDD and ordinary ZDDs to allow for dynamic indices.

Chapter 2

Preliminary - Decision Diagrams

In this chapter, we introduce some basic notations, mathematical concepts and definitions used throughout this thesis. We separate these preliminaries into sections related to sets and strings. We also give the definitions of sequence binary decision diagrams and zero-suppressed binary decision diagrams in the style of Bryant [6], which were originally introduced by Loekito *et al.* [31], and Minato [38], respectively.

2.1 Sets and Families

Let $e_1, \ldots, e_n$ be items such that $e_1 < e_2 < \cdots < e_n$. Let $S = \{a_1, \ldots, a_c\}$, $c \geq 1$, be a set of items. We denote the *size* of S by $|S| = c$. When there is no item in S, S is the empty set and denoted by $\emptyset$. A *family* is a subset of the power set of all items. A finite family F of sets is referred to as a *set family*.[1] The *join* of families F_1 and F_2 is defined as $F_1 \sqcup F_2 = \{\, S_1 \cup S_2 \mid S_1 \in F_1, S_2 \in F_2 \,\}$.

2.1.1 Zero-suppressed binary decision diagrams

A *zero-suppressed binary decision diagram* (a ZDD) [38] is a variant of a binary decision diagram [6], customized to manipulate finite families of set. A ZDD is a directed acyclic graph satisfying the following conditions. A ZDD has two types of nodes, terminal and nonterminal nodes. A *terminal node* v has as attribute a value $\mathit{value}(v) \in \{0, 1\}$, indicating whether it is a *0-terminal node* or a *1-terminal node*, denoted by $\mathbf{0}$ and $\mathbf{1}$, respectively. A *nonterminal node* v has as attributes an integer $\mathit{index}(v) \in \{1, \ldots, n\}$ called the *index*, and two children $\mathit{zero}(v)$ and $\mathit{one}(v)$, called the *0-child* and *1-child*.

[1] In the original ZDD paper by Minato, a set is called a combination, and a set family is called a combinatorial set.

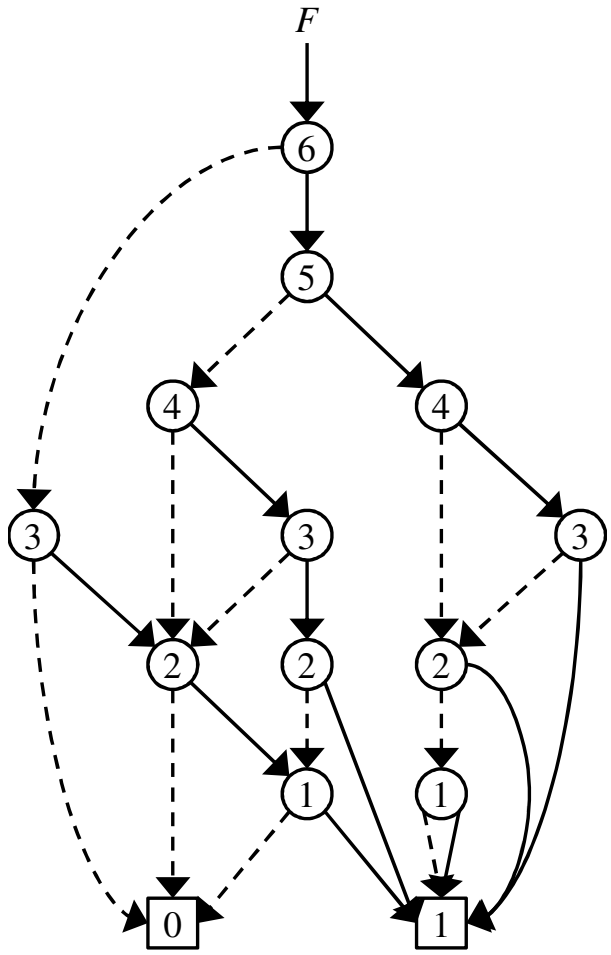


Figure 2.1: An example of ZDD

The edges from nonterminals to their 0-child (1-child resp.) are called *0-edges* (*1-edges* resp.). In the figures, terminal nodes are denoted by squares, and nonterminal nodes are denoted by circles. 0-edges are denoted by dotted arrows, and 1-edges are denoted by solid arrows. We define $triple(v) = \langle index(v), zero(v), one(v)\rangle$, called the *attribute triple* of v. For any nonterminal node v, $index(v)$ is larger than the indices of its children.[2] We define the *size* of the graph, denoted by $|G|$, as the number of its nonterminals.

Definition 1 (set family represented by ZDD) *A ZDD G rooted at a node $v \in V$ represents a finite family of sets $F(v)$ on U_n defined recursively as follows: (1) If v is a terminal node: $F(v) = \{\emptyset\}$ if $value(v) = 1$, and $F(v) = \emptyset$ if $value(v) = 0$. (2) If v is a nonterminal node, then $F(v)$ is the finite family of sets $F(v) = (\{e_{index(v)}\} \sqcup F(one(v))) \cup F(zero(v))$.*

[2]In ordinary BDD or ZDD papers, the indices are in ascending order from roots to terminals. For convenience, we employ the opposite ordering in this paper when we discuss ZDD.

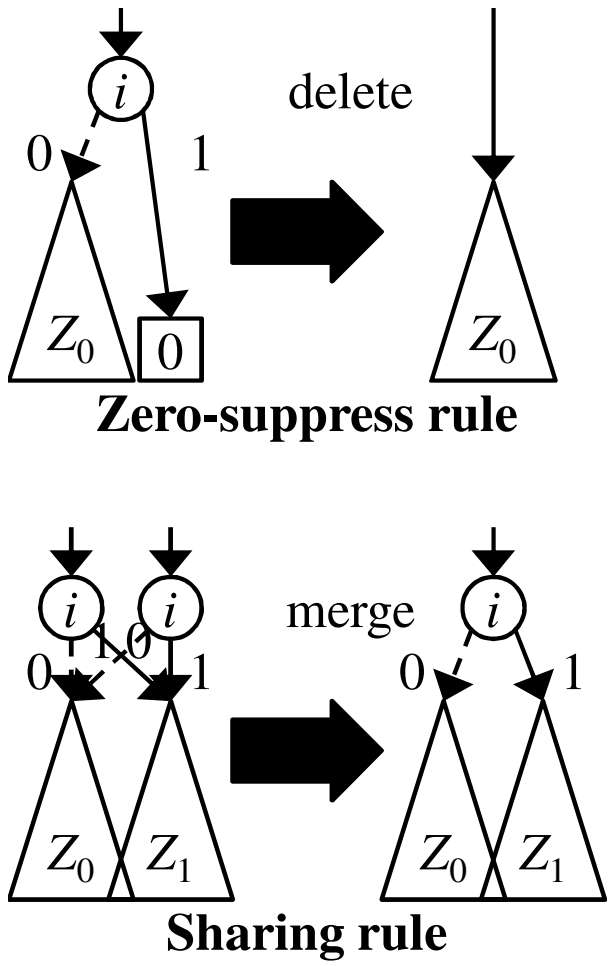


Figure 2.2: Reduction rules of ZDDs

The example in Fig. 2.1 represents a sets family $F = \{ \{6,5,4,3\}, \{6,5,4,2\}, \{6,5,4,1\}, \{6,5,4\}, \{6,5,2\}, \{6,5,1\}, \{6,5\}, \{6,4,3,2\}, \{6,4,3,1\}, \{6,4,2,1\}, \{6,2,1\}, \{3,2,1\} \}$.

A set $S = \{a_1, \ldots, a_c\}$ describes a path in the graph G starting from the root. At each nonterminal node, the path continues to the 0-child if $e_i \notin S$ and to the 1-child if $e_i \in S$. The path eventually reaches the 1-terminal (or 0-terminal resp.), indicating that S is accepted (or rejected resp.).

In ZDD, we employ the following two reduction rules to compress the graph: (a) Zero-suppress rule: A nonterminal node whose 1-child is the 0-terminal node. (b) Sharing rule: Two or more nonterminal nodes having the same attribute triple. By applying above rules, we can reduce the graph without changing its semantics. If we apply the two reduction rules as much as possible, then we obtain a canonical form for a given family of sets.

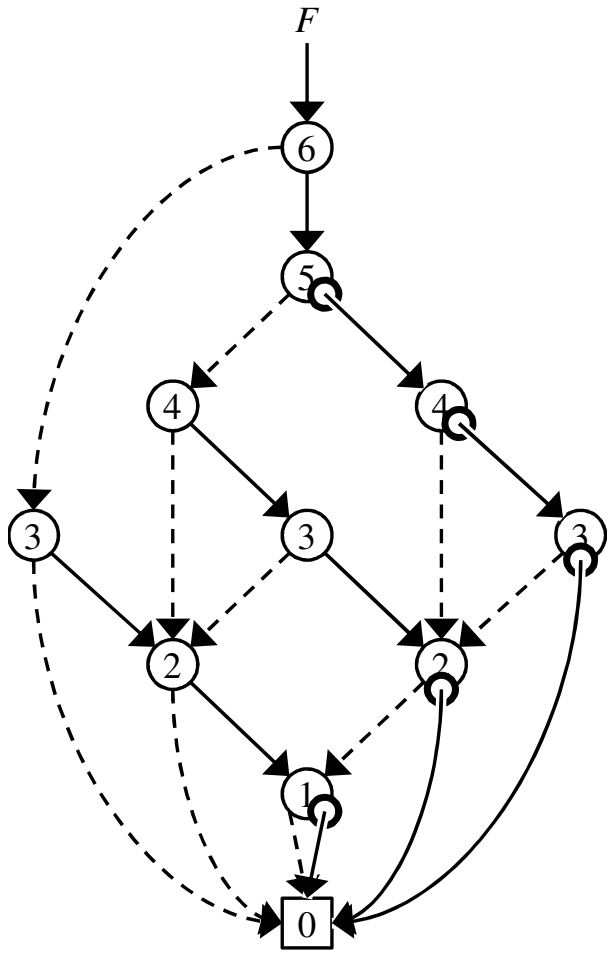


Figure 2.3: The ZDD using 0-element edges that is equivalent to the ZDD in Fig. 2.1

We can reduce the size of ZDDs by using a type of *attributed edges* [43] named *0-element edges*. By using the 0-element edge, which simplifies construction of DenseZDD. We can implement 0-element edges by 1-edges with *empty set flags*. This attribute indicates that the pointing subgraph includes the empty set $\emptyset$ in the family represented by the subgraph. If it is set, it means that the node pointed to by the 1-edge represents a set family including the empty set $\emptyset$. includes the empty set $\emptyset$. We have to place a couple of constraints on using 0-element edges to keep the uniqueness of the graphs: (1) Use the 0-terminal node only. (2) Do not use 0-element edges at the 0-edge on each node. Each nonterminal node v has an $\emptyset$-*flag* $empflag(v)$ on its 1-edge to implement 0-element edges. If $empflag(v) = 1$, the subgraph pointed by the v's 1-edge includes the empty set $\emptyset$ in the family represented by the subgraph. In this paper, effective $\emptyset$-flags are denoted as circles at starting points of 1-edges.

Table 2.1: Main operations supported by ZDD. The first group is the primitive ZDD operations used to implement the others, yet they could have other uses

$index(v)$	Returns the index of node v.
$zero(v)$	Returns the 0-child of node v.
$one(v)$	Returns the 1-child of node v.
$getnode(i, v_0, v_1)$	Generates (or makes a reference to) a node v with index i and two child nodes $v_0 = zero(v)$ and $v_1 = one(v)$.
$topset(v, i)$	Returns a node with the index i reached by traversing only 0-edges. If such a node does not exist, return the 0-terminal node.
$member(v, S)$	Returns *true* if the set $S \in F(v)$, and returns *false* otherwise.
$listup(v)$	Outputs every set $S \in F(v)$.
$count(v)$	Returns $\lvert F(v)\rvert$.
$offset(v, i)$	Returns a node u such that $F(u) = \{ S \subseteq U_n \mid S \in F(v), e_i \notin S \}$.
$onset(v, i)$	Returns a node u such that $F(u) = \{ S \backslash \{e_i\} \subseteq U_n \mid S \in F(v), e_i \in S \}$.
$apply_\diamond(v_1, v_2)$	Returns v such that $F(v) = F(v_1) \diamond F(v_2)$, for $\diamond \in \{\cup, \cap, \backslash, \oplus\}$.

Table 2.1 summarizes operations of ZDDs. The upper half shows the primitive operations, while the lower half shows other operations which can be implemented by using the primitive operations. The operations $index(v)$, $zero(v)$, $one(v)$, $topset(v, i)$ and $member(v, S)$ do not create new nodes. Therefore they can be done on a static ZDD. The operation $count(v)$ does not create any node; however we need an auxiliary array to memorize which nodes are already visited.

2.2 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

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

Figure 2.4: The attribute values for a vertex v.

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.

2.2.1 Sequence binary decision diagrams

In this subsection, we give the sequence BDD, introduced by Loekito *et al.* [31], 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 structure 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. 2.4. A graphical explanation of the correspondence between the attribute values and the vertex type is given in Fig. 2.5.

Definition 2 (Sequence BDD) *[31] 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*

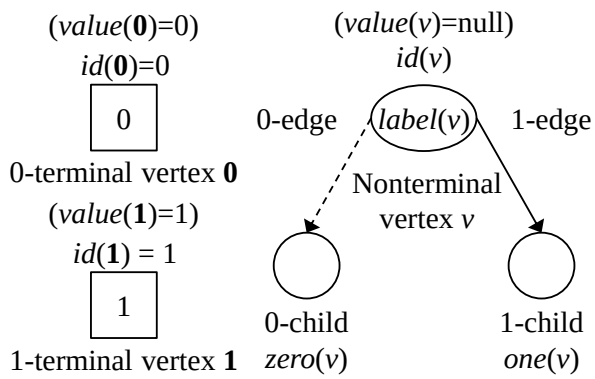


Figure 2.5: The 0-terminal, 1-terminal and nonterminal vertices.

attributes are shown in Table 2.4.

- *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.*

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, $\mathbf{0}$ or $\mathbf{1}$, 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. 3.3.

2.2.2 The first semantics

For representing a Boolean function, Bryant presented BDD [6] based on the Shannon expansion [51] 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 nontrivial 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

$$head(L) = \min\{\, c \in \Sigma \mid cs \in L, s \in \Sigma^* \,\}, \tag{2.1}$$

$$onset(L) = \{\, s \mid cs \in L, c = head(L) \,\}, \tag{2.2}$$

$$offset(L) = \{\, cs \mid cs \in L, c \neq head(L) \,\} \cup (\{\epsilon\} \cap L). \tag{2.3}$$

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

$$L = (head(L) \cdot onset(L)) \ \cup \ offset(L). \tag{2.4}$$

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. 2.6 illustrates the binary decomposition of finite language

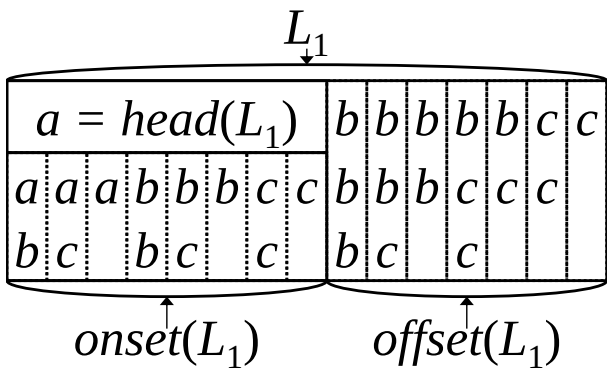


Figure 2.6: The binary decomposition of a finite language $L_1 = \{$ *aab*, *aac*, *aa*, *abb*, *abc*, *ab*, *acc*, *ac*, *bbb*, *bbc*, *bb*, *bcc*, *bc*, *cc*, *c* $\}$ in Example 1.

