On Representing the Set of All Parse Trees with a Decision Diagram

Kei Amii  
Kyoto University  
k.amii@iip.ist.i.kyoto-u.ac.jp

Masaaki Nishino  
NTT Communication Science Laboratories  
nishino.masaaki@lab.ntt.co.jp

Akihiro Yamamoto  
Kyoto University  
akihiro@i.kyoto-u.ac.jp

keywords: decision diagram, ZSDD, context-free grammar

Summary

In this paper, we construct decision diagrams (DDs) representing the set of all parse trees of a context-free grammar (CFG) from a sequence data and analyze DD size. CFG is widely used in the field of natural language processing and bioinformatics to estimate the hidden structures of sequence data. A decision diagram is a data structure that represents a Boolean function in a concise form. By using DDs to represent the set of all parse trees, we can efficiently perform many useful operations over the parse trees, such as finding parse trees that satisfy additional constraints and finding the most probable parse tree. Since the time complexity of these operations strongly depends on DD size, selecting an appropriate DD variant is important. Experiments on the parse trees of a simple CFG show that the Zero-suppressed Sentential Decision Diagram (ZSDD) is better than other DDs; we also give theoretical upper bounds on ZSDD size of a simple CFG. Moreover, we propose an efficient method based on CYK (Cocke-Younger-Kasami) algorithm to construct ZSDDs that can represent the set of all parse trees. Experiments show that the method can construct ZSDDs much faster than the naive method based on compiling a Boolean function.

1. Introduction

Context-free grammar (CFG) is a model often used to parse sequence data. It is widely used in the fields of natural language processing [Manning 99] and bioinformatics [Durbin 98]. The CYK (Cocke-Younger-Kasami) algorithm is a well-known technique for determining how a given sequence data can be generated from a given CFG, that is, obtaining parse trees. Unfortunately, the CYK algorithm cannot be used if some additional constraints are imposed on the form of parse trees, e.g., constraints on the number of times a rule can be used, or prohibition of the use of a particular pair of rules at the same time. Such additional constraints enable sequences to be analyzed with consideration of background knowledge.

In this paper, we introduce a method based on decision diagrams to derive parse trees that satisfy given constraints. Decision diagrams (DDs) are data structures that compactly represent Boolean functions or families of sets as directed acyclic graphs. A DD can support various operations and transformations on Boolean functions in time polynomial to DD size. Therefore, by using the operations of DDs, we can obtain parse trees satisfying constraints and find the most probable parse tree of probabilistic CFG. By using DDs, we can exploit any constraints that can be represented as a Boolean function. Typical constraints are as follows:

- Specifying the partial structure of parse trees.
- Restricting the number of times a rule is used in a parse tree
- Prohibiting the use of particular pair of rules used in a parse tree.

These constraints can be used in structured prediction tasks including syntactic parsing and sequence tagging to improve performance [Yoshikawa 09, Chang 12, Martins 11].

Since the efficiency strongly depends on DD size, it is important to utilize a succinct DD variant. Examples of such variants are Binary Decision Diagram (BDD) [Bryant 86], Zero-suppressed Binary Decision Diagram (ZDD) [Minato 93], Sentential Decision Diagram (SDD) [Darwiche 11a], and Zero-suppressed Sentential Decision Diagram.
(ZSDD) [Nishino 16]. Given a Boolean function or a family of sets, which DD variant is the smallest in representing it depends on the nature of the given function or family. Moreover, each DD variant has configurable parameters that influence DD size, namely variable ordering and vtrees. Thus, in order to handle parse trees efficiently, we need (1) to choose the most appropriate DD, and (2) to optimize its parameters. This paper compares the sizes of the four kinds of DDs discussed above assuming some orders of variables and some vtrees. Our experiments show that ZSDDs have smaller size than the others. We also give the theoretical upper bound of ZSDDs and the vtrees that yield the smallest size.

The time to construct a DD is also important when analyzing sequence data with DDs. In this research, we propose two construction methods: a naive method based on compiling a Boolean function into a DD and a method based on the CYK algorithm. We compare the time taken by the two methods to construct ZSDDs and show that the CYK-based method is much faster than the naive method.

In Chapter 2, we introduce related work. We review context-free grammars and decision diagrams as technical preliminaries in Chapter 3. Next, Chapter 4 introduces our method of constructing DDs to represent the set of all parse trees of a context-free grammar. We describe our experiment on four DDs, and show that ZSDD has the smallest size when representing the set of all parse trees of a simple context-free grammar in Chapter 5. ZSDD size is theoretically analyzed in Chapter 6, and we provide a summary and future work in Chapter 7.

2. Related Work

Syntactic parsing is one of the most important examples of structured prediction tasks, i.e., tasks of predicting inherent structure from its appearance. In NLP field, there are several important structured prediction tasks including syntactic parsing, sequence tagging and information extraction. Some previous work exploits background knowledge as constraints to improve performance on several structured prediction tasks [Chang 12, Martins 11, Yoshikawa 09]. Compared with previous approaches, our DD-based method is flexible since it can impose a wide range of constraints and can perform several types of inferences.

Operations and queries that DDs can support are investigated in [Broeck 15, Darwiche 11b, Minato 13]. It has been shown that many supported operations and queries take time that is polynomial to DD size. Properties on DD size have also been investigated and some significant theories have been proven. DD size strongly depends on variable ordering or vtree, but finding the optimal variable ordering for a BDD has been shown to be NP-hard [Tani 96]. It has also been shown that BDDs tend to be small if the variable ordering places variables related to each other close [Fujita 88]. Some methods to find better variable orders have been proposed [Drechsler 00, Fujita 91]. The question of which DD variant has smaller size is also important and has been investigated. ZDDs or ZSDDs tend to be smaller than BDDs or SDDs, respectively, when representing Boolean functions or families whose number of models, variable assignments that makes the function true, is much smaller than the number of all possible variable assignments [Minato 93, Nishino 16].