$L = L_1$, 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. (2.4), 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. □

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 3 (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 $value(v) = 0$, $L_G(v) = \emptyset$.*

Table 2.2: Main operations supported by SeqBDD. The first group is the primitive SeqBDD operations used to implement the others, yet they could have other uses

$label(v)$	Returns the label of node v.
$zero(v)$	Returns the 0-child of node v.
$one(v)$	Returns the 1-child of node v.
$getnode(c, v_0, v_1)$	Generates (or makes a reference to) a node v with label $c \in \Sigma$ and two child nodes $v_0 = zero(v)$ and $v_1 = one(v)$.
$member(v, s)$	Returns *true* if the string $s \in L(v)$, and returns *false* otherwise.
$listup(v)$	Outputs every string $s \in L(v)$.
$count(v)$	Returns $\|L(v)\|$.
$offset(v, c)$	Returns a node u such that $L(u) = \{\, as \mid a, c \in \Sigma, a \neq c, as \in L(v) \,\}$.
$onset(v, c)$	Returns a node u such that $L(u) = \{\, s \mid c \in \Sigma, cs \in L(v) \,\}$.
$apply_\diamond(v_1, v_2)$	Returns v such that $L(v) = L(v_1) \diamond L(v_2)$, for $\diamond \in \{\cup, \cap, \setminus, \oplus\}$.

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 $L = L(G)$.

Table 2.2 summarizes operations of SeqBDDs. The upper half shows the primitive operations, while the lower half shows other operations which can be implemented by using the primitive operations.

2.2.3 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.

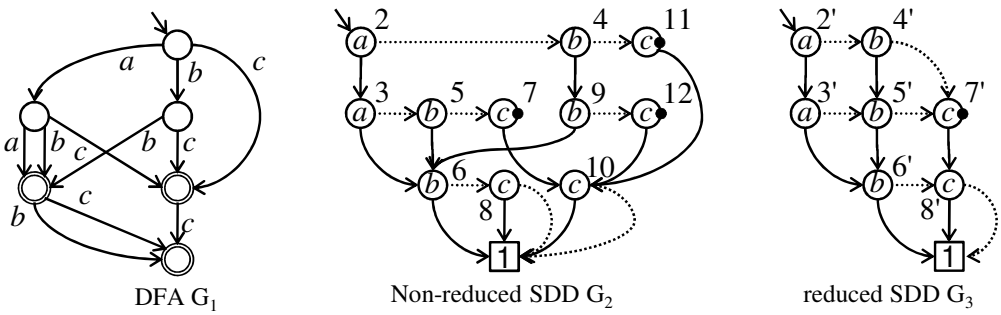


Figure 2.7: A minimal acyclic DFA G_1 (left), a non-reduced sequence BDD G_2 (middle), and a reduced sequence BDD G_3 (right) on $\Sigma_1 = \{a, b, c\}$ for the same finite language L_1 in Example 1. The numbers at each node indicate vertex id. The 0-terminal $\mathbf{0}$ is omitted, and all edges incoming to $\mathbf{0}$ are indicated by a small black dot.

Definition 4 (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 $\mathbf{1}$ 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 [23]). The second semantics of a sequence BDD is given by its NFA.

Definition 5 (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

```
procedure Decide(v: a vertex, s = a_1 ··· a_n: a string of length n ≥ 0 on Σ):
for (i ← 1, ..., n)
    while (label(v) ≺_Σ a_i) v ← zero(v);
    if (a_i ≺_Σ label(v)) return "Reject";
    else v ← one(v);
while (value(v) = null) v ← zero(v);
if (value(v) = 0) return "Reject";
else return "Accept";
```

Figure 2.8: The procedure for deciding the acceptance of a string s by a sequence BDD vertex v.

$A(G)$. Fortunately, however, the next lemma says that the decision can be done by the deterministic procedure Decide as shown in Fig. 2.8.

Lemma 2 *The procedure* Decide *in Fig. 2.8 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. 2.7, 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.*

2.2.4 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 6 *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 7 *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 diagrams of Bryant [6]. 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 [39]. 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 $\mathit{value}(v) = \mathit{value}(v')$. 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 $\mathit{triple}(v) = \mathit{triple}(v')$, the claim obviously holds. (II) Otherwise, $\mathit{triple}(v) \neq \mathit{triple}(v')$. Since $\mathit{label}(v) = \mathit{label}(v')$, there are two cases below: In the case (II.i) that $\mathit{zero}(v) \neq \mathit{zero}(v')$. Generally, we observe that for any vertices v and v' if $G(v) \approx G(v')$, then $G(\mathit{zero}(v)) \approx G(\mathit{zero}(v'))$ and $G(\mathit{one}(v)) \approx G(\mathit{one}(v'))$. From this observation, it follows from the assumption that $G(\mathit{zero}(v)) \approx G(\mathit{zero}(v'))$. Since, $|G(\mathit{zero}(v))|$ 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 $\mathit{one}(v) \neq \mathit{one}(v')$, 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 $\mathit{head}(L(v)) = \mathit{label}(v)$, $\mathit{onset}(L(v)) = L(\mathit{one}(v))$, and $\mathit{offset}(L(v)) = L(\mathit{zero}(v))$ 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, $L(\mathit{one}(v)) \neq \emptyset$. Thus, Claim (2) immediately follows from (2) of Lemma 1. □

Chapter 3

Fundamental Properties of Sequence Binary Decision Diagrams

This chapter studies sequence binary decision diagrams as a data structure for finite sets of strings. We prove various basic results that we will need for further understanding this novel data structure. We include the proofs here for completeness. First, We show the minimality of sequence BDDs in the same manner as the famous Myhill-Nerodes theorem. Secondly, we define operations that sequence BDDs support. Then, we present algorithms for minimization and basic binary set operations, union, intersection, difference, and symmetric difference, and analyze the time and space complexity of them. Thirdly, we show how to construct sequence BDDs from given finite strings. Fourthly, we reveal the relationship between sequence BDDs and acyclic directed finite automata that also represent finite sets of strings. Finally, we show some experimental results.

3.1 Introduction

3.1.1 Background

Compact string indexes for storing sets of strings are fundamental data structures in computer science, and have been extensively studied for decades [4, 8, 9, 11, 20, 45]. Examples of compact string indexes include tries [1, 9], finite automata and transducers [11, 23], suffix trees [37], suffix arrays [34], DAWGs [4], and factor automata (FAs) [45]. 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 [9, 20]. 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*) [9, 11, 23] 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 [32, 46]. 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.* [31] proposed the class of *sequence binary decision diagrams* which is a compact representation of finite sets of strings that allows a variety of operations to be performed for sets of strings.

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.

3.1.2 Main results

In this chapter, 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 chapter. 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 [6, 31] in the style of Knuth's *melding* for BDDs [28]. 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 [31]. 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 prop-

erty, 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.

3.1.3 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 [11, 32, 33, 46, 45]. 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 [22] presented an $O(n \log n)$ state-minimization algorithm for a given DFA. Daciuk *et al.* [11] presented incremental and off-line algorithms for constructing minimal ADFA, to which Watson [55] added a survey of construction algorithms for minimal automata. Blumer *et al.* [4] and Crochemore [8] presented linear-time algorithms for constructing minimal ADFAs, called *DAWGs* and *factor automaton*, for all suffixes and all factors of a string, respectively. Mohri [45] gave an algorithm for constructing a factor automaton from an ADFA. Denzumi *et al.* [12, 16] 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 [23] gave classic examples of a quadratic-time product algorithm for computing the union, intersection, and difference of two DFAs. Loekito *et al.* [31] 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. 3.4 generalizes their algorithm for all Boolean set operations. The work most closely related to that presented in this thesis is that of Lucchesi *et al.* [33] as well as of Bubenzer [7]. Lucchesi *et al.* [33] introduced *binary automata*, which are essentially

the same as sequence BDDs when restricted to acyclic ones, and discuss their minimization procedure. Bubenzer [7] studied a bottom-up minimization of acyclic DFAs using tabulation similar to Reduce described in Sec. 3.3. 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 [6, 28, 39, 56]. 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 [6]. On the other hand, a variant of BDDs with zero-suppress and subgraph-sharing rules, called *zero-suppressed BDDs* (*ZDDs*), was proposed by Minato [39] in the 90s for sparse combinatorial sets, and later applied to databases and data mining [42]. Loekito *et al.* [31] 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 [31].

Sequence BDDs are closely related to the notion of *persistent data structures* [25, 26], 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* [25, 26] if all versions can be both accessed and modified by a unary operation, and is called *confluently persistent* [26, 27] if it is fully persistent and all versions can be combined by a binary operation [26]. It is not easy to make a data structure confluently persistent [26], whereas there are general transformation methods for fully persistent ones [25]. In this context, our shared reduced sequence BDD described in Sec. 3.3 is a confluently persistent data structure for storing sets of strings in compressed form.

3.1.4 Organization of this chapter

In Sec. 4.2, we introduce a formalization of sequence BDDs. In Sec. 3.2, we give a characterization of minimal sequence BDDs, and the shared sequence BDD environment.

We give a linear-time minimization algorithm in Sec. 3.3. In Sec. 3.4, we discuss upper and lower bounds of the time complexities of Boolean set operation. In Sec. 3.5, we show linear-time construction algorithms. In Sec. 3.6, we give the size bounds for sequence BDDs and DFAs. In Sec. 3.7, we show an application of a sequence BDD and give an algorithm that constructs a factor graph from a sequence BDD. In Sec. 4.5, we show experimental results. In Sec. 4.6, we present our conclusion.

3.2 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 8 (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. 2.7, 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 [23, 49] 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 9 (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 $v \in V$, $\phi(v) = (label(v) \cdot \phi(one(v))) \cup \phi(zero(v))$.*

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, 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.

Proof: The proof of (1) ⇒ (2): It is immediate from Lemma 7. The proof of (2) ⇒ (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. $\square$

3.3 Operations

3.3.1 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 $\mathcal{E} = (G, \mathit{uniqtable}, \mathit{cache})$ 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 $v = \mathit{uniqtable}(c, v_0, v_1)$ 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 $\mathit{triple}(v) = \tau$. 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.

Table 3.1: 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**	$O(1)$
$\mathit{Getvertex}(c, v_0, v_1)$	A vertex with a given triple	$O(1)$
$\mathit{Reduce}(G)$	G reduced to canonical form	$O(\|G\|)$
$\mathit{Meld}(G_1, G_2)$	$L_1 \langle op \rangle L_2$ for $\langle op \rangle \in \{\cup, \cap, -, \oplus\}$	$T_{Meld}(G_1, G_2) = O(\|G_1\|\|G_2\|)$
$\mathit{Concat}(G_1, G_2)$	$L_1 \cdot L_2$	$O(\|G_1\| \cdot 2^{\|G_2\|})$
$\mathit{MakeString}(s)$	A sequence BDD for a string s	$O(\|s\|)$
$\mathit{AddString}(G, s)$	$L \cup \{s\}$	$O(\|\Sigma\| \cdot \|s\|)$
$\mathit{DeleteString}(G, s)$	$L - \{s\}$	$O(\|\Sigma\| \cdot \|s\|)$
$\mathit{equals}(G_1, G_2)$	$L(G_1) = L(G_2)$?	$O(1)$
$\mathit{isContained}(G_1, G_2)$	$L(G_1) \subseteq L(G_2)$?	$O(T_{Meld}(G_1, G_2))$
$\mathit{Print\text{-}one}(G)$	some string of $L(G)$	$O(\mathsf{maxlen}(L(G)))$
$\mathit{Print\text{-}all}(G)$	$L(G)$	$O(\mathsf{maxlen}(L(G))\|L(G)\|)$
$\mathit{Count}(G)$	$\|L(G)\|$	$O(\|G\|)$

3.3.2 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 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 3.1. 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.

3.3.3 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. 2.2.2, 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. 3.1, 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*

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 3.1: 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.

of a symbol in Σ *and two existing vertices in* G*, such that* $c \prec_\Sigma label(v_0)$*. 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 [3]), then Getvertex can be implemented in $O((\log\log N)^2)/\log\log\log N)$ worst case time and $O(N)$ words of space.

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 [26] for sets of variable-length strings.

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** (($u \leftarrow cache[\text{“}Reduce(v)\text{”}]$) exists) **return** u;
3: **else** $u \leftarrow$ Getvertex($label(v)$, Reduce($zero(v)$), Reduce($one(v)$));
4: $\quad cache[\text{“}Reduce(v)\text{”}] \leftarrow u$;
5: $\quad$ **return** u;

Figure 3.2: An off-line minimization algorithm for a sequence BDD.

3.3.4 Linear-time minimization algorithm

Using the algorithm Getvertex in the previous subsection, we show in Fig. 3.2 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. 3.2 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 [3]).*

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 $|\Sigma|$.*

3.4 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 [31] based on a recursive synthesis algorithm Apply for Boolean operations by Bryant [6] in the style of Knuth [28].

3.4.1 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 $\mathsf{sign}(P)$ 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.

Definition 10 (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. 3.3, 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 [28] and by induction on top-left decomposition, we can show the correctness of $\mathsf{Meld}_\diamond$.

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** ($u = \mathbf{0}$ **or** $v = \mathbf{0}$ **or** $u = v$)
2: **if** ($F_\diamond[\mathsf{sign}(u), \mathsf{sign}(v)] = 0$) **return** $\mathbf{0}$; {See Sec. 3.4.1 for $F_\diamond$.}
3: **else if** $u \neq \mathbf{0}$ **return** u;
4: **else if** $v \neq \mathbf{0}$ **return** v;
5: **else if** (($w \leftarrow cache[\text{“}Meld_\diamond(u, v)\text{”}]$) exists) **return** w;
6: **else**
7: **if** ($label(u) \prec_\Sigma label(v)$)
$w \leftarrow \mathsf{Getvertex}(label(u), \mathsf{Meld}_\diamond(zero(u), v), \mathsf{Meld}_\diamond(one(u), \mathbf{0}))$;
8: **else if** ($label(v) \prec_\Sigma label(u)$)
$w \leftarrow \mathsf{Getvertex}(label(v), \mathsf{Meld}_\diamond(u, zero(v)), \mathsf{Meld}_\diamond(\mathbf{0}, one(v)))$;
9: **else if** ($label(u) =_\Sigma label(v)$)
$w \leftarrow \mathsf{Getvertex}(label(u), \mathsf{Meld}_\diamond(zero(u), zero(v)), \mathsf{Meld}_\diamond(one(u), one(v)))$;
10: $cache[\text{“}Meld_\diamond(u, v)\text{”}] \leftarrow w$;
11: **return** w;

Figure 3.3: An algorithm $\mathsf{Meld}_\diamond$ for Boolean string set operations, where $\diamond \in \{\cup, \cap, \oplus, \backslash, /, \emptyset, LHS, RHS\}$ and function $F_\diamond$ are defined in Sec. 3.4.1.

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.

3.4.2 Input-sensitive complexity of binary synthesis

First, we start with input-sensitive analysis of the time complexity of $\mathsf{Meld}_\diamond$ of Fig. 3.3. 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. 3.3 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 $\#\mathsf{Meld}^1_\diamond \leq |P| \cdot |Q|$ (*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 $\#\mathsf{Meld}_\diamond \leq 2 \cdot \#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 2 and 3, we have that $\#\mathsf{Meld}_\diamond \leq 2 \cdot |P| \cdot |Q| + 1 = O(|P| \cdot |Q|)$. 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 $\mathsf{Getvertex}$ by adding a new node. Thus, the algorithm adds at most $|R| \leq \#\mathsf{Getvertex} \leq \#\mathsf{Meld}_\diamond = O(|P| \cdot |Q|)$ 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 $\diamond \in \mathcal{O}$, the reduced output size $|R|$ is bounded from above by $O(|P| \cdot |Q|)$.*

3.4.3 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 [56], 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 $\mathsf{Getvertex}$ by removing Line 1 and 2 of Fig. 3.1 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. 3.3, 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. 3.3 equipped with $\mathsf{Getvertex}^*$. Since each call of $\mathsf{Getvertex}^*$ increases the output size by at least one, we have $\#\mathsf{Getvertex}^* \leq |R^*|$ (*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 $\mathsf{Getvertex}$, we have $\#\mathsf{Meld}^1_\diamond \leq \#\mathsf{Getvertex}^*$ (*Claim 5*). From the proof for Theorem 4, we have $\#\mathsf{Meld}_\diamond \leq 2{\cdot}\#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 3, 4, and 5 above, we now have $\#\mathsf{Meld}_\diamond \leq 2{\cdot}\#\mathsf{Meld}^1_\diamond + 1 \leq 2{\cdot}\#\mathsf{Getvertex}^* + 1 \leq 2{\cdot}|R^*| + 1 = O(|R^*|)$. 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. □

3.4.4 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. [57] 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. 3.3 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 [57]. The theorem can be shown in a way similar to that in [57]. Here, we give a rough sketch of the proof. Let $\Sigma = \{0, 1\}$. For a fixed positive integer n, we define

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

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

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

We have

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

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 $a_1 b_1 \ldots a_n b_n c_1 \ldots c_m$ 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 $|P|, |Q| = O(2^n)$.