It has been theoretically proven that there is a class of Boolean functions (families of sets) whose members can be represented by an SDD of polynomial size but that have exponential size if BDD representation is used [Bova 16]. The sizes of BDDs that represent some functions have been investigated. The hidden weighted bit function is considered to be the simplest function with exponential BDD size [Bollig 91]. The integer multiplication function is also considered to be important, and its upper and lower bound is given in [Amano 07, Bryant 91, Woelfel 05].

The time to construct a DD is also important when solving actual problems. In some situations, constructing DDs by a naive method of repeatedly applying binary logical operations may take too long. Efficient methods for constructing DDs that represent a specific function have been proposed such as compiling a Boolean function in conjunctive normal form into SDD [Oztok 15]. An important problem with DDs is how to represent specific graph substructures, because DDs that can represent graph substructures have many potential applications. Knuth proposed a top-down algorithm to construct a BDD or a ZDD to represent a set of all connected components (numbering more than $10^{10}$), and showed that the resulting BDD or ZDD has size of the order of hundreds [Knuth 11]. An efficient algorithm to construct ZSDDs that can represent the set of specific graph substructures was proposed in [Nishino 17]. These algorithms to construct DDs that represent graph substructures run much faster than naive methods that compile a Boolean function that corresponds to a graph into a DD. Our method can construct a DD that represents parse trees, but it is different from previous top-down algorithms.

With regard to parsing sequences, some studies use probabilistic CFGs to treat not only the most probable parse tree but multiple parse trees. It was shown that the set of the best $k$ parse trees can be enumerated based on the
the former probabilistic inference task can be formulated as
\[
P(c) = \sum_{T \in \mathcal{T}_f(F)} P(T), \quad P(T) = \prod_{r_k \in T} q_k,
\]
and the later is formulated as the problem of finding the most probable parse tree \(T^*\) that satisfies
\[
T^* = \arg \max_{T \in \mathcal{T}_f(F)} P(T).
\]

For example, given CFG \(G\), a sentence \(c = (c_1, \ldots, c_N)\), and constraint over parse tree \(F: \mathcal{T}_f \to [t, \infty]\), let \(\mathcal{T}_f(F) := \{T \in \mathcal{T}_f | F(T) = \infty\}\). Then the problem is to implicitly obtain \(\mathcal{T}_f(F)\) and/or obtain \(T^*\) given \(\mathcal{T}_f(F)\).

### 3.2 \((X, Y)\)-Decomposition

Due to space limits, we focus on ZSDD among the variants of DD. Although ZSDD was invented to represent families of sets, it can also represent Boolean functions because a family of sets is equivalent to a Boolean function. For example, family of sets \(\{\{a\}, \{b, c\}\}\) whose universe is \(\{a, b, c\}\) corresponds to Boolean function \(a \land \neg b \land \neg c\) \(\lor (\neg a \land b \land c)\). For simplicity we call \(a, b, c, d\) Boolean variables even if they appear in a family of sets due to the equivalency of family of sets and Boolean functions.

\((X, Y)\)-decomposition is a method that divides a given family of sets into smaller subsets, a significant step in composing ZSDDs. Let \(f\) be a family of sets, and \(X, Y\) be groups of variables that yield a partition of the variables of \(f\). It follows that the function can be decomposed as
\[
f(X, Y) = [p_1(X) \cup s_1(Y)] \cup \cdots \cup [p_n(X) \cup s_n(Y)],
\]
where \(p_i(X), s_j(Y)\) are subfamilies whose variables are \(X\) and \(Y\), respectively. Operations \(\cup\) and \(\sqcup\) are union and join, respectively. They are defined as \(f \sqcup g = \{a \mid a \in f\) and \(a \in g\)\) and \(f \cup g = \{a \cup b \mid a \in f\) and \(b \in g\)\). We call \(p_1, \ldots, p_n\) primes and \(s_1, \ldots, s_n\) subs. We assume \(p_1, \ldots, p_n\) are not empty and they form a partition of the power set of the universe. That is, they satisfy \(p_i \cap p_j = \emptyset\) for all \(i \neq j\) and \(\bigcup_{i=1}^n p_i\) equals the power set. If \(p_i = \emptyset\) for all \(i\), we say the decomposition is an \((X, Y)\)-partition, and denote it as \((p_1, s_1), \ldots, (p_n, s_n)\). Moreover, if \(s_i \neq s_j\) for all \(i \neq j\) is satisfied, we say the \((X, Y)\)-partition is compressed. For example, given \(f = \{\{a, b\}, \{b\}, \{b, c\}, \{c, d\}\}\), \(X = \{a, b\}\), and \(Y = \{c, d\}\), the compressed \((X, Y)\)-partition is
\[
[\{\{a, b\}\} \cup \emptyset] \cup [\{b\} \cup \{0, \{c\}\}] \cup [\emptyset \cup \{c, d\}],
\]
where \(a, b, c, d\) are Boolean variables.
3.3 vtree

We introduce vtree, another significant concept in the ZSDD composition. A vtree represents a family of sets as a directed acyclic graph by applying the \((X, Y)\)-partition recursively. That is, ZSDD divides a given family of sets into \(p_1, \ldots, p_n\) and \(s_1, \ldots, s_n\) by applying the \((X, Y)\)-partition. A ZSDD represents a family of sets by recursively applying \((X, Y)\)-partitions, where partition order is determined by the vtree. We can make a unique ZSDD given a family of sets and a vtree. A vtree is a binary tree whose leaves correspond to variables. We show a vtree example in Figure 1(a).

A vtree node represents the partition of variables into two groups: variables that appear in the left subtree and those that appear in the right subtree. In this figure, root node \(v = 3\) represents \((X, Y)\)-partition where \(X = \{a, b\}\) and \(Y = \{c, d\}\). Similarly, node \(v = 1\) represents a partition where \(X = \{b\}\) and \(Y = \{a\}\). In this way, every non-leaf vtree node represents a partitioning. We use \(v^L\), \(v^R\) to represent the left and the right child vtree nodes of \(v\), respectively. We say a vtree is right-linear if each left-child of an internal node is a leaf. To avoid confusion, we use the term vnode to refer to nodes in the vtree and denote them as \(v, v^L, v^R, v_1, v_2, \ldots\).

### 3.4 Zero-suppressed Sentential Decision Diagram (ZSDD)

We recursively define the Zero-suppressed Sentential Decision Diagram (ZSDD) as follows. Let \(\alpha\) be a ZSDD and \(\langle \alpha \rangle\) be the family of sets that \(\alpha\) represents.

**Definition 3.4.** \(\alpha\) is a ZSDD that respects vtree node \(v\) if:
- \(\alpha = \varepsilon\) or \(\alpha = \bot\).
- \(\alpha = X\) or \(\alpha = \pm X\) and \(v\) is a leaf with variable \(X\).
- \(\alpha = (\{X\})\) and \((\pm X) = (\{X\}, \emptyset)\)
- \(\alpha = \{(\beta_1, \gamma_1), \ldots, (\beta_n, \gamma_n)\}\), \(v\) is internal, where \(p_i = \langle \beta_i \rangle\) and \(s_i = \langle \gamma_i \rangle\), \((\beta_1, \ldots, \beta_n)\) are ZSDDs that respect the subtrees of \(v^L\), \((\gamma_1, \ldots, \gamma_n)\) are ZSDDs that respect the subtrees of \(v^R\), and \(p_1, \ldots, p_n\) is a partition.

\(\varepsilon, \bot, X, \pm X\) are called terminal ZSDDs. Other ZSDDs represent \((X, Y)\)-partition \(\{(p_1, s_1), \ldots, (p_n, s_n)\}\) corresponding to vnode \(v\).

Figure 1(b) is an example of a ZSDD that represents the family of sets \(\{(a, b), \{b\}, \{b, c\}, \{c, d\}\}\) given the vtree in Figure 1(a). We represent \((X, Y)\)-partition as a circle node, and call it a decision node. A decision node has child nodes, and each child node is represented as a pair of boxes. The left box of a child node corresponds to prime \(p_i\), while the right box corresponds to sub \(s_i\). We call the ZSDD generated by \((X, Y)\)-partition a subfunction on \(X\). We define the size of a decision node as the number of pairs of primes and sub covered by the node, and the size of ZSDD as the sum of the sizes of all decision nodes.

Operation \(\text{apply}\) takes time polynomial to ZSDD size as it takes two ZSDDs \(\alpha, \beta\) and binary operation \(\circ\) as inputs, and returns a ZSDD representing \(\langle \alpha \circ \langle \beta \rangle \rangle\) [Nishino 16]. This allows us to use a ZSDD to analyze sequences with CFG. If we can construct ZSDD \(\alpha\) representing the sets of all valid parse trees and ZSDD \(\beta\) representing constraints, we can obtain ZSDD \(\langle \alpha \rangle \cap \langle \beta \rangle\) represents the set of parse trees that satisfy the constraints by using apply operation. In the following, we denote \(\alpha \circ \beta\) to represent performing \(\text{apply}\) operations between two ZSDDs \(\alpha, \beta\), where \(\circ\) is either of \(\sqcup\) (union), \(\cap\) (intersection), or \(\sqcap\) (join). Additionally, once such a ZSDD is constructed, we can solve the problem of (2) in time linear with the size of \(\tau^*\) by performing a dynamic programming algorithm [Darwiche 09].

### 4. Construction of ZSDDs

In this Chapter, we introduce two methods to construct ZSDDs that represent the set of all parse trees. We choose ZSDDs among the known DD variants because we find that a ZSDD has the smallest size when representing the set of all parse trees (See Chapter 5). All the DD variants we compare in this paper support important operations for parse trees, which includes finding a parse tree with the highest score or evaluating the sum of scores of possible parse trees, in time linear with the size of DDs. Here we assume the score functions are linear. In other words, we assume that each variable has a score, and the score of a set can be defined as the sum of score of variables included in the set. If we use the probability of rules as the score, we can efficiently perform probabilistic CFGs (PCFGs) inference.

The input of the methods is a sequence of terminal symbols \(c_1 \ldots c_N\) and CFG \(G = (V, \Sigma, P, S)\) in Chomsky normal form, where \(c_1, \ldots, c_N \in \Sigma\). Output is a ZSDD representing the set of all parse trees of \(c_1 \ldots c_N\), where the
variables of the ZSDD represent the kind of rules and the range. The first method is a naive one based on designing a Boolean function that represents the set of all parse trees and compiling the function. The other method is an efficient one using binary operation of ZSDDs based on CYK algorithm.