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

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

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

Procedure $\mathsf{MakeString}(s \in \Sigma^*$: string):

Output: The root of the reduced sequence BDD for a string s in linear form;

if $(s = \varepsilon)$ **return 1**;

else return $\mathsf{Getvertex}(s[1], \mathbf{0}, \mathsf{MakeString}(s[2] \cdots s[|s|]))$;

Figure 3.4: 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.

Second, we show that $\#\mathsf{Meld}_{\diamond} \geq 2^{2n}$. 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, $\mathsf{Meld}_{\diamond}(P_x, Q_x)$ is called for each x. Moreover, $P_{a_1 \ldots b_n} \neq P_{a'_1 \ldots b'_n}$ whenever $a_i \neq a'_i$ for some i and $Q_{a_1 \ldots b_n} \neq Q_{a'_1 \ldots b'_n}$ whenever $b_i \neq b'_i$ 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 $\#\mathsf{Meld}_{\diamond} \geq 2^{2n}$. $\square$

3.5 Linear-time On-the-Fly and Off-line Construction

In Fig. 3.4, using algorithm $\mathsf{Meld}_{\diamond}$, we present efficient on-the-fly construction algorithms, $\mathsf{AddString}(P, s)$ and $\mathsf{DeleteString}(P, s)$, 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 upper bound of the running time of $\mathsf{Meld}_{\diamond}$ 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. 3.4 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 $\mathsf{Meld}_\cup$. 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, $\mathsf{Meld}_\cup$ follows the path in P whose labels spell a prefix of string s by making $O(|\Sigma| \cdot |Q|)$ calls of $\mathsf{Meld}_\cup$ 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.* [11] presented an on-the-fly construction algorithm with an insert for ADFA. Interestingly, we see that our algorithm performs the same computation as their algorithm in [11, 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. 3.5 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

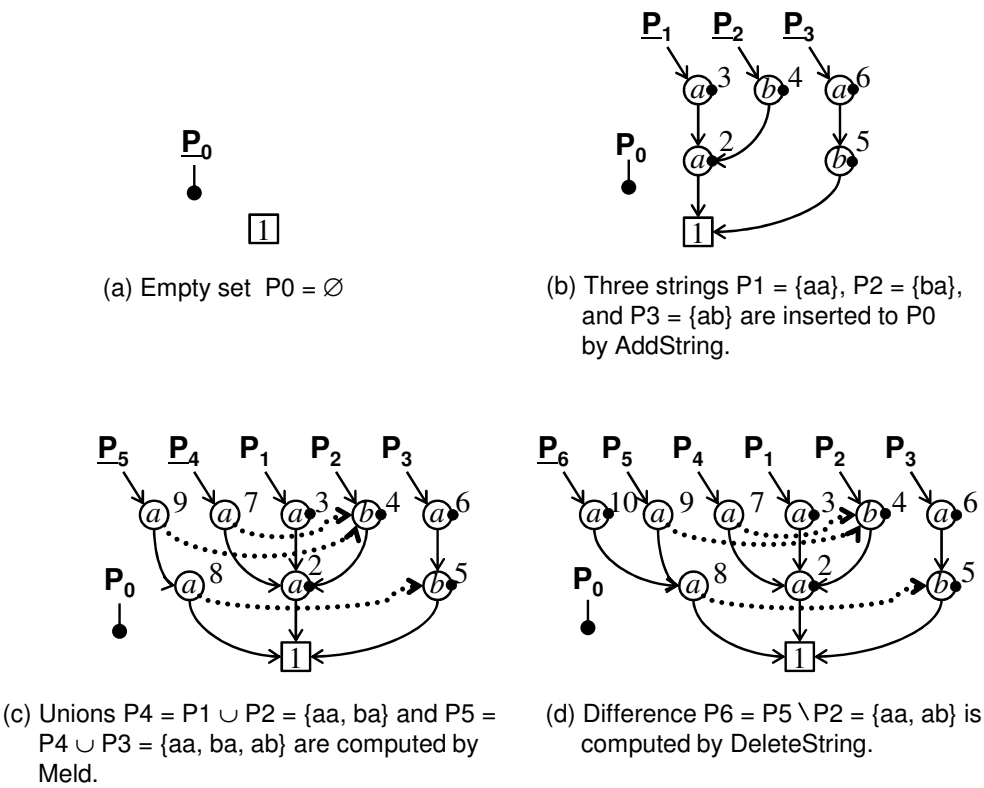


Figure 3.5: An execution example of AddString, DeleteString, and $\mathsf{Meld}_{\cup}$ in a shared sequence BDD environment $\mathcal{E}$, where the computation proceeds in write-only manner. In the figure, each circle stands for a vertex, and an associated symbol and a number indicate its vertex label 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**.

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 $\mathsf{Construct}(L)$ 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* $\mathsf{Construct}(L)$ *computes the reduced sequence BDD for L in $O(n)$ expected time and space, where $n = ||L||$ and the running time is independent of $|\Sigma|$.*

3.6 Space-Bounds for Sequence Binary Decision Diagrams and Acyclic Automata

3.6.1 Finite automata

We presume a basic knowledge of the automata theory. For a comprehensive introduction to it, see [23, 49] 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.

3.6.2 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 $|B| \leq n$. $\square$

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.

3.6.3 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 $F' = \{\, \widetilde{v} \in \Gamma' \mid \widetilde{v} = [v_1, \ldots, v_k] \text{ and } v_k = \mathbf{1} \,\}$, 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 $a_0 \prec a_1 \prec \cdots \prec a_m$. $\square$

For BDDs [6] and ZDDs [39], it is well-known that the choice of ordering $\prec$ on Σ, called *variable ordering*, affects the size of the resulting BDD [6, 28, 39]. 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 [4, 8].

Theorem 12 (Blumer et al. [4], Crochemore [8]) *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 $w = cb^na$ 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 $B_w^\pi = FSDD(w)^\pi$, and A_w be the minimal automaton for $Fact(w)$ with state set Γ_w. We have $|B_w^\pi| \leq |A_w|$ and $|\Gamma_w| \leq |B_w^\rho| + 1$ by Theorems 10 and 11, respectively. Blumer et al. [4, Lemma 1.6] showed that $|A_w| \leq |\Gamma_w| + |w| - 2$. Hence, $|B_w^\pi| \leq |A_w| \leq |\Gamma_w| + |w| - 2 \leq |B_w^\rho| + |w| - 1$. In fact, for $w = a^nb$, $\pi = \langle b \prec a \rangle$, $\rho = \langle a \prec b \rangle$, we have $|B_w^\pi| = 2n - 1$ and $|B_w^\rho| = n - 1$. □

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** ($u \leftarrow cache[\text{“}RecFSDD(v)\text{”}]$ 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: $cache[\text{“}RecFSDD(v)\text{”}] \leftarrow u$;

8: **return** u;

Figure 3.6: A recursive procedure RecFSDD that constructs the factor sequence BDD of an input sequence BDD vertex v.

3.7 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.

3.7.1 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 $||L||$ [4, 10], the size of $FSDD(G)$ is also linear in $||L(G)||$. (See [17] for the tight upper bound 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. 3.6, we present a simple recursive algorithm, RecFSDD, which computes $FSDD(G)$ from an input sequence BDD

Table 3.2: Outline of data sets.

Dataset							SDD				
name	#letter (byte)	#line	#unique line	line length (byte)		$\|\Sigma\|$	#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

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. 3.6 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 [19] and other top-down algorithms [45, 54]. In fact, we observed that applications of Pref and $\mathsf{Meld}_\cup$, resp., correspond to the ε-edge attachment and determinization in [45]. Experiments in Sec. 4.5 showed that RecFSDD ran in almost linear time to the input size $|G|$ on some real data sets.

3.8 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.

3.8.1 Experimental setting

In the experiments, we used the following data sets, whose characteristics are shown in Table 3.2. 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

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

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. 3.3, Meld described in Sec. 3.4, and an off-line construction algorithm using AddString described in Sec. 3.5. The first version[2] [13] is implemented in a functional language, Erlang [30] and used in Exp. 1. The second version is implemented on the top of the SAPPORO BDD package [40] 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.

3.8.2 Results

Exp. 1. In Fig. 3.7, 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. 3.8, 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.

[2] `https://github.com/shu-den/seqbdd`

Exp. 3. We measured the running time of $\mathsf{Meld}_\diamond$ 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. 3.9 shows the running time on Bible for union, intersection, and difference, while Fig. 3.10 shows the running time for the union on Bible, Ecoli150, Random4, and Random128. In both cases, we observed that $\mathsf{Meld}_\diamond$ ran in almost input linear time, although its theoretical time complexity is quadratic. As the output size was smaller, $\mathsf{Meld}_\diamond$ ran faster. For other settings, we obtained similar results.

Exp. 4. In Figs. 3.11 and 3.12, 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. 3.12 is the enlarged view of Fig. 3.11. 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.

3.9 Conclusion

In this chapter, we considered the class of sequence binary decision diagrams (sequence BDDs) proposed by Loekito *et al.* [31], 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 $\mathsf{Meld}_\diamond$ 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 $\mathsf{Meld}_\diamond$ 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 [24, 45]. 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 [20], on sequence BDDs.

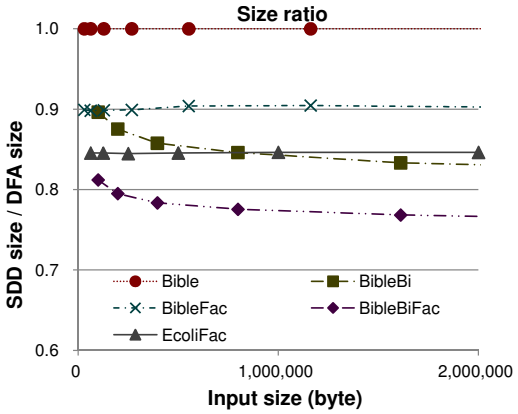


Figure 3.7: Exp 1. Ratio between the sizes of minimum sequence BDDs and ADFAs.

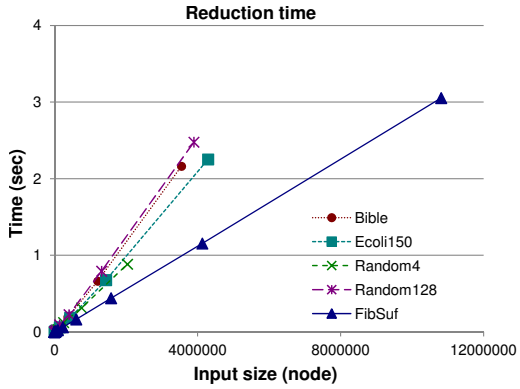


Figure 3.8: Exp 2. Computation time of Reduce with increasing sequence BDD size.

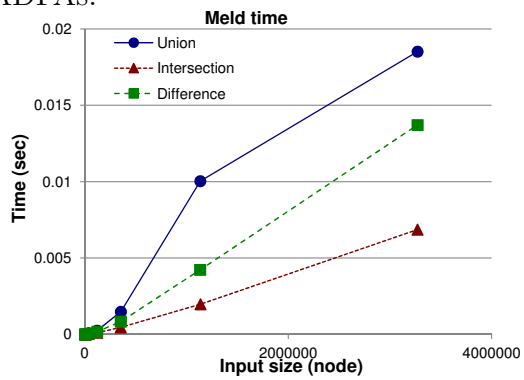


Figure 3.9: Exp 3. Computation time of $\mathsf{Meld}_\diamond$ for $\diamond \in \{\cup, \cap, \setminus\}$ with increasing sum of input sequence BDD sizes.

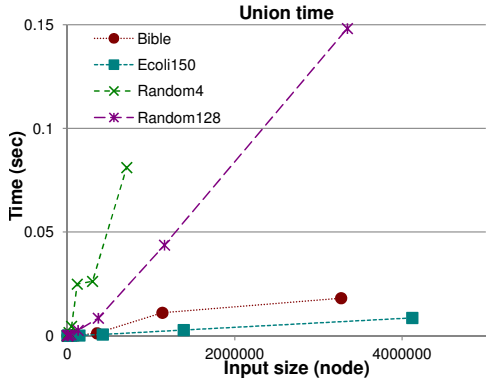


Figure 3.10: Exp 3. Computation time of $\mathsf{Meld}_\cup$ with increasing sum of input sequence BDD sizes.

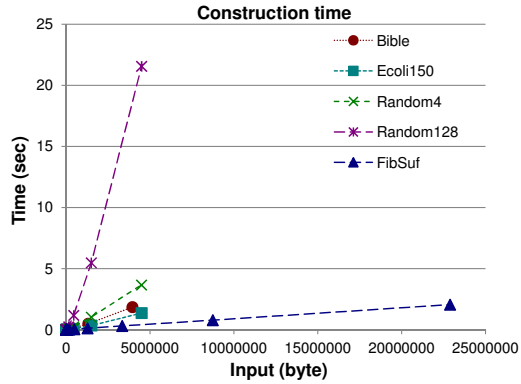


Figure 3.11: Exp 4. Computation time of Construct with increasing size of a data set.

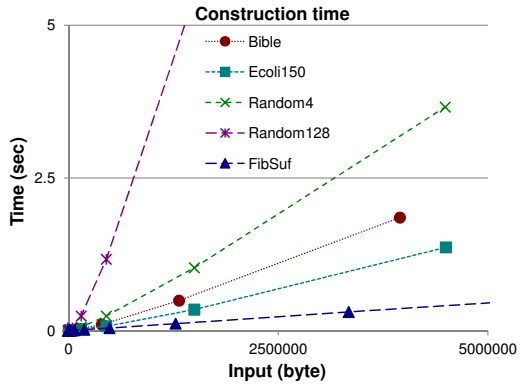


Figure 3.12: Exp 4. Computation time of Construct with increasing size of a data set (enlarged view).

Chapter 4

Complete Inverted Files on Sequence Binary Decision Diagrams

This chapter studies complete inverted files. Complete inverted files are abstruct types of data structure to support string searching. For a given text and a given pattern, complete inverted files answer whether the pattern occurs in the text as a substring, how many times the pattern occurs, and where the pattern occurs. For decades, complete inverted files for strings are widely studied and many efficient data structures are proposed. Directed acyclic graphs may represent a exponentially large number strings in comparison to their graph size. Therefore, constructing complete inverted files for directed acyclic graphs without expanding them is an important problem. However, if we consider a directed acyclic graph as an input instead of ordinay string, complete inverted files for that have not been studied well. We propose complete inverted files based on sequence BDDs for both string input and directed acyclic graph input.

4.1 Introduction

Recent emergence of massive text and sequence data has been increased the importance of string processing technologies. In particular, complete inverted files for efficient text retrieval and analysis has attracted much attention in many applications such as bioinformatics, natural language processing, and sequence mining. A complete inverted file for a text w is a data structure that stores all factors of w allowing three functions; *find*, *freq*, and *locations*. The *find* returns wheter a pattern occurs in a text or not. The *freq* returns the number of occurences of the pattern. And, the *locations* returns the all positions where the pattern occurs. In many real applications, indices that store occur-

rence information are highly required. Sequence binary decision diagrams are compact representation for manipulating sets of strings, proposed by Loekito, *et al.* [31]. In this chapter, we consider the problem of constructing a complete inverted file on sequence BDD framework. We define complete inverted files on SeqBDDs, named SeqBDD-FP (See Fig. 4.1), and propose an algorithm to construct a SeqBDD-FP from an input text. We also define a complete inverted file for a directed acyclic graph (DAG) and present an efficient construction algorithm to construct a SeqBDD-FP for a input DAG, which is given as an SeqBDD. We can construct complete inverted files for multiple texts using existing data structures by simply concatenating all these texts. However, those methods cannot deal with a very large number of strings that DAGs can represent by sharing its subgraphs. For example, regular expressions without infinite loop and human genomes with many replacements can be represented much more compactly by DAGs than by explicit representations. Siren *et al.* proposed general compressed suffix array (GCSA)[52] that is an index for DAG using succinct data structures. However, there is no other research on complete inverted files on graphs. We also show some experimental results for real data, and compare our SeqBDD-FP and GCSA. Our method will be useful for fast pattern matching applications and sequence mining.