We use \( c_{ij} \) to represent the subsequence \( c_i \ldots c_j \). In order to represent the set of parse trees as a family of sets, we introduce Boolean variable \( b_{ijk} \) to represent the appearance of production rules in a parse tree, where \( 1 \leq i \leq j \leq N \) and \( 1 \leq k \leq |P| \). We let \( b_{ijk} = 1 \) when substring \( c_{ij} \) is generated by applying the \( k \)-th production rule \( r_k \). Each Boolean variable corresponds to a usage of a production rule, and the set of parse trees can be represented as a family of sets. An example of correspondence is illustrated in Figure 2.

\[
\begin{align*}
1. & \quad S \rightarrow AB \\
2. & \quad B \rightarrow BC \\
3. & \quad A \rightarrow a \\
4. & \quad B \rightarrow b \\
5. & \quad C \rightarrow c
\end{align*}
\]

An example of correspondence between a parse tree and a family of sets. Let \( V = \{S, A, B, C\} \). \( \Sigma = \{a, b, c\} \). \( P \) be the set of rules on the left, and the input sequence \( c_{1N} \) be \( abc \). A set \( \{b_{131}, b_{223}, b_{113}, b_{224}, b_{335}\} \) represents a valid parse tree. On the right parse tree, each variable in parenthesis correspond to an usage of each rule.

\[ S(b_{131}) \quad B(b_{223}) \quad A(b_{113}) \quad B(b_{224}) \quad C(b_{335}) \]

Fig. 2

4.1 Representation as a Boolean Function

We define the Boolean function that represents parse trees in this Chapter. A family of sets can be written as variable sets that make the Boolean function \( true \).

Let \( S^m_{ij} \) be the set of the first generation rules that is used to derive \( c_{ij} \) from \( X_n \). It follows that \( S^m_{i,1} \) is defined as

\[
S^m_{i,1} := \{k \in [K] \mid n_k = n\}. \quad (5)
\]

\( S^m_{ij} \) can be defined recursively as

\[
S^m_{ij} := \bigcup_{m=i}^{j-1} \{k \mid r_k = (X_n \rightarrow X_n', X_n''), \quad |S^{n'}_{im}| > 0, |S^{n''}_{m+1j}| > 0\}.
\]

(6)

where \( i < j \). If \( S^m_{iN} \neq \emptyset \), the given sequence \( \sigma \) can be generated and parse trees can be obtained by tracing indices in \( S^m_{ij} \).

The Boolean function represents constraints so that the set of variables taking \( 1 \) can construct valid parse trees, i.e. those that denote the structure correctly. Each subsequence, \( c_{ij} \), is generated by exactly one top production rule. Therefore, we need a constraint that makes multiple variables among \( b_{ij} \), not true. This is written as

\[
H_{ij} = \bigwedge_{1 \leq k < l \leq |P|} \neg b_{ijk} \lor \neg b_{ijl}.
\]

(7)

We define the following constraint to ensure one of the rules in \( S^n_{ij} \) is used:

\[
F^n_{ij} = \bigvee_{k \in S^n_{ij}} b_{ijk}.
\]

(8)

\( F^n_{ij} \) takes 1 if \( c_{ij} \) is generated from non-terminal \( X_n \). If \( r_k = (X_n \rightarrow X_n', X_n'') \) is used, the rules that correspond to \( n' \) and \( n'' \) must be used:

\[
D_{ijk} = \neg b_{ijk} \lor \bigvee_{m=i}^{j-1} F^n_{im} \land F^n_{m+1j}.
\]

(9)

Finally, a constraint on arbitrary pairs \( b_{ijv} \) and \( b_{klw} \) such that \( i < k < j < l \) is defined as

\[
C = \bigwedge_{i < k < l < j \leq t \leq l \leq \lfloor |P| \rfloor} \neg b_{ijv} \lor \neg b_{klw}.
\]

(10)

If both of \( b_{ijv} \) and \( b_{klw} \) take 1, \( c_{ij} \) is generated by using \( r_v \) and \( c_{kl} \) is generated by \( r_w \). As a consequence, the two ranges overlap and a valid parse tree cannot be constructed, so this constraint is necessary.

By using the constraints (7)-(10), we can represent a Boolean function that returns \( true \) when given a set of rules that yields correct generation as

\[
\left( \bigwedge_{1 \leq i < j \leq N} H_{ij} \right) \land \left( \bigwedge_{1 \leq i < j \leq N, 1 \leq k \leq |P|} D_{ijk} \right) \land C. \quad (11)
\]

In the naive method, we compile the Boolean function into a ZSDD. First of all, we construct a ZSDD corresponding to each literal, and then combine them by using apply operation along with \( \lor, \land \) operations in the Boolean function. We need \( O(|P|^2 N^4) \) operations to construct the ZSDD representing parse trees.

In the next section, we introduce an efficient construction method which needs \( O(|P|^2 N^4) \) apply operations. Since the time of one apply operation depends on the size of input ZSDD sizes, we cannot compare the complexity of these methods based on only the number of apply operations. Instead, we compare construction times by experiments.

4.2 The Efficient Method

We introduce a method that uses the CYK algorithm to construct the ZSDD. Let \( \alpha_{ij} \) be a ZSDD that represents the set of all parse trees of substring \( c_{ij} \) from non-terminal
symbol \( X_n \). Set families corresponding to ZSDD \( \alpha_{ijn} \) can be defined recursively as
\[
\langle \alpha_{ijn} \rangle = \bigcup_{r_k=(X_n \to c_i)} \{ \{ b_{ijk} \} \}. 
\tag{12}
\]
Otherwise,
\[
\langle \alpha_{ijn} \rangle = \bigcup_{r_k=(X_n \to X_m \cdot X_n')} j^{-1} (\langle \alpha_{irn} \rangle \sqcup \langle \alpha_{r+1jn'} \rangle) \sqcup \{ \{ b_{ijk} \} \}. 
\tag{13}
\]
Algorithm 1 shows an efficient ZSDD construction procedure that exploits the above recursive definition of the set of parse trees. ZSDD \( \alpha_{ijn} \), corresponding to terminal symbol \( c_i \), is given by lines 1-5. Here we use \( \beta_{ij} \) to be the ZSDD that satisfies \( \langle \beta_{ij} \rangle = \{ \{ b_{ijk} \} \} \). \( \beta_{ij} \) can be computed in a constant time [see [Nishino 16]]. If the \( k \)-th rule \( r_k \) is in the form \( X_n \to c_i \), \( \{ b_{ijk} \} \) is an element of \( \langle \alpha_{ijn} \rangle \).