4.2 Preliminaries

Let $\Sigma = \{a, b, \ldots\}$ be a countable alphabet of symbols. We denote the reversed string of a string $x \in \Sigma^*$ by $x^R = x[|x|] \cdots x[1]$. For a string s, if $s = xyz$ for $x, y, z \in \Sigma^*$, then we call x, y, and z a *prefix*, a *factor*, and a *suffix* of s, respectively. The sets of prefixes, factors, and suffixes of a string s are denoted by $\mathit{Prefix}(s)$, $\mathit{Factor}(s)$, and $\mathit{Suffix}(s)$, respectively. Given a set S of strings, let the sets of prefixes, factors, and suffixes of the strings in S be denoted by $\mathit{PREFIX}(S)$, $\mathit{FACTOR}(S)$, and $\mathit{SUFFIX}(S)$, respectively.

4.2.1 Complete inverted file

The notion of an inverted file for a textual database is common in the literature on information retrieval, but precise definitions of this concept vary. We use the following definition. Given a finite alphabet Σ, and a text word $w \in \Sigma^*$, a complete inverted file for (Σ, w) is an abstract data type that implements the following functions:

- (1) *find*: $\Sigma^* \to Factor(w)$, where $find(x)$ is the longest prefix y of x such that y occurs in w, that is, $x = yz$, $x, y, z \in \Sigma^*$, and y is a factor of a text w.
- (2) *freq*: $Factor(w) \to \mathbb{N}$, where $freq(x)$ is the number of times x occurs as a factor of the text w.
- (3) *locations*: $Factor(w) \to \mathbb{N}^*$, where $locations(x)$ is the set of end positions within the text in which x occurs.

In this chapter, we consider the problem of constructing a complete inverted file for a text w. The function $locations(x)$ returns the sequence BDD that represents set of integers $epos_w(x)$ as set of binary strings in our method. We describe sequence BDDs can implement complete inverted files compactly.

Example 6 *Let $w = abaababa$ be a given text. Then, $find(baabbaab) = baab$, $freq(ba) = 3$, and $locations(ba) = \{3, 6, 8\}$.*

4.3 SeqBDD-FP for a string

We begin with a brief look at some aspects of the substring index of a fixed, arbitrary string w. In particular, for each factor x of w we will be interested in the set of positions in w at the ends of occurrences of x. We describe the basic data structure used to implement a complete inverted file for a text w based on a sequence BDD.

In our method, occurrence positions are represented as a set of binary strings instead of a simple list of integers. If a factor x occurs at position i, our inverted file stores $x \cdot binstr(i)$. That is, a factor x of w is followed by its occurrence positions in the complete inverted file. Then, we can know the occurrences of x after traversing the path corresponding to x. All equivalent subgraphs are online minimized automatically by always using Getnode when a node with some triple is needed. Therefore, the subgraphs which represent binary strings also share their equivalent subgraphs and become compact.

Definition 11 *Let w be any string. Then, we define two languages.*

- $\mathcal{L}_{epos}(w) = \{x \cdot binstr(k) : x \in Factor(w), k \in epos_w(x)\}$,
- $\mathcal{L}_{bpos}(w) = \{x^R \cdot binstr(k) : x \in Factor(w), k \in bpos_w(x)\}$.

Definition 12 *The sequence BDD for factors and positions (SeqBDD-FP) of $w \in \Sigma^*$ is the sequence BDD*
$F = \langle \Sigma \cup \{0,1\}, V, \tau, \mathbf{0}, \mathbf{1}, \boldsymbol{r} \rangle$ *such that* $L(\boldsymbol{r}) = \mathcal{L}_{epos}(w)$.

The SeqBDD-FP for $w = abcbc$ is given in Figure 4.1. The "factor part" of the sequence BDD for a string w is equivalent to the directed acyclic word graph (DAWG) [4] of w, whose nodes are defined based on the equivalence class of substrings. Note that the sequence BDDs that represent binary strings play a role like the identification pointers in the compact DAWG [5].

Theorem 14 *Using SeqBDD-FP* $F = \langle \Sigma \cup \{0,1\}, V, \tau, \mathbf{0}, \mathbf{1}, \boldsymbol{r} \rangle$ *for a string* $w \in \Sigma^*$, *for any string* $x \in \Sigma^*$, $y = find(x)$ *can be determined in time* $O(|\Sigma||x|)$. *For any* $x \in Factor(w)$, $freq(x)$ *can be determined in time* $O(|\Sigma||x|)$ *if* $\mathsf{Card}(\boldsymbol{r})$ *is already executed at least once. Otherwise, it takes* $O(M)$ *time where* M *is the number of nodes reachable from the node representing* $locations(x)$. *And,* $locations(x)$ *can be done in* $O(freq(x) \log n)$ *time.*

Proof: To implement *find*, we begin at the root $\mathbf{r}$ and trace a path corresponding to the letters of x as long as possible. This "search path" is determined and continues until the longest prefix y of x in $Factor(w)$ has been found. To implement *freq*, we note that $freq(x) = |\{z : xz \in \mathcal{L}_{epos}(w)\}| = |epos_w(x)|$ for any $x \in Factor(w)$. The algorithm Card computes the cardinality of the language that each sequence BDD node represents and stores each result in *cache* [28]. So, $freq(x)$ can be obtained by following the procedure of *find* and then returning the result of Card of the node stored in the *cache*. $\mathsf{Card}(\boldsymbol{r})$ is executed in linear time to the input sequence BDD size. Since this node represents the language $M = \{z : xz \in \mathcal{L}_{epos}(w)\}$, we can obtain the node that represents $\{b : b \in M, b \in \{0, 1\}^*\}$ by traversing 0-edges until getting a node labeled by *0* or *1*. Clearly all queries are $O(|\Sigma||x|)$. □

Our algorithm to construct $\mathsf{BuildSeqBDD\text{-}FP}$ is described in Fig. 4.2. The rough space complexity and time complexity to construct $\mathsf{SeqBDD\text{-}FP}$ are both $O(n^2 \log n)$. The union operation is computed in $O(|P||Q|)$ time for two sequence BDDs P and Q [17, 15]. In fig. 4.3, we show the algorithm $\mathsf{BinSeqBDD}$(k) that constructs a sequence BDD that

represents a binary representation of a natural number k. That is, $L(\mathsf{BinSeqBDD}(k))$ is $\{binstr(k)\}$. We can also construct a sequence BDD for $\mathcal{L}_{bpos}$ with some modification of BuildSeqBDD-FP. That is swapping $|w|$ with 0 in line 1 and line 5, and changing the for loop in line 2 from descending order $|w|, \ldots, 1$ to ascending order $1, \ldots, |w|$.

For a given text w and its factor x, it takes $O(freq(x) \log |w|)$ time to compute occurrence list of x after obtaining the sequence BDD for $locations(x)$, because occurrences are represented as binary strings and every node has just one label. On the other hand, there are advantages due to sequence BDD representation, especially when $freq(x)$ is large. A list of integers in ordinary representation requires $O(freq(x))$ space and time to examine all positions. By sharing structures, these positions can be represented compactly in our method. Compared with an ordinary list merging technique, integer set intersection can be done efficiently by this representation. For example, for given two factors x and y, finding the positions that both occur within l symbols is computed with some modifications. At first, we construct the sequence BDD for $\mathcal{L}'_{epos}(w) = \{x \cdot binstr(k+j) : x \in Factor(w), k \in epos_w(x), 0 \leq j \leq l\}$. Next, obtain the sequence BDDs for $locations(x)$ and $locations(y)$. Then, the positions we want are computed by the intersection operation of these two sequence BDDs.

4.4 SeqBDD-FP for a DAG

We now show our algorithm that constructs a complete inverted file for a directed acyclic graph given as an sequence BDD. When we want to construct indices for sets of strings, we can build SeqBDD-FP with pairs of string IDs and positions instead of just positions. However, using node IDs is an easier way to implement that after constructing DAGs for given sets of strings. There is another case that original inputs are given as DAGs. In such case, constructing indices after expanding the DAGs require much time. Therefore, we need direct computation method of SeqBDD-FP from DAGs directly. First we note that the complete inverted file for an sequence BDD S is defined as follows. In our method, we use node identifiers (IDs) instead of positions for ordinary texts, and factors correspond to paths in the input sequence BDD.

Given an sequence BDD S, a complete inverted file for S for it is an abstract data type that implements the following functions:

- (1) *find*: $\Sigma^* \to FACTOR(L(S))$, where $find(x)$ is the longest prefix y of x such that y occurs in $L(S)$, that is, y is a factor of a string in $L(S)$.
- (2) *freq*: $FACTOR(L(S)) \to \mathbb{N}$, where $freq(x)$ is the number of nodes reachable by paths corresponding to x that begins from any nodes in S.
- (3) *locations*: $FACTOR(L(S)) \to \mathbb{N}^*$, where $locations(x)$ is the set of IDs of nodes in S to which paths lead that corresponding to x.

In our method, the set of node IDs that *locations* returns is represented by an sequence BDD for the set of binary strings of the IDs.

Let S be an sequence BDD. For any $x \in FACTOR(L(S))$, $enode_S(x)$ denotes the set of all IDs of nodes in S following the paths corresponding to x and traversing some 0-edges, $bnode_S(x)$ denotes the set of all IDs of nodes in S which represent a language M such that $x \in PREFIX(M)$.

Definition 13 *We define* $\mathcal{L}_{enode}(S) = \{x{\cdot}binstr(i) : x \in FACTOR(L(S)), i \in enode_S(x)\}$, *and* $\mathcal{L}_{bnode}(S) = \{x^R \cdot binstr(i) : x \in FACTOR(L(S)), i \in bnode_S(x)\}$. *The SeqBDD-FP for S is the sequence BDD G such that* $L(G) = \mathcal{L}_{enode}(S)$.

The SeqBDD-FP for an sequence BDD S such that $L(S) = \{aaab, aac, abc, bab\}$ is given in Fig. 4.7, and Fig. 4.6 shows the input sequence BDD.

Theorem 15 *Using SeqBDD-FP G, for any word $x \in \Sigma^*$, $y = find(x)$ can be determined in time $O(|\Sigma||x|)$. For any $x \in FACTOR(L(S))$, $freq(x)$ can be determined in time $O(|\Sigma||x|)$. Other two operations also require the same time complexity as in Theorem 14.*

Proof: We can implement *find*, *freq* and *locations* as in SeqBDD-FP for a text. □

Fig. 4.5 shows an algorithm to build the SeqBDD-FP for an sequence BDD S. The algorithm in Fig. 4.4 is used for prepocessing of constructing SeqBDD-FP. The basic action of the algorithm for an sequence BDD S is to construct the SeqBDD-FPs for sequence BDDs rooted by each node recursively, synchronized with the depth-first traversal of S. We can construct reversed version of the SeqBDD-FP. It allows for the computation of the exact number of paths corresponding to queries. It also allows for returning the node IDs at which the paths begin. Such an sequence BDD is constructed by executing

BuildSeqBDD-FP_G after appliying the algorithm that construct an sequence BDD for reversed $L(S)$, which is proposed by Aoki *et al.* [2].

First, we append sequence BDDs for node IDs to the input by AppendID. Next, we construct reversed sequence BDD of it, but we do not reverse the sequence BDDs that represent node IDs as binary strings. Then, we can construct the sequence BDD for $\mathcal{L}_{bnode}(S)$ by execute BuildSeqBDD-FP_G0 on the obtained sequence BDD. The space and time complexity to construct SeqBDD-FP for a DAG are still unknown.

4.5 Experimental Results

Setting: In the experiments, we used the following data sets. As real data sets, we used `E.coli`, `bible.txt`, and `world192.txt` obtained from the Canterbury corpus[1]. From these data sets, we obtained the following derived data sets: BibleAll is the set of all lines drawn from `bible.txt`. Ecoli150 and Ecoli500 are the set of factors drawn from `E.coli` by cutting the whole sequence at every 150-th or 500-th letter, respectively. We made subsets of these data sets by randomly taking l lines varying $l = 10, 30, 100, \ldots$ for BibleAll, Ecoli150, and Ecoli500.

We implemented our shared and reduced sequence BDD environment on the top of the *SAPPORO BDD package* [40] for BDDs and ZDDs written in C and C++, where each node is encoded in a 64-bit integer and a node triple occupies approximately 50 to 55 bytes on average including hash entries in *uniqtable*. We performed experiments on a machine that consists of eight quad-core 3.1 GHz Intel Xeon CPU E7-8837 SE processors (i.e, 32 CPU cores in total) and 1 TB DDR2 memory shared among cores. To construct SeqBDD-FP both for strings and for DAGs, we implemented BinSeqBDD, BuildSeqBDD-FP, AppendID, and BuildSeqBDD-FP_G.

Experiment 1: SeqBDD-FP construction for a string. First, Fig. 4.8, and Fig. 4.9 show the results. From Fig. 4.8 and Fig. 4.9, the size of SeqBDD-FPs and running time of SeqBDD-FP looks like $O(n)$ or $O(n \log n)$ for n length text. The number of nodes are between $12n$ to $15n$.

Experiment 2: SeqBDD-FP construction for a DAG. Fig. 4.10 demonstrates that the SeqBDD-FPs are close to linear in the size of the input sequence BDDs. The

[1] `http://corpus.canterbury.ac.nz/descriptions/#cantrbry`

Table 4.1: Comparison of size

input DAG	#nodes	#seq.	size (bytes)	
			GCSA	SeqBDD-FP
SNP1000	1,001,000	2^{1000}	1,256,160	472,910,400
SNP100	1,010,000	2^{10000}	1,299,990	498,199,440
SNP10	1,100,000	2^{100000}	1,572,880	767,679,480

number of nodes are almost twice as that of the input sequence BDD. As can be seen from Fig.4.11, BuildSeqBDD-FP_G seems to run in almost $O(N \log N)$ time for N sized input sequence BDDs, in this experiments.

Experiment 3: Comparison with General Compressed Suffix Array. In Table. 4.1 and 4.2, we show the size of the input DAGs, the time to construct each data structure, and the time to execute find operation. We used published implementation of GCSA [2] for comparison. The input DAGs *SNPk* are generated artificially. Its alphabet is $\{A, C, G, T\}$. They are almost linear strings of length 1,000,000, but there are letters that can be changed another one (SNPs) per k letters. For example, *SNP8* can be ACGT(A—T)GCTCAG(C—T)GA$\cdots$. We executed *find* operation on both data structure to search 1,000,000 queries of length 10 that occur in the input DAGs. We assume a sequence BDD node requires 30 bytes. We can see that GCSA is 375 to 490 times more compact than SeqBDD-FP. In addition, construction time of GCSA is 12 to 20 times shorter than that of SeqBDD-FP. On the other hand, *find* operations on SeqBDD-FP are done in 6 times shorter time than GCSA. It is interesting that more frequent SNPs make SeqBDD-FP construction slower, but GCSA construction faster. Since sequence BDD uses pointers much, sequence BDD is much larger than GCSA that is based on succinct data structure. However, GCSA is difficult to update because it is static data structure. And, SeqBDD-FP can be updated dynamically and efficiently by sequence BDD operations. Therefore, SeqBDD-FP is appropriate for fast searching and frequently updated data.

[2] `http://www.cs.helsinki.fi/group/suds/gcsa/`

Table 4.2: Comparison of performance

input DAG	construction time (sec)		find time (sec)	
	GCSA	SeqBDD-FP	GCSA	SeqBDD-FP
SNP1000	9.53	112.63	12.71	2.17
SNP100	7.43	125.98	13.24	2.18
SNP10	6.09	175.40	11.45	2.22

4.6 Conclusions

We proposed SeqBDD-FP that is a complete inverted file based on sequence BDD both for a string and for a DAG. They allow all queries to be solved in $O(|\Sigma||x|)$ time for n sized input. We gave algorithms that construct SeqBDD-FPs. From the experimental results, their sizes are compact and our algorithms BuildSeqBDD-FP and BuildSeqBDD-FP_G run in almost $O(n \log n)$ time. The exact size bound of SeqBDD-FP and the exact time complexity of our algorithms are not obvious. When compared with GCSA, SeqBDD-FP requires more space and time to construct. However, SeqBDD-FP allows faster searching and dynamic updating. To propose more efficient construction algorithms of SeqBDD-FP is our future work. We also should analyze the time and space complexity of SeqBDD-FP. Position restricted search with SeqBDD-FP is also a challenging problem.