To construct \( \alpha_{ijn} \) (\( i < j \)), lines 10-13, we combine two ZSDDs \( \alpha_{irn'} \) and \( \alpha_{r+1jn'} \) (\( i \leq r < j \)) based on the existence of rule \( r_k = X_n \to X_m \cdot X_n' \). In the ZSDD constructed by the Algorithm, each element represents one parse tree. \( \langle \alpha_{irn'} \rangle \sqcup \langle \alpha_{r+1jn'} \rangle \) represents all combinations of two sets of partial parse trees on \( c_1 \ldots c_r \) and \( c_r+1 \ldots c_j \), and \( \langle \alpha_{irn'} \rangle \sqcup \langle \alpha_{r+1jn'} \rangle \sqcup \{ \{ b_{ijk} \} \} \) is a family of sets representing parse trees on \( c_1 \ldots c_j \) whose start symbol is \( X_n \). By applying these operations, we can construct \( \alpha_{1NS} \).

In this method, the number of operations is \( O(|P|^2 N^3) \), which is smaller than the \( O(|P|^3) \) of the naive method. Since the time of each operation depends on the size of input ZSDDs, we measure the time taken in experiments.

5. Experiment

In this Chapter, we show results of two experiments. One finds which DD variants and which vtree are best at representing the set of parse trees. The other shows the efficiency of our construction methods.

We use BDD, SDD, ZDD, and ZSDD to represent the set of all parse trees of the context-free grammars described below and compare them in terms of size. The experimental environment is as follows.

- CPU: 3.4 GHz Intel Core i7
- Memory: 20 GB 1333 MHz DDR3
- OS: Mac OS X Yosemite

We use SDD library\(^1\) written in programming language C and ZSDD library\(^2\) written in C++. Since BDD is a strict subset of SDD, we use the SDD library when constructing BDD. Similarly, we use ZSDD library when constructing ZDD. We construct BDD/ZDD by giving a right-linear vtree to SDD/ZSDD.

The experiment uses 3 CFGs; \( G_1 \) and \( G_2 \) are simple and typical, and \( G_3 \) is used in the field of bioinformatics [Durbin 98].

\( G_1: A \to AA, \ A \to a \)
\( G_2: A \to AA, \ A \to AB, \ B \to BA, \ A \to a, \ B \to b \)
\( G_3: S \to aS \mid cS \mid gS \mid uS \mid Sa \mid Sc \mid Sg \mid Su \mid aS \mid uSa \mid cSg \mid gSc \mid SS \mid \epsilon \)

Let \( A, B, S \) be a non-terminal symbol, and \( a, b, c, g, u \) be a terminal symbol. We determine the DD size as a function of the length of sequence data \( n \). In grammar \( G_1 \), the number of terminal and non-terminal symbols is only one, so we use \( b_{ij} \) instead of \( b_{ijk} \). We parse sequences like \( aaaa \). In grammar \( G_2 \), the sequence is composed of a repetition of \( ab \) as in \( ababab \). In grammar \( G_3 \), we use two RNA sequences formatted into \( N = 8 \) and \( N = 20 \); original data taken from RNAcentral [Consortium 14].

The problem of finding optimal variable ordering for a BDD is NP-hard [Tani 96]. However, it is known BDDs tend to be small if the variable order places related variables close to each other [Fujita 88]. Thus, we examine the following 4 typical vtrees that place related variables

---

\(^1\) http://reasoning.cs.ucla.edu/sdd/

\(^2\) https://github.com/nonnmsk/zsdd

---

**Algorithm 1** ZSDD construction based on the CYK algorithm

**Input:** Sequence \( c_1 \ldots c_N \) and CFG \( G = (V, \Sigma, P, S) \) in Chomsky normal form.

**Output:** A ZSDD representing the set of all parse trees.

1. for \( i = 1 \) to \( N \) do
2. for \( X_n \in V \) do
3. \( \alpha_{ijn} \leftarrow \bot \)
4. for \( r_k = X \to c_i, r_k \in P \) do
5. \( \alpha_{ijn} \leftarrow \alpha_{ijn} \cup \beta_{ijk} \)
6. for \( t = 1 \) to \( N - 1 \) do
7. for \( i = 1 \) to \( N - t \) do
8. \( j \leftarrow i + t \)
9. for \( X_n \in V \) do
10. \( \alpha_{ijn} \leftarrow \bot \)
11. for \( r = i \) to \( i + j - 1 \) do
12. for \( r_k = X_n \to X_m \cdot X_n', r_k \in P \) do
13. \( \alpha_{ijn} \leftarrow \alpha_{ijn} \cup \beta_{ijk} \)
14. **Return** \( \alpha_{1NS} \).
close. We show examples of each in Figure 3; sequence length is 3. BDD and ZDD are limited to decomposition by a single variable, and are equivalent to SDD and ZSDD given right-linear vtrees.

**vtree1** A right-linear vtree decomposing in descending order of \( j - i \). If the values are equal, we use descending order of value \( i \). This corresponds to the serial variables from root to leaves in Figure 3(a).

**vtree2** A vtree combining right-linear vtrees composed of variables whose values, \( j - i \), are equal. This decomposes variables that have the same height, Figure 3(b), at once.

**vtree3** A vtree combining two right-linear vtrees. One corresponds to terminal symbols, and the other corresponds to non-terminal symbols. At root vnode \( v \), \( v^t \) corresponds to terminal symbols and \( v^r \) corresponds to non-terminal symbols, see Figure 3(c).

**vtree4** A vtree combining right-linear vtrees composed of the variables that have equal value, \( i \). This corresponds to making the right-linear vtree from left in Figure 3(d).