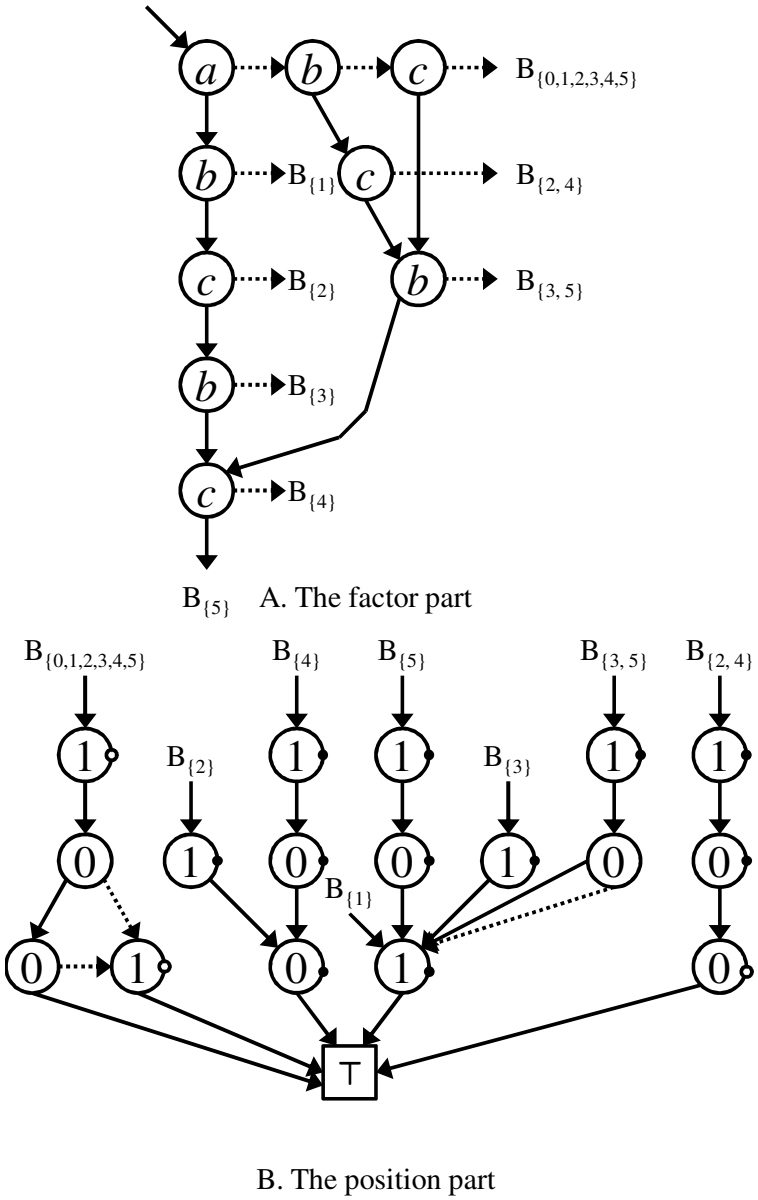


Figure 4.1: An example of a complete inverted file based on sequence BDD, SeqBDD-FP, for a string $w = abcbc$. The 0-terminal **0** is omitted. The dotted arrows represent 0-edges, and the solid arrows represent 1-edges. All 0-edges coming to **0** and **1** are indicated by a small black dot and white dot on the right side, respectively.

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

Proc BuildSeqBDD-FP(w: string):

Return: F: SeqBDD-FP for w;

1: $P_{|w|} \leftarrow$ BinSeqBDD($|w|$);
2: $F_{|w|} \leftarrow P_{|w|}$;
3: **for** $i = |w|, \ldots, 1$
4: $\quad P_{i-1} \leftarrow$ Getnode($w[i]$, BinSeqBDD($i-1$), P_i);
5: $\quad F_{i-1} \leftarrow$ Union(F_i, P_i);
6: **return** F_0;

Figure 4.2: An algorithm BuildSeqBDD-FP for constructing the SeqBDD-FP of an input string w.

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

Proc $\mathsf{BinSeqBDD}(k$: natural number):

Return: B: Sequence BDD such that $L(B) = \{binstr(k)\}$;

1: **return** $\mathsf{BinSeqBDD0}(k, \lfloor \log_2(k+1) \rfloor)$;

Proc $\mathsf{BinSeqBDD0}(k, l$: natural number):

Return: B: Sequence BDD that $L(B) = \{l$ length binary string of $k\}$;

1: **if** $(l = 0)$ **return 1**;
2: **else if** $(B \leftarrow cache[\text{“}BinSeqBDD(k,l)\text{”}]$ exists) **return** B;
3: **else**
4: **if** $(k\&(1 << l) \neq 0)$
5: $B \leftarrow \mathsf{Getnode}(1, \mathbf{0}, \mathsf{BinSeqBDD0}(k\&((1 << l) - 1), l - 1))$;
6: **else** $B \leftarrow \mathsf{Getnode}(0, \mathbf{0}, \mathsf{BinSeqBDD0}(k\&((1 << l) - 1), l - 1))$;
7: $cache[\text{“}BinSeqBDD(k,l)\text{”}] \leftarrow B$;
8: **return** B;

Figure 4.3: An algorithm $\mathsf{BinSeqBDD}$ for constructing the sequence BDD for $\{binstr(k)\}$. Bitwise AND operation and bit left shift operaton are denoted by & and <<, respectively.

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

Proc AppendID(P: Sequence BDD):

Return: R: Sequence BDD such that

$L(R) = \{x \cdot binstr(Q.ID) : x \in L(P), Q$ is a sequence BDD node reachable from root via the path corresponding to x and following 0-edges$\}$;

```
1: if (P = 0) return BinSeqBDD(0);
2: else if (P = 1) return BinSeqBDD(1);
3: else if (R ← cache["AppendID(P)"] exists) return R;
4: else
5:     ⟨x, P_0, P_1⟩ ← τ(P);
6:     R ← Union(Getnode(x, AppendID(P_0), AppendID(P_1)), BinSeqBDD(P.id));
7:     cache["AppendID(P)"] ← R;
8: return R;
```

Figure 4.4: An algorithm AppendID for constructing the sequence BDD with node IDs by binary strings.

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

Proc BuildSeqBDD-FP_G(S: Sequence BDD):

Return: F: Position SeqBDD-FP for S;

1: **return** BuildSeqBDD-FP_G0(AppendID(S));

Proc BuildSeqBDD-FP_G0(P: Sequence BDD):

Return: G: Sequence BDD such that

$L(G) = \{z : z \in SUFFIX(L(P)),\ z \in \Sigma^+ \cdot \{0, 1\}^*\}$;

1: **if** ($P = \mathbf{0}$ or $P = \mathbf{1}$) **return** P;

2: **else if** ($G \leftarrow cache[$"*BuildSeqBDD-FP_G*(P)"$]$ exists) **return** G;

3: **else**

4: $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;

5: **if** ($x \in \{0, 1\}$) **return** P;

6: $G \leftarrow$ BuildSeqBDD-FP_G0(P_0) $\cup$ BuildSeqBDD-FP_G0(P_1) $\cup$Getnode($x, \mathbf{0}, P_1$) ;

7: $cache[$"*BuildSeqBDD-FP_G*(P)"$] \leftarrow G$;

8: **return** G;

Figure 4.5: An algorithm constructs the SeqBDD-FP for the input sequence BDD S. Union operations are denoted by $\cup$.

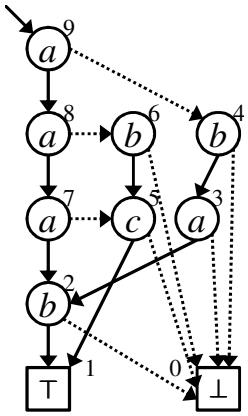


Figure 4.6: An sequence BDD for $\{aaab, aac, abc, bab\}$. The dotted arrows represent 0-edges, and the solid arrows represent 1-edges. Node IDs are given on the side of each nodes.

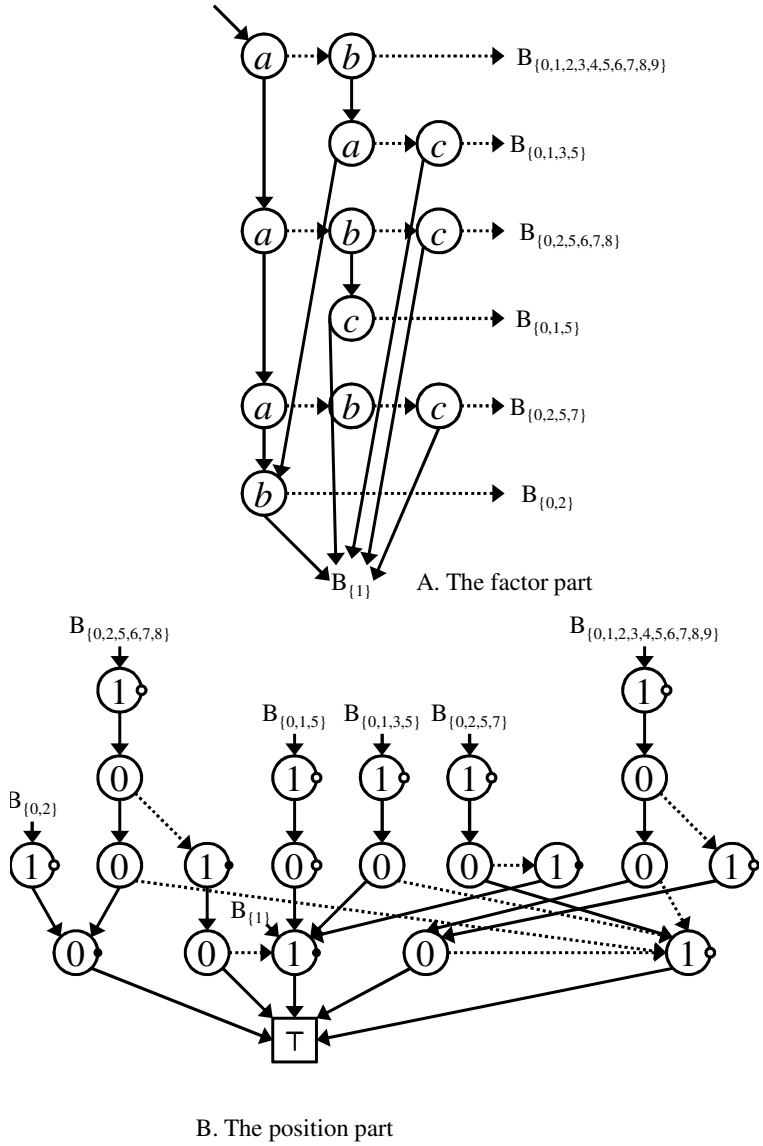


Figure 4.7: An example of a SeqBDD-FP for the sequence BDD in Fig. 4.6. The 0-terminal **0** is omitted. The dotted arrows represent 0-edges, and the solid arrows represent 1-edges. All 0-edges incoming to **0** and **1** are indicated by a small black dot and white dot on the right side of a node, respectively.

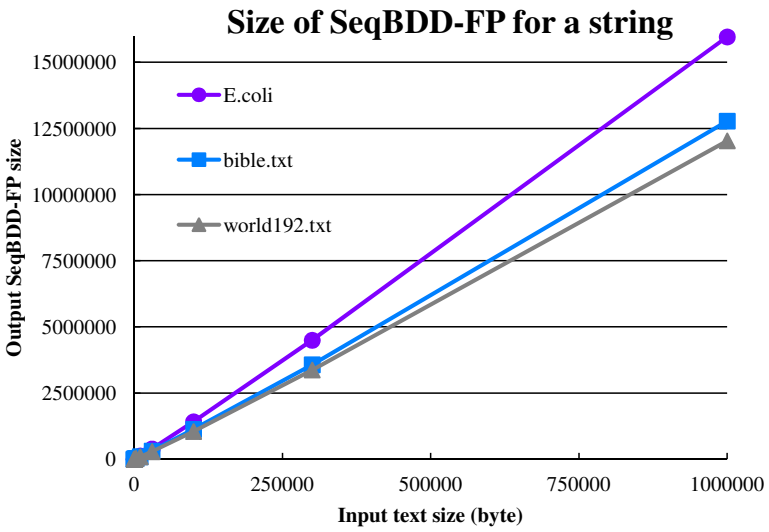


Figure 4.8: Sequence BDD size of SeqBDD-FP with increasing length of input string.

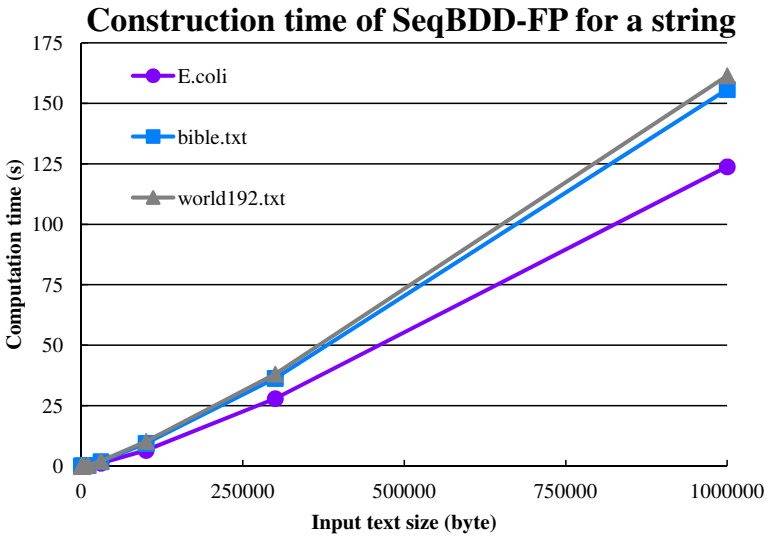


Figure 4.9: Computation time of BuildSeqBDD-FP for a string with increasing length of input string.

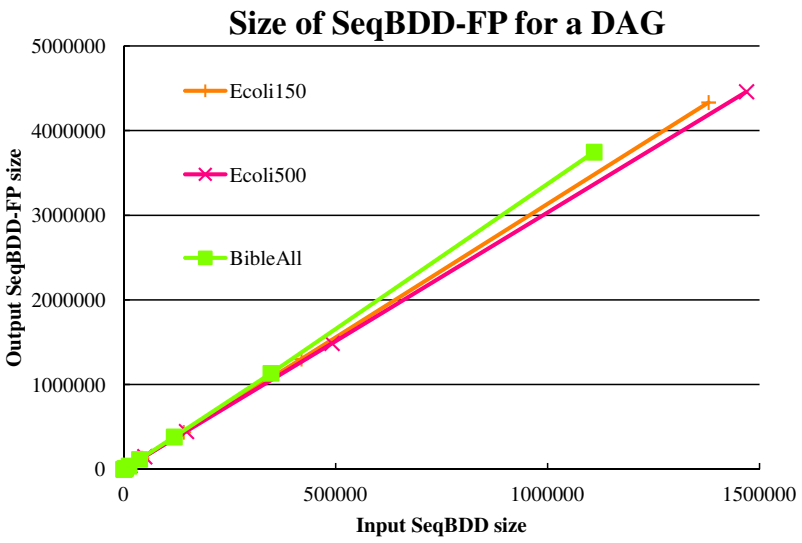


Figure 4.10: Sequence BDD size of SeqBDD-FP for a DAG with increasing input sequence BDD size.

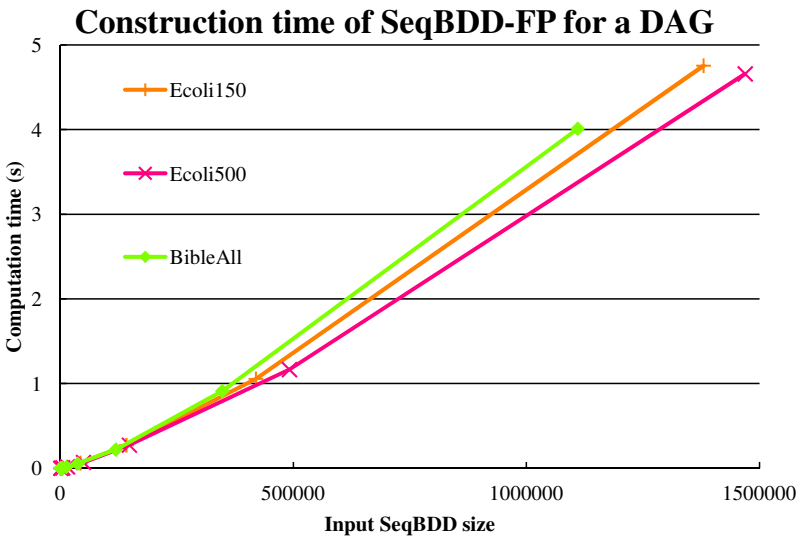


Figure 4.11: Computation time of BuildSeqBDD-FP_G with increasing input sequence BDD size.

Chapter 5

Compact and Fast Zero-suppressed Binary Decision Diagrams

In this chapter, we study indices for families of sets. Our goal is to make a data structure that compactly store families of sets, have fast algorithm to search a given set, and allow listing up the all sets stored in. Zero-suppressed BDD is a nice data structure to manipulate familes of sets. However, it has some problems. First problem is that ZDD requires much memory space because it heavily uses pointers. Second problem is that searching a set on ZDD is slow when the ZDD represents a large sparse family of sets. Large sparse family of sets means that the number of sets in the family is many but relatively small in comparison to the number of possible sets in the universal sets when the size of the universal set is large. Such data sets frequently appear in practical applications. Since we have to traverse pointers to continue searching, we need to visit nodes labeled with items we do not interested in. To solve these problems, we employ succinct data structures. A succinct data structure stores objects using space asymptotically equal to the information-theoretic lower bound, while simultaneously supporting a number of primitive operations on the objects in constant time. First, we propose a method to represent a given ZDD by combination of three succinct data structurs. Next, we show how to convert the ZDD to our proposal method. We evaluate the power of our method by some computational experiments.

5.1 Introduction

Among unique canonical representations of Boolean functions, BDDs are smaller than others such as CNF, DNF, and truth tables for many classes of functions. BDDs have

the following features:

- Boolean functions are uniquely represented like other representations.
- Multiple functions are stored compactly by sharing common subgraphs.
- Fast logical operations are executed on Boolean functions.

Zero-suppressed Binary Decision Diagrams (ZDDs) [38] are variation of traditional BDDs, used to manipulate families of sets. Using ZDDs, we can implicitly enumerate combinatorial item set data and efficiently compute set operations over the ZDDs.

Though BDDs are more compact than other representations of Boolean function and set families, they are still large; a node of a BDD uses 20 to 30 bytes depending on implementations [40]. BDDs become inefficient if the graph size is too large to be held in memory. Therefore the aim of this chapter is to reduce the size (number of bits) used to represent BDDs. We classify implementations of BDDs into three types:

- Dynamic: The BDD can be modified. New nodes can be added to the BDD.
- Static: The BDD cannot be modified. Only query operations are supported.
- Freeze-dried: All the information of the BDD is stored, but it cannot be used before restoration.

Most of the current implementations of BDDs are dynamic. There is previous work on freeze-dried representations of BDDs by Starkey and Bryant [53] and later, by Mateu and Prades-Nebot [36]. Hansen, Rao and Tiedemann [21] developed a technique to compress BDD and reduce the size of the BDD to 1-2 bits per node. However there is no implementation of BDDs that is specialized for static case.

In addition, the existing BDDs have another problem. When we store a huge sparse family of sets on a large number of items, it takes much time to check whether a given set is included in the family. That is, membership operation on existing BDDs needs time linear to the number of all items in the worst case because we have to traverse nodes one by one to find the node labeled with the item we interested in and those nodes cannot be skipped.

This is the first to propose a static representation of ZDDs, which we call *DenseZDDs*. We show trade-off results for dynamic operations and compactness with fast membership operation. The size of ZDDs in our representation is much smaller than an existing dynamic representation [40]. Not only compact, DenseZDD supports much faster membership operations than [40]. Membership operations on DenseZDDs are dramatically speeded up especially for higher BDDs, that each set in the BDDs can be small but the total number of items are huge, because we can jump into nodes labeled with desired item in constant time on DenseZDDs. Thus, the worst case time complexity of membership operations are improved from the time linear to the total number of items to the time linear to the size of set we want to check its existence. For example, a collection of items bought by many customers in a supermarket is such a data. Experimental results show that DenseZDDs are five times smaller and membership queries are twenty to several hundred times faster, compared to [40]. Note that our technique can be directly applied to compress traditional BDDs too.

5.2 Preliminary

5.2.1 Succinct data structures for rank/select

Let B be a binary vector of length u, that is, $B[i] \in \{0, 1\}$ for any $0 \leq i < u$. The rank value $rank_c(B, i)$ is defined as the number of c's in $B[0..i]$, and the select value $select_c(B, j)$ is the position of j-th c ($j \geq 1$) in B from the left. Note that $rank_c(B, select_c(B, j)) = j$ holds if $j \leq rank_c(B, n-1)$, the number of c's in B. The predecessor $pred_c(B, i)$ is defined as the position j of the rightmost $c = B[j]$ to the left of $B[i]$. The predecessor is computed by $pred_c(B, i) := select_c(B, rank_c(B, i))$.

The Fully Indexable Dictionary (FID) is a data structure for computing rank and select on binary vectors [50].

Theorem 16 (Raman et al. [50]) *For a binary vector of length u with n ones, its Fully Indexable Dictionary uses $\binom{u}{n} + \mathcal{O}(u \log\log u / \log u)$ bits of space and computes $rank_c(B, i)$ and $select_c(B, i)$ in constant time on the $\Omega(\log u)$-bit word RAM.*

This data structure uses asymptotically optimal space because any data structure for storing the vector uses $\lceil \binom{u}{n} \rceil$ bits in the worst case. Such a data structure is called a *succinct data structure*.

5.2.2 Succinct data structures for trees

An ordered tree is a rooted unlabeled tree such that children of each node have some order. A succinct data structure for an ordered tree with n nodes uses $2n + o(n)$ bits of space and supports various operations on the tree such as finding the parent or i-th child, computing the depth or the preorder of a node, etc., in constant time [47]. An ordered tree with n nodes is represented by a string of length $2n$ called a balanced parentheses sequence (BP), defined by a depth-first traversal of the tree. Starting from the root, we write an open parenthesis '(' if we arrive at a node from above, and a close parenthesis ')' if we leave from a node upward.

In this chapter, the following operations are used. Let P denote the BP sequence of a tree. A node is identified with the position of the open parenthesis in P representing the node.

- $\mathit{depth}(P, i)$: the depth of a node at position i. (The depth of a root is 0.)
- $\mathit{preorder}(P, i)$: the preorder of a node at position i.
- $\mathit{level\_ancestor}(P, i, d)$: the position of the ancestor with depth d of node i.
- $\mathit{parent}(P, i)$: the position of the parent of node i (identical to $\mathit{level\_ancestor}(P,\ i,\ \mathit{depth}(P, i) - 1)$).
- $\mathit{degree}(P, i)$: the number of children of node i.
- $\mathit{child}(P, i, d)$: the d-th child of node i.

The operations take constant time.

A brief overview of the data structure is the following. The BP sequence is partitioned into equal-length blocks. The blocks are stored in leaves of a rooted tree called range min-max tree. In each leaf of the range min-max tree, we store the maximum and the minimum values of node depths in the corresponding block. In each internal node, we store the maximum and the minimum of values stored in children of the node. By using this range min-max tree, all tree operations are implemented efficiently.

5.2.3 Problems of existing ZDDs

Let m be the number of nodes of a given ZDD and n be the number of distinct indices of nodes. Existing ZDD implementations have the following problems. First, they require too much memory to represent a ZDD. Second, the $member(v, S)$ operation is too slow, needing $\Theta(n)$ time in the worst case. In practice, the size of query sets is usually much smaller than n, and so an $\mathcal{O}(|S|)$ time algorithm is desirable. However it is impossible to attain this in the current implementation [40] because the $member(v, S)$ operation is implemented by using the $zero(v)$ operation repeatedly.

For example, we traverse 0-edges 255 times when we search $S = \{e_1\}$ on the ZDD for $F = \{\{e_1\}, \ldots, \{e_{256}\}\}$. If we translate a ZDD to an equivalent automaton by using an array to store pointers (see Fig. 5.1), we can execute searching in $\mathcal{O}(|S|)$ time. ZDD nodes correspond to labeled edges in the automaton. However, the size of such automaton via a straightforward translation can be $\Theta(n)$ times larger than the original ZDD [17, 15] in the worst case. Therefore, we want to perform $member(v, S)$ operations in $O(|S|)$ time on ZDDs.

Minato proposed Z-Skip Links [41] to accelerate the traversal of ZDDs of large-scale sparse datasets. His method adds one link per node to skip nodes that are concatenated by 0-edges. Therefore the memory requirement of this augmented data structure cannot be smaller than original ZDDs. Z-Skip-Links make membership operations much faster than using conventional ZDD operations when handling large-scale sparse datasets. However, the computation time is probabilistically analyzed only for average case.

5.3 Data Structure

In this section, we describe our data structure *DenseZDD* which solves the two problems defined in Section 5.2.3. We obtain the following results.

Theorem 17 *Let u be the size of the ZDD that removes the zero-suppress rule only for nodes pointed to by 0-edges. A ZDD with m nodes on n items can be stored in $m \log u + 2(m + u) + o(u)$ bits so that the primitive operations except $getnode(i, v_0, v_1)$ are done in constant time. In other words, u is the size of the ZDD with dummy nodes that are described below. The $getnode(i, v_0, v_1)$ operation is done in $\mathcal{O}(\log m)$ time.*

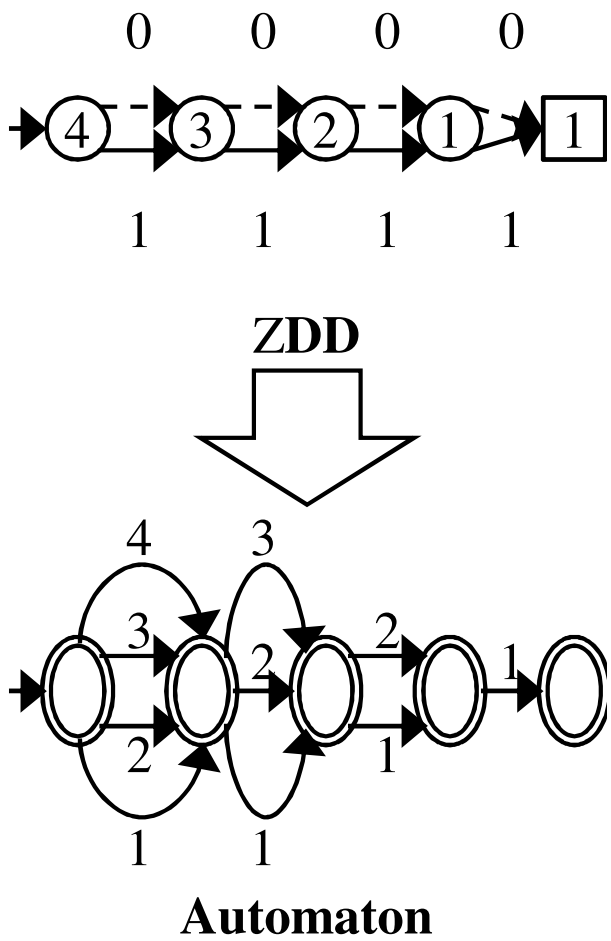


Figure 5.1: Worst-case example of a straightforward translation

Proof: We first prove Theorem. 17. From the above discussion, the BP U of zero-edge tree costs $2u = \mathcal{O}(mn)$ bits where u is the size of corresponding quasi-ZDD. The one-child array needs $m \log m$ bits for 1-children and m bits for $\emptyset$-flags. Using FID, the dummy node vector is stored in $m(1 + \log u) + o(u)$ bits. Therefore, the DenseZDD can be stored in $m \log u + 2(m + u) + o(u)$ bits and primitive operations except *getnode* are done in constant time because the $rank_1$, $select_1$, and any tree operations take constant time. Since the *getnode* finds a target node by binary search, it takes $\mathcal{O}(\log m)$ (described in Sec. 5.5).

Theorem 18 *A ZDD with m nodes on n items can be stored in $\mathcal{O}(m(\log m + \log n))$ bits so that the primitive operations are done in $\mathcal{O}(\log m)$ time except getnode(i, v_0, v_1). The getnode(i, v_0, v_1) operation is done in $\mathcal{O}(\log^2 m)$ time.*

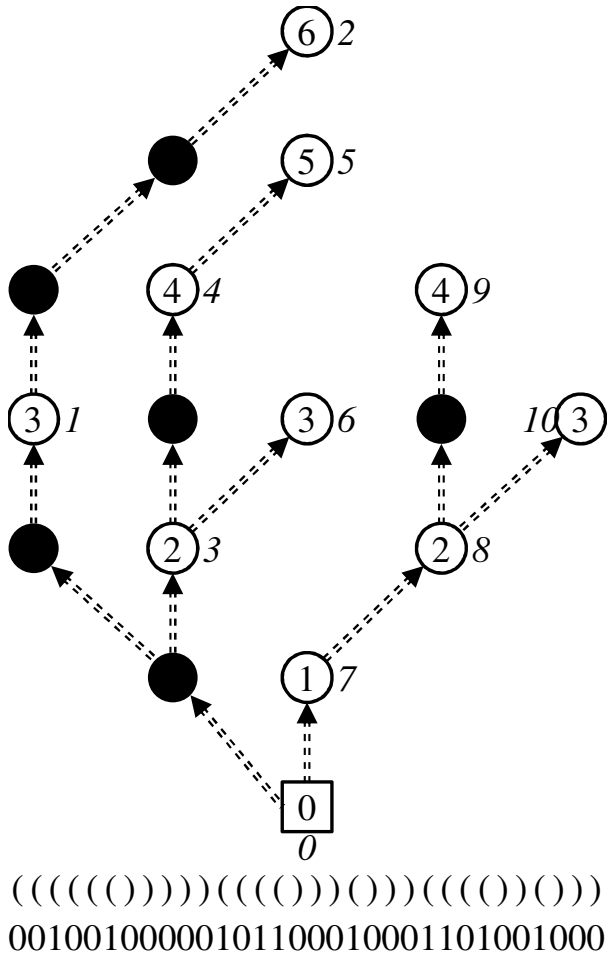
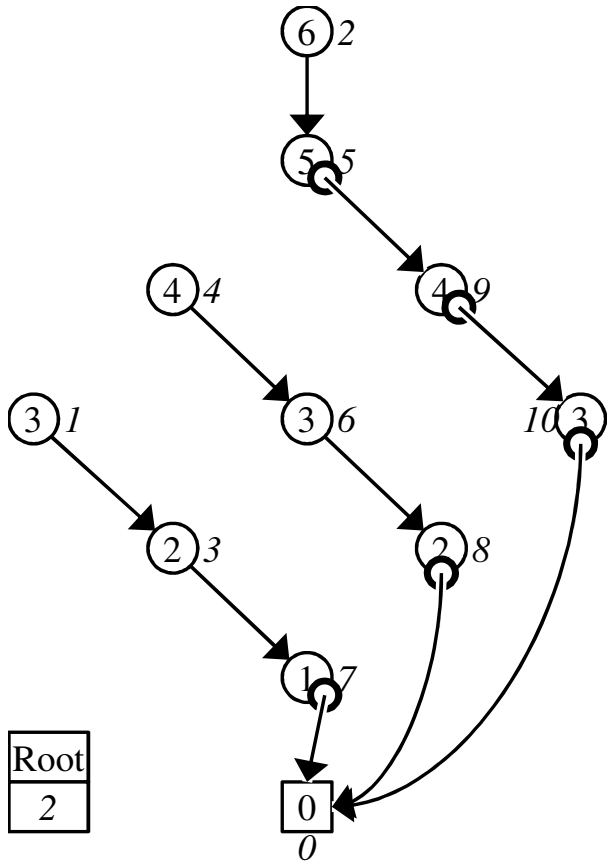


Figure 5.2: A zero-edge tree and a dummy node vector obtained from the ZDD in Fig. 2.3

Proof: When we compress U, it can be stored in $\mathcal{O}(m \log n)$ bits and the min-max tree is stored in $\mathcal{O}(m(\log n + \log m)/\log m)$ bits. The dummy node vector can be compressed in $m(2 + \log m) + o(m)$ bits by FID with sparse array. But, the time order of any tree operations and the $rank_1$ operation is changed from constant time to $\mathcal{O}(\log m)$ time. Therefore, the DenseZDD can be stored in $\mathcal{O}(m(\log m + \log n))$ bits and primitive operations take $O(\log m)$ times larger than the above time because all of them use tree operations or $rank_1$ on M.

5.3.1 DenseZDD

A DenseZDD $DZ = \langle U, M, I \rangle$ is composed of three data structures: a zero-edge tree U, a dummy node vector M, and a one-child array I.