We gave vtree1 to BDD and ZDD, and vtrees 2,3,4 to SDD and ZSDD. We can construct vtrees on general CFG as follows. At first, we make right-linear vtrees composed of \( b_{ij1}, b_{ij2} \ldots \) like Figure 3 (a). Let this vtree called \( VT_{ij} \). And we replace \( b_{ij} \) in Figure 3 for \( VT_{ij} \).

### 5.1 The Size and the Construction Time

Table 1 shows the size of each DD. We can find that the combination of ZSDD and vtree2 yields the smallest size when \( n \) is large in \( G_1 \). In \( G_2 \), the combination of ZSDD and vtree3 is the best, but there is virtually no difference compared to vtree2. In \( G_3 \), vtree4 yields the smallest size. Although the best vtree depends on grammar, ZSDD is always superior to the other variants. ZSDD is known to be smaller than SDD when the model is small [Nishino 16], which might be why ZSDD is smaller than SDD in our experiment. Since all kinds of DD support same operations on parse trees, the smallest DD is the best to analyze sequences.

Table 2 shows the time to construct the ZSDD of \( G_1 \) respecting vtree2. We find that CYK-based method offers much faster construction than the naive CNF-based method even when the size of the ZSDD representing the set of all parse trees is quite large.

#### 5.2 Operation on Parse Trees

In this section, we consider sequence analysis with background knowledge which can be performed by applying operations on the set of parse trees and extracting parse trees that satisfy additional constraints. We treat three kinds of constraints on parse trees as follows.

- Specifying the partial structure.
- Restricting on the number of time a rule is used.
- Prohibition of the use of a particular pair of rules at the same time.

We can impose these constraints by using binary operations supported by ZSDDs. Let \( \alpha \) be the ZSDD representing the set of all parse trees and \( \beta \) be the ZSDD representing additional constraints. Then \( \langle \alpha \rangle \cap \langle \beta \rangle \) is the family of sets satisfying constraints. It holds that ZSDD supports \( \cap \) operation in the time linear to \( |\alpha| \cdot |\beta| \), the product of sizes of input ZSDDs.

We show how to construct ZSDD \( \beta \). To specify partial structure of parse trees, we choose variables corresponding the partial structure. Let \( B \) be the sets of variables that must be true in the parse tree. Since a variable corresponds to a part of parse trees, specifying the list of variables that must be used is specifying the partial structure. Then it holds that

\[
\langle \beta \rangle = \bigwedge_{b_{ijk} \in B} b_{ijk}. \tag{14}
\]

When restricting on the number of a rule \( r_k \) used is \( t \) times, we enumerate all the combination of used variables.

\[
B_k = \{ b_{ijk} \mid i \leq j \} \tag{15}
\]

\[
F_k = \{ B \subseteq B_k \mid |B| = t \} \tag{16}
\]

\( F_k \) is the family of sets and represents all the combination.
These Boolean functions can be compiled into ZSDD easily. We can extract parse trees satisfying the given constraints by applying intersection of the set of all parse trees and a Boolean function representing constraints.

We use some examples of concrete constraints and show experimental results of operations on the set of parse trees. On $G_1$, we extract parse trees with the following partial structures:

1) $B = \{b_{24}\}$.
2) $B = \{b_{14}, b_{18}, b_{24}, b_{56}, b_{58}\}$.

On $G_2$, we restrict on the usage of rules as follows:

3) $S \rightarrow aS$ can be used only once,
4) $S \rightarrow aS$ and $S \rightarrow Sa$ cannot be used at the same time.

Table 3 shows the result. The operation times are all less than 20msec. It shows that we can extract the set of parse trees that satisfy additional constraints in some reasonable time.

### 6. Upper Bound on the ZSDD Size

Given the above result, we discuss here the upper bound of ZSDD size.

**Theorem 6.1.** The size of a ZSDD representing a set of parse trees on vtree2 is $O(N2^N 3^{N/4} |V|^N)$, where $N$ is the length of sequence and $|V|$ is the number of non-terminal symbols.

In the following, we prove this theorem. Before analyzing the size, we define the underlying concepts. $b_{ij}$ is a Boolean variable corresponding to the initial rule used to generate $j - i + 1$ symbols $\alpha_i \ldots \alpha_j$. Let $j - i + 1$ be the height of $b_{ij}$, and $B_r$ be the set of variables whose height is $r$. In general, $|B_r| = n - r + 1$. Trace right edges from the root $n - r + 1$ times in vtree2, then the set of variables appearing in their left children equals $B_r$. For example, the left child of root vnode is $B_0 = \{b_{1N}\}$, the left child of its right child is $B_{N-1} = \{b_{1N-1}, b_{2N}\}$, and so on. If vnode $v$’s left child corresponds to $B_r$, we call $v$ a height-decompose vnode. We use a different analysis approach depending on whether the vnode that the decision node respects is a height-decompose vnode or not. In Figure 3(b), vnode1,2 are height-decompose vnodes.

We investigate the upper bound of (1) decision nodes and (2) child nodes of each decision node. Each decision node in ZSDD represents a subfunction. It is known that no two decision nodes that correspond to equivalent subfunctions can coexist in a ZSDD. Therefore, we iden...
affirm the upper bound of the size of decision nodes by discussing the number of different subfunctions that can appear in a ZSDD. We use a different analysis approach depending on whether the vnode that the decision node respects is a height-decompose vnode or not.