1	2	3	4	5	6	7	8	9	10
3	5	7	6	-9	8	0	0	-10	0

Figure 5.3: A one-child array obtained from the ZDD in Fig. 2.3

Zero-edge tree : The spanning tree of ZDD G formed by the 0-edges is called the *zero-edge tree* of G and denoted by T_Z. In a zero-edge tree, all 0-edges are reversed and the 0-terminal node becomes the root of the tree. The preorder rank of each node is used to identify it. Zero-edge trees are based on the same idea as left or right trees by Maruyama *et al.* [35].

An important difference between our structure and theirs is the existence of *dummy nodes*. We call nodes in the original ZDD as *real nodes*. We use the zero-edge tree with dummy nodes, denoted by T'_Z. We create dummy nodes on each 0-edge to guarantee that the depth of every real node v in the zero-edge tree equals $index(v)$. We define the depth of the 0-terminal node, the root of this tree, to be 0. Let U be the BP of T'_Z. The length of U is $\mathcal{O}(mn)$ because we create $n-1$ dummy nodes for one real node in the worst case. An example of a zero-edge tree and its BP are shown in Fig. 5.2. Black circles are dummy nodes and the number next to each node is its preorder rank. The 0-terminal

node is ignored in the BP because we know the root of a zero-edge tree is always that node.

Dummy node vector : A bit vector of the same length as U is used to distinguish dummy nodes and real nodes. We call it the *dummy node vector* of T'_Z and denote it by B_D. The i-th bit is 1 if and only if the i-th parenthesis of U is '(' and its corresponding node is a real node in T'_Z. An example of a dummy node vector is also shown in Fig. 5.2. The 0-terminal node is also ignored. Let the FID of B_D be M. Using M, we can determine whether a node is dummy or real, and compute preorder ranks among only real nodes. We can also obtain positions of real nodes on BP from their preorder ranks by the select operation on M.

One-child array : An integer array to represent the 1-child of each node is called the *one-child array* and denoted by C_O. This array contains node preorder ranks of all 1-children in preorder on T_Z. That is, its i-th element is the preorder rank of the 1-child of the nonterminal node whose preorder rank is i. We also require one bit for each element of the one-child array to store the $\emptyset$-flag. If $empflag(v) = 1$ for a nonterminal node v, the corresponding element in the one-child array will be negative. An example of a one-child array is shown in Fig. 5.3. Let I be the compressed representation of C_O. In I, one integer is represented by $\lceil \log(m+1) \rceil + 1$ bits, including one bit for the $\emptyset$-flag.

5.4 Algorithm

5.4.1 Conversion of an ordinary ZDD to a DenseZDD

We show how to construct the DenseZDD. We first build the zero-edge tree from the given ZDD. A pseudo-code is given in Fig. 5.6 in the appendix. The zero-edge tree consists of all 0-edges of the ZDD, with their directions being reversed. For a nonterminal node v, we say that v is a 0^r-child of $zero(v)$. To make a zero-edge, we use a list *revzero* in each node, which stores 0^r-children of the node. The lists for all the nodes are computed by a depth-first traversal of the ZDD. This is done in $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ space, since each node is visited at most twice and the total size of *revzero* is the same as the number of nonterminal nodes.

Global vaiables: $L_1, \ldots, L_n$ are list which are empty initially.

ALGORITHM Compute_Preorder (L_0)

Input: L_0: a list stores only $\langle \{\mathbf{0}\}, [0, \mathit{stsize}(\mathbf{0}) - 1] \rangle$;

```
for i = 0, ..., n
  for each ⟨A, [l, r]⟩ ∈ L_i in arbitrary order % A is a set of nodes
    for each v ∈ A in descending order of ⟨prank(one(v)), empflag(v)⟩
      prank(v) ← l++;
      for each j ∈ { j | w ∈ revzero(v), j = index(w) } in descending order
        A ← { w | w ∈ revzero(v), index(w) = j };
        r ← l + sum{ stsize(w) | w ∈ B };
        append ⟨B, [l, r]⟩ to L_j;
        % That is, the prank of descendants of nodes in B are in [l, r].
        l ← r + 1;
return;
```

Figure 5.4: An algorithm which computes the preorder rank $\mathit{prank}(v)$ of each node v. Sets of nodes are implemented by arrays or lists in this code. The $\mathit{prank}(\mathbf{0})$ is 0.

We obtained a zero-edge tree T, but it is not an ordered tree. We define preorder rank $\mathit{prank}(v)$ for every node v before traversal. The nodes in *revzero* are sorted in descending order of their pairs $\langle \mathit{index}, \mathit{prank} \rangle$, that is, $\mathit{index}(\mathit{revzero}[i]) \geq \mathit{index}(\mathit{revzero}[i+1])$ for $1 \leq i < |\mathit{revzero}(v)|$. Then, nodes with higher indices are visited first. This ordering is useful to reduce the number of dummy nodes and to implement ZDD operations simply. It seems impossible to define visiting order of nodes by preorder rank of their 1-children during computing preorder, but it is possible. Since a ZDD node v satisfies $\mathit{index}(v) > \mathit{index}(\mathit{zero}(v))$ and $\mathit{index}(v) > \mathit{index}(\mathit{one}(v))$, we can decide *prank* for every node by the pseudo code in Fig. 5.4, which is a BFS algorithm based on *index* value starting from 0-terminal. To compute *prank* efficiently, we construct the temporary BP for the zero-edge tree. Using the BP, we can compute the size of each subtree rooted by v in T in constant time and compact space.

Next, we create dummy nodes imaginarily. For a node v, we create $q = \max\{\, i \in \{1, \ldots, n\} \mid i = \mathit{index}(\mathit{revzero}[j]) - 1, 1 \leq j \leq |\mathit{revzero}(v)| \,\}$ dummy nodes $d_1, \ldots, d_q$ such

ALGORITHM Convert_ZDD_BitVectors $(v, paren, dummy, onechild)$

Input: ZDD node v, list of parentheses $paren$,

list of bits $dummy$, list of integers $onechild$

```
i = index(v);
for each w ∈ revzero(v) in ascending order of prank(w);
    while i + 1 < index(w)
        append '(' to paren, and '0' to dummy;
        ++i;
    append '(' to paren, and '1' to dummy;
    append prank(one(w)) · (−1 · empflag(w)) to onechild;
    Convert_ZDD_BitVectors(w, paren, dummy, onechild);
    append ')' to paren, and '0' to dummy;
while i > index(v)
    append ')' to paren, and '0' to dummy;
    −−i;
return;
```

Figure 5.5: Algorithm for obtaining the BP representation of the zero-edge tree, the dummy node vector, and the one-child array.

that $triple(d_i) = \langle index(v) + i, d_{i-1}, \mathbf{0} \rangle$, and $empflag(d_i) = 0, 1 \leq i \leq q$. For convention, d_0 denotes v.

To sum up, the DenseZDD for the given ZDD is composed of the zero-edge tree, the one-child array, and the dummy node vector. We traverse the zero-edge tree in DFS order as if dummy nodes exist and construct the BP representation U, the dummy node vector M, and the one-child array I. The BP and dummy node vector are constructed for the zero-edge tree with dummy nodes. On the other hand, the one-child array ignores dummy nodes. DenseZDD $DZ = \langle U, M, I \rangle$ is obtained. Pseudo-codes are given in algorithms in Fig. 5.5, and 5.6 in the appendix.

ALGORITHM Construct_DenseZDD (W: list of ZDD root)

Output: DenseZDD DZ

1: **for each** $v \in V$ compute *revzero* fields for all descendants of v;
2: compute *stsize* field for all 0^r-decsendants;
3: **Compute_Preorder**($\{\langle\{\mathbf{0}\}, [0, stsize(\mathbf{0}) - 1]\rangle\}$);
4: create empty lists $paren, dummy, onechild$;
5: append '(' to *paren*, and '0' to *dummy*;
6: **Convert_ZDD_BitVectors**($\mathbf{0}$, paren, dummy, onechild);
7: append ')' to *paren*, and '0' to *dummy*;
8: make BP U from *paren*;
9: make FID M from *dummy*;
10: make compressed representation I of *onechild*;
11: **return** $DZ \leftarrow \langle U, M, I\rangle$;

Figure 5.6: Algorithm for constructing the DenseZDD from a source ZDD.

5.4.2 Primitive ZDD operations

We show how to implement primitive ZDD operations on DenseZDD $DZ = \langle U, M, I\rangle$ except *getnode*. We give an algorithm for *getnode* in Section 5.5.

In the zero-edge tree, there are two types of nodes: real nodes and dummy nodes. Real nodes are those in the ZDD, while dummy nodes have no corresponding ZDD nodes. Real nodes are numbered from 1 to m based on preorders in the tree. Below a node is identified with this number, which we call its *node number*. We can convert between the node number i of a node and the position p in the BP sequence U by $p := select_1(M, i)$ and $i := rank_1(M, p)$. The 0-terminal has node number 0 and nonterminal nodes have positive node numbers. If a node number of a negative value is used, it means a node with an $\emptyset$-flag.

In addition, we consider an additional primitive operation for DenseZDDs: $chkdum(p)$. This operation checks if a node at position p on U is a dummy node or not. If it is a dummy *chkdum* returns false; otherwise it returns true. This operation is implemented by simply looking at the p-th bit of M. If the bit is 0, then the node is dummy; otherwise

it is a real node.

$index(i)$: Since the item of the node is the same as the depth of the node, we can obtain $index(i) := depth(U, select_1(M, i))$.

$one(i)$: $one(i)$: Because 1-children are stored in preorder of the parents of nodes, we can obtain $one(i) := I[i]$.

$topset(i, d)$: The node $topset(i, d)$ is the ancestor of node i in the zero-edge tree with index d. A naive solution is to iteratively climb up the zero-edge tree from node i until we reach a node with index d. However, as shown above, the index of a node is identical to its depth. By using the power of the succinct tree data structure, we can directly find the answer by $topset(i, d) := rank_1(M, level\_ancestor(U, select_1(M, i), d))$.

$zero(i)$: Implementing the *zero* operation requires a more complicated technique. Consider a subtree T of the zero-edge tree consisting of the node i, its real parent node r, all real children of r, and dummy nodes between those nodes. As a pre-condition, the zero-edge tree is constructed by Algorithm 5.6 in the appendix. That is, for the children of r, the nodes with higher *index* value have smaller preorder, and the imaginary parents of the children are dummy nodes (or i) that are added on the edge between r and the child having the highest *index* value. Computing $zero(i)$ is equivalent to finding r. Because the children of r are ordered from left to right in descending order of their depths, and dummy nodes are shared as much as possible, the deepest node in T is on the leftmost path from r. Furthermore, the parents of other real children are also on the leftmost path. This property also holds in the original zero-edge tree. The dummy node vector B_D stores flags in the preorder in the zero-edge tree. Then $B_D[p_r] = B_D[p_i] = 1$, where p_r and p_i are positions of nodes r and i in the BP sequence U, and $B_D[j] = 0$ for any $p_r < j < p_i$. Therefore we can find p_r by a rank operation. In summary, $zero(i) := rank_1(M, parent(U, select_1(M, i)))$.

5.4.3 Compressing the balanced parentheses sequence

The balanced parentheses sequence U is of length $2u$, where u is the number of nodes including dummy nodes. Let a ZDD have m real nodes and the number of items be n, u

is mn in the worst case. Here we compress the BP sequence U.

The BP sequence U consists of at most $2m$ runs of identical symbols. To see this, consider the substring of U between the positions for two real nodes. There is a run '))))...' followed by a run '((((...' in the substring. We encode lengths of those runs using some integer encoding scheme such as the delta-code or the gamma-code [18]. An integer $x > 0$ is encoded in $\mathcal{O}(\log x)$ bits. Because the maximum length of a run is n, U can be encoded in $\mathcal{O}(m \log n)$ bits. The range min-max tree of U has $2m/\log m$ leaves. Each leaf corresponds to a substring of U that contains $\log m$ runs. Then any tree operation can be done in $\mathcal{O}(\log m)$ time. The range min-max tree is stored in $\mathcal{O}(m(\log n + \log m)/\log m)$ bits.

We also compress the dummy node vector B_D. Because its length is $2u \leq 2mn$ and there are only m ones, it can be compressed in $m(2 + \log m) + o(u)$ bits by FID. The operations $select_1$ and $rank_1$ take constant time. We can reduce the term $o(u)$ to $o(m)$ by using a sparse array [48]. The operation $select_1$ is done in constant time, while $rank_1$ takes $\mathcal{O}(\log m)$ time. From the discussions above, we can prove Theorem 17 and Theorem 18. For the proof, see the appendix.

5.5 Hybrid Method

In this section, we show how to implement dynamic operations on DenseZDD. Namely, we need to implement the $getnode(i, v_0, v_1)$ operation. Our approach is to use a hybrid data structure using both the DenseZDD and a conventional dynamic ZDD. Assume that initially all the nodes are in a DenseZDD. Let m_0 be the number of initial nodes. In a dynamic ZDD, the operation $getnode(i, v_0, v_1)$ is implemented by a hash table indexed with the triple $\langle i, v_0, v_1 \rangle$.

We show first how to check whether the node $v := getnode(i, v_0, v_1)$ already exists. That is, we want to find a node v such that $index(v) = i$, $zero(v) = v_0$, $one(v) = v_1$. If v does not exist, we create such a node using the hash table as well as a dynamic ZDD. If it exists, in the zero-edge tree, v is a real child node of v_0. Consider again the subtree of the zero-edge tree rooted at v_0 and having all real children of v_0. All children of v_0 with index i share the common (possible dummy) parent node, say w. Because w is on the leftmost path in the subtree, it is easy to find it. Namely, $w :=$

Table 5.1: Properties of original ZDDs

data set	#items	#nodes	#itemsets
		Z	
grid5	40	584	8,512
grid10	180	377,107	4.1×10^{20}
grid15	420	1.5×10^{8}	2.3×10^{48}
webview100	952	2,299	11,928
webview50	1,617	6,060	70,713
webview20	2,454	30,413	634,065
webview10	2,905	93,900	4.4×10^{6}
webview5	3,149	353,092	2.7×10^{7}
webview0	3,149	465,449	3.2×10^{7}
randjoin128	32,696	6,751	2.5×10^{8}
randjoin2048	32,768	377,492	1.8×10^{13}
randjoin8192	32,768	1.3×10^{6}	3.7×10^{15}
randjoin16384	32,768	1.9×10^{6}	2.8×10^{16}

$level\_ancestor(U, select_1(M, rank_1(M, v_0)+1), i)$. The node v is a child of w with $one(v) = v_1$. Because all children of w are sorted in the order of *one* values by the construction algorithms, we can find v by a binary search. For this, we use *degree* and *child* operations on the zero-edge tree.

Theorem 19 *The existence of getnode(i, v_0, v_1) can be checked in $\mathcal{O}(t \log m)$ time, where t is the time complexity of primitive ZDD operations.*

If the BP sequence is not compressed, *getnode* takes $\mathcal{O}(\log m)$ time. Otherwise it takes $\mathcal{O}(\log^2 m)$ time. We should check the hash table before checking the zero-edge tree if dynamic nodes are already exist. As well as a conventional ZDD, hashing increases constant fanctors of time bounds significantly and add space bound $\mathcal{O}(x \log x)$ where x is the number of dynamic nodes.

5.6 Experimental Results

We ran experiments to evaluate the compression, construction, and operation times of DenseZDDs. We implemented the algorithms described in Sec. 5.3 and 5.4 in C/C++

Table 5.2: Comparison of performance, where δ denotes the dummy node ratio

data set	size (bytes)			comp. ratio		δ
	Z	DZ	DZ_c	DZ	DZ_c	
grid5	17,520	2,350	2,196	0.134	0.126	0.28
grid10	11,313,210	1,347,941	1,265,773	0.119	0.112	0.20
grid15	4,342,789,110	678,164,945	647,843,001	0.156	0.149	0.19
webview100	68,970	11,455	7,967	0.166	0.115	0.75
webview50	181,800	44,846	24,596	0.246	0.135	0.88
webview20	912,390	290,661	140,873	0.318	0.154	0.92
webview10	2,817,000	1,034,471	477,299	0.367	0.169	0.93
webview5	10,592,760	3,871,679	1,851,889	0.365	0.174	0.93
webview0	13,963,470	4,964,303	2,413,625	0.355	0.172	0.92
randjoin128	202,530	408,149	99,117	2.015	0.489	0.99
randjoin2048	11,324,760	2,415,648	1,511,658	0.213	0.133	0.82
randjoin8192	38,094,930	5,328,502	4,386,452	0.139	0.115	0.42
randjoin16384	56,447,280	7,056,418	6,113,910	0.125	0.108	0.14

languages on top of the SAPPORO BDD package [40]. The package is general implimentation of ZDD with 0-element edges, and uses 30 bytes per ZDD node. We performed experiments on eight quad-core 3.09 GHz AMD Opteron 8393 SE processors (i.e, 32 CPU cores in total) and 512 GB DDR2 memory shared among cores running. SUSE 10 Our algorithms use a single core since they are not parallelized.