6.1 Decision Nodes Corresponding to Height-Decompose Vnode

The decision nodes associated with height-decompose vnodes denote subfunctions representing the set of partial parse trees whose height is at most $r$. Each partial tree whose height is at least $r + 1$. A subfunction represents a set of Boolean variables whose height is at most $r$, and the set forms a parse tree in combination with variables whose height is at least $r + 1$. Before finding the upper bound of the number of subfunctions, we define the grouping of symbols in a sequence as follows. Given a set of variables $X$ whose height is at least $r + 1$, we define the grouping for $X$ as follows. For each $i = 1, \ldots, n$, we choose $b_{jk} \in X$ such that the height of $b_{jk}$ is the smallest among the variables in $X$ satisfying $j \leq i \leq k$, and assign label $b_{jk}$ to the terminal symbol $\alpha_i$. Then, we define the grouping of $X$ by the partition such that two symbols belong in the same group if and only if they have the same label. If the set is valid, namely it can compose valid parse trees with variables whose height is at most $r$, each terminal symbol always has a label.

**Lemma 6.2.** If two valid assignments $I_1, I_2$ to Boolean variables whose height is at least $r + 1$ give equivalent groupings on the sequence data, two subfunctions $f, g$ over variables whose height is lower than $r$ that give valid parse trees when combined with each assignment $I_1, I_2$ are equivalent.

**Proof.** The above definition of grouping makes terminal symbols belonging to the same group always appear as a substring. If terminal symbols $\alpha_i, \alpha_{i+1}$ are in different groups, we cannot use $b_{jk}$ ($j \leq i < k$) whose height is $r$ or less to obtain valid parse trees. Therefore, a subfunction whose height is $r$ or less is defined as a set of partial parse trees on successive terminal symbols in each group. Partial parse trees on a group whose height is $r$ or less are consistent regardless of how the group is made. From the above, if groupings are equivalent, their subfunctions are also equivalent.

Figure 4 shows an example of equivalent subfunctions. The parse trees and assignments are different if we consider heights at least $r + 1$, but the subfunctions are equivalent.

From Theorem 6.2, we find the number of subfunctions can be bounded by using the number of groupings. Let maximum grouping number be the maximum group size of groupings. The following lemma shows that possible maximum grouping size number is bounded.

**Lemma 6.3.** Maximum grouping number $l$ satisfies $r + 1 \leq l \leq 2r$.

**Proof.** When the maximum grouping number is $l$, the largest height among label $b_{jk}$ of each terminal symbol $\alpha_i$ is $l$. Since we consider assignment with height $r + 1$ or more, $r + 1 \leq l$ must be satisfied.

On the other hand, we will show the above is invalid when $2r < l$. We assume assignment $I$ that satisfies $2r < l$.
and whose height is \( r + 1 \) or more. If we divide the largest group into two, the length of one side is \( r + 1 \) or more. Since the length of the group is equal to the height of the label, the height of label \( b_{l,k} \) of terminal symbols in this group is \( r + 1 \) or more. That is, the maximum grouping number of \( I \) is less than \( r + 1 \), which is invalid.

Lemma 6.4. Let \( a_{l,N} \) be the number of groupings whose maximum grouping number is \( l \) or less on \( N \) symbols, then

\[
a_{l,N} = \begin{cases} 
2^{N-1} & (N \leq l) \\
\sum_{i=N-l}^{N-1} a_{l,i} & (N > l),
\end{cases}
\]

Proof. In the case that \( N \leq l \), an arbitrary grouping satisfies the condition, so \( a_{l,N} = 2^{N-1} \). Otherwise, the length of head group is \( 1, \ldots, l \). We consider the remaining groupings; their lengths are \( N-1, \ldots, N-l \). Therefore, \( \sum_{i=N-l}^{N-1} a_{l,i} \) groupings exist in total.

Lemma 6.5. The upper bound of the number of subfunctions on \( B_r \) is taken to be \((a_{2r,N} - a_{r,N})|V|^N = O(2^N|V|^N)\).

Proof. The number of subfunctions on \( B_r \) is equal to the number of groupings of assignments whose height is \( r + 1 \) or more, and the maximum grouping number \( l \) satisfies \( r + 1 \leq l \leq 2r \) from Lemma 6.3. Since \( a_{r,n} \) is the number of groupings whose maximum number is \( r \) or less from Lemma 6.4, the upper bound on an assignment of non-terminal symbol is given by \( a_{2r,N} - a_{r,N} \). The top non-terminal symbol of each group has \( |V| \) cases at most and the number of group is \( N \) or less. Therefore, the number of subfunctions on \( B_r \) is taken to be \((a_{2r,N} - a_{r,N})|V|^N = O(2^N|V|^N)\).

6.2 Child Nodes Corresponding to Height-Decompose vnode

In the following, we give the upper bound of child nodes that a decision node of \( B_r \) may have. The number of child nodes is equal to the number of elements of \((X,Y)\) -partition on a subfunction. Since the number of elements of \((X,Y)\) -partition is bounded by the number of possible assignments that can form a valid parse tree, it corresponds to the number of possible assignments over \( B_r \) for a decision node that respects a height-decomposable vnode.

Lemma 6.6. Let \( f(g) \) be the number of assignments over \( B_r \) in a group whose length is \( g \). Then, \( f(g) = 1 \) when \( g < r + 1 \), \( f(g) = 3 \) when \( g = r + 1 \), and \( f(g) \leq 3 \) when \( r + 1 < g \leq 2r \).

Proof. When \( g < r + 1 \), all variables in \( B_r \) take 0, so \( f(g) = 1 \).

When \( g = r + 1 \), the number of variables in \( B_r \) is 2.

Both variables cannot take 1 at the same time. If \( r \geq 3 \), only variables whose heights are at most \( r \) can make a valid parse tree so both variables can take 0 at the same time (Impossible if \( r = 2 \)). Since there are three kinds of sets of used variables, namely one variable or none,

\[
f(g) = 3 \quad \text{if} \quad g = 2 \]

When \( r + 1 < g < 2r \), \( b_{1}, r, \ldots, b_{g-r+1,g} \) are the variables in \( B_r \). If \( b_{i, i+r} = 1 \) \((1 < i < g - r + 1)\), a variable whose height is at least \( r + 1 \) must take 1. This, however, contradicts the grouping, so only \( b_{1}, r \) or \( b_{g-r+1,g} \) can take 1. (1)If \( r + 1 < g < 2r \), neither variable can take 1 because of Eq. (10). Therefore, \( f(g) \leq 3 \). (2)If \( g = 2r \), both variables must take 1. Therefore, \( f(g) = 1 \leq 3 \).