We show the characteristics of the ZDDs in Table 5.1. Original ZDDs are denoted by Z. DenseZDDs without/with compression of the balanced parentheses sequences of the zero-edge trees are denoted by DZ/DZ_c , respectively.

As real data sets, for $N = 0, 5, 10, 20, 50, 100$, the source ZDD *webview*N was constructed from the data set BMS-Web-View-2[1] by using mining algorithm LCM over ZDD [44] with minimum support N. For artificial data sets, the ZDD *grid*N represents all self-avoiding paths on an $N \times N$ grid graph from the top left corner to the bottom right corner [29][2]. Finally, *randjoin*N is a ZDD that represents the join $C_1 \sqcup \cdots \sqcup C_4$ of four ZDDs for random families $C_1, \ldots, C_4$ consisting of N sets of size one drawn from the set of $n = 32768$ items.

In Table 5.2 we show the sizes of the original ZDD, the DenseZDD with/without

[1]http://fimi.ua.ac.be

[2]An algorithm animation: http://www.youtube.com/watch?v=Q4gTV4r0zRs

Table 5.3: Converting time

data set	conversion time (sec)			
	read	convert	const.	comp.
grid5	0.001	0.001	0.009	0.000
grid10	0.461	0.634	0.449	0.060
grid15	124.887	407.502	112.379	8.186
webview100	0.004	0.002	0.017	0.001
webview50	0.013	0.019	0.050	0.004
webview20	0.066	0.036	0.313	0.022
webview10	0.217	0.226	0.564	0.041
webview5	0.256	0.690	1.361	0.055
webview0	0.551	0.927	1.644	0.108
randjoin128	0.004	0.053	0.149	0.008
randjoin2048	0.243	0.742	0.946	0.029
randjoin8192	0.858	2.573	1.259	0.043
randjoin16384	1.270	5.016	1.471	0.070

compression and their compression ratio. We compressed FID for dummy node vector if the dummy node ratio is more than 75%. In almost cases, we observe that the DenseZDD is from 2.5 to 9 times smaller than the original ZDD, and that compressed DenseZDD is from 6 to 9 times smaller than the original ZDD. The compressed DenseZDD is quarter the size of the DenseZDD in the best case, and is half the size of the original ZDD in the worst case. For most of our data sets, the ratio δ of the number of dummy nodes to the size of DenseZDD is roughly 90%, except for *gridN* and *randjoin16384*.

In Table 5.3 and 5.4, we show the conversion times from ZDDs to DenseZDDs, traversal times, and search times on ZDDs and DenseZDDs. Conversion time is composed of four parts: time to read a file containing a stored ZDD and reconstruct the ZDD, convert it to raw parentheses, bits, and integers, construct succinct representation of them, and compress the BP of the zero-edge tree. The conversion time appears almost linear in the input size showing its scalability for large data. Traverse operation used $zero(v)$ and $one(v)$, while membership operation used $topset(v, i)$ and $one(v)$. We observed that the DenseZDD has almost twice longer traverse time and more than 10 times shorter search time than an original ZDD. These results show the efficiency of our implementation of the $topset(v, i)$ operation on DenseZDD using level-ancestor operations.

From the above results, we conclude that DenseZDDs are more compact than ordinary

Table 5.4: Traversing time and random searching time

data set	traverse time (sec)			search time (sec)		
	Z	DZ	DZ_c	Z	DZ	DZ_c
grid5	0.000	0.001	0.006	0.029	0.038	0.229
grid10	0.075	0.247	1.388	0.005	0.013	0.056
grid15	41.214	102.673	398.397	0.006	0.011	0.064
webview100	0.000	0.004	0.007	0.777	0.129	0.376
webview50	0.002	0.020	0.005	1.314	0.273	0.102
webview20	0.005	0.014	0.027	1.875	0.046	0.101
webview10	0.017	0.042	0.073	1.901	0.043	0.100
webview5	0.066	0.154	0.250	1.966	0.045	0.099
webview0	0.091	0.207	0.346	1.706	0.049	0.105
randjoin128	0.001	0.002	0.003	0.527	0.044	0.095
randjoin2048	0.093	0.126	0.145	8.071	0.044	0.098
randjoin8192	0.338	0.240	0.304	15.604	0.039	0.092
randjoin16384	0.676	0.353	0.447	19.501	0.040	0.093

ZDDs unless the dummy node ratio is extremely high, and the membership operations for DenseZDDs are much faster if the number of items is large or the dummy node ratio is small. We observed that in DenseZDDs, traversal time is approximately double and search time approximately one-tenth compared to the original ZDDs. The traversal is accelerated especially for large-scale sparse datasets because the number of nodes connected by 0-edges grows as large as the index number n. Recently, processing of "Big Data" have attracted a great deal of attention, and we often deal with a large-scale sparse dataset, which has more than ten thousands of items as the columns of a dataset. In the era of Big Data, we expect that DenseZDD will be effective for various real-life applications, such as data mining, system diagnosis, and network analysis.

5.7 Conclusions

In this chapter, we have presented a compressed index for static ZDDs named DenseZDD. We also proposed a hybrid method for dynamic operations on DenseZDD so that we can manipulate DenseZDD and conventional ZDD together. For future work, the one-child array should be stored in more compact space. Constructing the DenseZDD from the normal ZDD using external memory is an important open problem. We will implement the hybrid method on our ZDD package and convert/update algorithm with less memory.

We expect that our technique can be extended to other variants of BDDs. Note that the method presented in this chapter is possible to be transfered to sequence BDDs.

Chapter 6

Conclusions

In this thesis, we focused on decision diagrams that was initiated by Bryant [6]. The first decision diagram is binary decision diagram to represent Boolean functions. And, Minato introduced zero-suppressed binary decision diagrams for manipulating families of sets. Based on zero-suppressed binary decision diagrams, Loekito et al. proposed sequence binary decision diagrams for sets of strings. One of our goals was to solve problems on sets and strings using decision diagrams. Our results imply that sequence BDDs are suitable data structures to implement complete inverted files for directed acyclic graphs. In addition, we propose the static version of zero-suppressed BDD that allows faster membership operation than existing one on very sparse families of sets with a large number of items. Our main results are as follows. First, in Chapter 3, we consider the class of sequence binary decision diagrams (SeqBDDs) proposed by Loekito *et al.* [31], 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 shared sequence BDD environment. We also showed experimental results which shows the efficiency of the sequence BDDs and proposed algorithms for wide range of data. On Boolean set operations, we showed that the $\mathsf{Meld}_\diamond$ 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 $\mathsf{Meld}_\diamond$ has input linear time complexity. 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 [20], on sequence BDDs. Next, in Chapter 4, we propose new complete inverted files called SeqBDD-FPs. We also present algorithms to construct

them and to retrieve occurrence information from them. Computational experiments are executed to show the efficiency of SeqBDD-FPs. As future works, extensions of this approach to other suffix indexes would be interesting. Also, it would be an interesting future problem to study lowerbounds of the worst case time complexity of construction of sparse suffix trees with an arbitrary index set of size k when $O(k)$ space is allowed. Finally, in Chapter 5, we present a succinct data structures of Zero-suppressed Binary Decision Diagrams. The method also allows fast membership operation. We can obtain more compressed structure with Variable Length Array (VLA). Instead of arbitrary DFS order, we traverse 0-edge trees in the order that subtrees pointed by 1-edges much times visited first, and we use VLAs for 1-edge arrays. Then 1-edge array can be represented in more compact form. All levels of 1-edge arrays and rank dictionaries can be concatenated each other with a little modification. We expect that our technique can be extended to other variants of BDDs.

Now, we have obtained basic knowledges to further develop research of sequence BDD. Indices of DAG has many valuable applications such as genome matching, index for continuous data, and so on. The technique of DenseZDD can be applied sequence BDD in the same manner, and it is useful technique in real applications that hybrid method to handle dynamic and static decision diagrams at the same time. It is an open problem that the complexities of more complicated set operations on sequence BDDs. Besides, it is an open question whether there are algorithms to construct complete inverted files for directed acyclic graph in time linear to input or output. Another interesting question is whether each component of DenseZDD can be smaller. As a future problem, more efficient sequence BDD construction should be considered, especially for DAG indices. I believe that these decision diagrams will be helpful to solve the many problems in modern society.

Acknowledgments

First of all, I would like to thank my supervisor Professor Shin-ichi Minato, for his enthusiasm and for always finding time to discuss interesting angles and new insights, despite his busy schedule. I really appreciate his valuable comments, constant support and his approach to students in general. I also would like to thank Hiroki Arimura, the Professor of Laboratory of Information Knowledge Network and my former supervisor, providing me the subject to research and for his countless discussions and helpful comments on the implementation of SeqBDD. I would like to thank Thomas Zeugmann, Takuya Kida, Koji Tsuda, Charles Jordan, Takeaki Uno, Masayuki Takeda, Ayumi Shinohara, Shunsuke Inenaga, Rajeev Raman, Roberto Grossi, and Bruce W. Watson for their discussions and valuable comments.

I would like to express my deep gratitude to Ryo Yoshinaka for his useful comments on the structure of a SeqBDD compared to an ADFA his interest in the results of this thesis. My thanks go out to Kunihiko Sadakane for his great advise and many fruitful discussions. His suggestions were very inspiring for me. I also would like to thank our secretaries Yu Manabe, Yukie Watanabe, and Sachiko Soma for them invaluable support.

I would also like to thank Hiroshi Aoki, Yuma Inoue, Hirohito Sasakawa, Kunihiro Wasa and all the other students in our laboratories for creating a great working environment. I have been fortunate to have had as my senior colleague in ERATO Minato Discrete Structure Manipulation Project: Yasuyuki Shirai, Takeru Inoue, Norihito Yasuda, Hiroaki Iwashita, Yukinobu Hamuro, Yasuo Tabei, Yoshiki Saitoh, Jun Kawahara, Takahisa Toda, Shogo Takeuchi, Ryutaro Kurai, and all the other members in the project. It has been my good fortune to enjoy the company, support, and kindness of colleagues in the laboratory. Many friends have greatly helped me. Finally, I would like to thank my parents and brother for their unwavering support which significantly helped me to finish this thesis.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Hiroshi Aoki, Shigeru Yamashita, and Shin ichi Minato. An efficient algorithm for constructing a sequence binary decision diagram representing a set of reversed sequences. In *Proceedings of 2011 IEEE International Confenrece on Granular Computing*. IEEE, 2011.

[3] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.

[4] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Journal of Theoretical Computer Science*, 40(0):31–55, 1985.

[5] Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[6] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[7] Johannes Bubenzer. Minimization of acyclic DFAs. In *Proceedings of the Prague Stringology Conference 2011 (PSC 2011)*, pages 132–146. Czech Technical University in Prague, 2011.

[8] Maxime Crochemore. Transducers and repetitions. *Journal of Theoretical Computer Science*, 45(1):63–86, 1986.

[9] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[10] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002.

[11] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000.

[12] Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato. Substring indices based on sequence BDDs. Technical report, TCS Technical Report Series A, TCS-TR-A-10-40, Hokkaido University, 2010.

[13] Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato. Implementation of sequence bdds in erlang. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang 2011, pages 90–91. ACM, 2011.

[14] Shuhei Denzumi, Hiroki Arimura, and Kunihiko Sadakane. Manipulatable compressed string indexing technology for pattern matching. *Institute of Electronics, Information and Communication Engineers*, 97(12):1080–1085, 2014. (in Japanese).

[15] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, and Shin ichi Minato. Sequence binary decision diagram: Minimization, relationship to acyclic automata, and complexities of boolean set operations. *Discrete Applied Mathematics*, (0), 2014. (in print).

[16] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, and Shin-ichi 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, Hokkaido University, 2011.

[17] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, and Shin-ichi Minato. Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations. In *Proceedings of the Prague Stringology Conference 2011 (PSC 2011)*, pages 147–161. Czech Technical University in Prague, 2011.

[18] Peter Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[19] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Journal of Software: Practice and Experience*, 33(11):1035–1049, 2003.

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

[21] Esben Rune Hansen, S. Srinivasa Rao, and Peter Tiedemann. Compressing binary decision diagrams. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, pages 799–800. ACM, 2008.

[22] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and computation*, pages 189–196. Academic Press, 1971.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd. edition, 2006.

[24] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Construction of the CDAWG for a trie. In *Proceedings of the Prague Stringology Conference 2001 (PSC 2001)*, pages 37–48. Czech Technical University, 2001.

[25] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Science*, 38(1):38–86, 1989.

[26] Haim Kaplan. Persistent data structures. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook on Data Structures and Applications*, chapter 31. CRC Press, 2005.

[27] Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.

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

[29] Donald E. Knuth. *The Art of Computer Programming, volume 4A, Combinatorial Algorithms, Part 1.* Addison-Wesley, 1st edition, 2011.

[30] Jim Larson. Erlang for concurrent programming. *Communications of the ACM*, 52(3):48–56, 2009.

[31] Elsa Loekito, James Bailey, and Jian Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowledge and Information Systems*, 24(2):235–268, 2009.

[32] Cláudio L. Lucchesi and Tomasz Kowaltowski. Applications of finite automata representing large vocabularies. *Journal of Software: Practice and Experience*, 23(1):15–30, 1993.

[33] Cláudio L. Lucchesi, Tomasz Kowaltowski, and Jorge Stolfi. Minimization of binary finite automata. *Journal of the Brazilian Computer Society*, 1(3):5–11, 1995.

[34] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[35] Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, 18(0), 2013.

[36] Pere Mateu-Villarroya and Joseph Prades-Nebot. Lossless image compression using ordered binary-decision diagrams. *Electronics Letters*, 37(3):162–163, 2001.

[37] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[38] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceeding of Design Automation Conference (DAC 1993)*, pages 272–277. IEEE, 1993.

[39] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[40] Shin-ichi Minato. SAPPORO BDD package. Hokkaido University, 2011. unreleased.

[41] Shin-ichi Minato. Z-skip-links for fast traversal of zdds representing large-scale sparse datasets. In *Proceedings of the 21st Annual European Symposium (ESA 2013)*, pages 731–742. LNCS 8125, Springer, 2013.

[42] Shin-ichi Minato and Hiroki 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 (KDID 2006)*, pages 152–169. LNCS 4747, Springer, 2006.

[43] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th Design Automation Conference (DAC 1990)*, pages 52–57. IEEE, 1990.

[44] Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In *Proceedings of the 12th Advances in Knowledge Discovery and Data Mining (PAKDD 2008)*, pages 234–246. LCNS 5012, Springer, 2008.

[45] Mehryar Mohri, Pedro Moreno, and Eugene Weinstein. Factor automata of automata and applications. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA 2007)*, pages 168–179. LNCS 4783, Springer, 2007.

[46] Mehryar Mohri, Michael Riley, and Richard Sproat. Algorithms for speech recognition and language processing. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL 1996)*, pages 231–238. MIT Press, 1996.

[47] Gonzalo Navarro and Kunihiko Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, 2014.

[48] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70. SIAM, 2007.

[49] Dominique Perrin. Finite automata. In Jan van Leuwen, editor, *Handbook of Theoretical Computer Science,*, volume B. Formal Models and Semantics, pages 1–57. Elsevier, 1990.

[50] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable

dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43:1–43:25, 2007.

[51] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.

[52] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In *Proceedings of 11th Workshop on Algorithms in Bioinformatics (WABI 2011)*, pages 270–281. LNCS 6833, Springer, 2011.

[53] Mike Starkey and Randy Bryant. Using ordered binary-decision diagrams for compressing images and image sequences. Technical Report CMU-CS-95-105, Carnegie Mellon University, 1995.

[54] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical methods for constructing suffix trees. *VLDB Journal*, 14(3):281–299, 2005.

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

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

[57] Ryo Yoshinaka, Jun Kawahara, Shuhei Denzumi, Hiroki Arimura, and Shin-ichi 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, Hokkaido University, 2011.