Lemma 6.7. The number of assignments over \( B_r \) is \( 3^{N/(r+1)} \) when \( r \geq 3 \), and \( 2^{N/(r+1)} \) when \( r = 2 \).

Proof. If \( g_i \) is the length of the \( i \)-th group, the number of assignments over \( B_r \) is \( \prod_i f(g_i) \). When \( r \geq 3 \), it is the solution of the following optimization problem.

\[
\max \prod_i f(g_i) \text{ subject to } \sum_i g_i = N, \quad r + 1 \leq \max_i g_i \leq 2r
\]

When \( g_i = n/(r+1) \), \( \prod_i f(g_i) \) takes its maximum value, which is \( 3^{N/(r+1)} \). Similarly, when \( r = 2 \) we can obtain the maximum value, \( 2^{N/(r+1)} \).

Lemma 6.8. The size of ZSDD respecting height-decompose vnode is \( O(N2^N3^{N/4}) \).

Proof. The number of child nodes is \( O(2^N|V|^N)(1 + 2^{N/3} + \sum_{r=3}^{N} 3^{N/(r+1)}) = O(N2^N3^{N/4}|V|^N) \).

6.3 Other vnodes

We consider vnodes in partial vtrees that are right-linear vtrees. In right-linear vtrees, each decision node corresponds to the case that one variable is used. Therefore, the size is given by the upper bound of assignments of variables and the number of variables.

Lemma 6.9. The number of decision nodes on \( B_r \) is less than \( O(N2^N) \).

Proof. Since \( |B_r| = N - r + 1 \), the number of assignments is less than \( 2^{N-r+1} \) and \( 2^{N-r+1} < 2^N \). \( B_r \) has \( N \) or fewer variables; one variable corresponds to one decision node. Thus, the number of decision nodes is at most \( O(N2^N) \).

Decision nodes of right-linear vtrees have two child nodes, so the size in \( B_r \) is at most \( O(N2^N) \). Since the size of
height-decompose vnodes is $O(2^{N^2} \cdot 3^{N^4} |V|^N)$, it does not influence the order. Thus, the size of a ZSDD representing a set of parse trees on $N$ symbols is $O(N^2 2^{N^2} 3^{N^4} |V|^N)$ and Theorem 6.1 holds.

7. Conclusion

We compared the size of decision diagrams representing a set of parse trees of a simple CFG, and gave the upper bound of ZSDD size; ZSDD was found to offer an improvement of Boolean comparison method based on binary decision diagrams, Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 2–5 (1988)

[Fujita 91] Fujita, M., Matsunaga, Y., and Kakuda, T.: On variable ordering of binary decision diagrams for the application of multi-level logic synthesis, Proceedings of the European Conference on Design Automation, (EDAC), pp. 50–54 (1991)

[Jimenez 00] Jimenez, V. M. and Marzal, A.: Computation of the n best parse trees for weighted and stochastic context-free grammars, Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural Syntactic Pattern Recognition (SSPR), pp. 183–192 (2000)

[Knuth 11] Knuth, D. E.: The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part I, Addison-Wesley (2011)

[Manning 99] Manning, C. D. and Schütze, H.: Foundations of Statistical Natural Language Processing, Cambridge: MIT press (1999)

[Minato 13] Minato, S.: Techniques of BDD/ZDD: Brief history and recent activity, IEICE Transactions on Information and Systems, Vol. E96-D, No. 7, pp. 1419–1429 (2013)

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

[Nishino 16] Nishino, M., Yasuda, N., Minato, S., and Nagata, M.: Zero-suppressed sentential decision diagrams, Proceedings of the 30th Conference on Artificial Intelligence (AAAI), pp. 1058–1066 (2016)

[Nishino 17] Nishino, M., Yasuda, N., Minato, S., and Nagata, M.: Compiling graph substructures into sentential decision diagrams, Proceedings of the 31st Conference on Artificial Intelligence (AAAI 2017), pp. 1213–1221 (2017)

[Woelfel 05] Woelfel, P.: Bounds on the OBDD-size of integer multiplication via universal hashing, Journal of Computer and System Sciences, Vol. 71, No. 4, pp. 520–534 (2005)

[Yoshikawa 09] Yoshikawa, K., Riedel, S., Asahara, M., and Matsumoto, Y.: Jointly identifying temporal relations with markov logic, pp. 405–413 (2009)

Received March 16, 2018.

Author’s Profile

Kei Amii

He received the B. E. degree in informatics from Kyoto University in 2016. He is studying discrete structure at the master’s course in Kyoto University.

Masaaki Nishino (Member)

He received the B. E., M. E., and Ph.D. degrees in informatics from Kyoto University in 2006, 2008, and 2014, respectively. He joined Nippon Telegraph and Telephone Corporation in 2008, and he is currently with NTT Communication Science Laboratories. His current research interests include Data Structure, Natural Language Processing and Combinatorial Optimization.
Akihiro Yamamoto (Member)
He received the B.S. degree from Kyoto University in 1985, and Dr.Sci. degree from Kyushu University in 1990. Currently, he is a Professor of the Department of Intelligence Science and Technology, Graduate School of Informatics at Kyoto University. He has made research contributions to foundations of artificial intelligence science, with a particular focus on application of mathematical logic to machine learning. His recent research interest includes developing theory of machine learning and knowledge discovery with discrete mathematics. He is a member of IPSJ and JSSST.