algorithms

Article

Storing Set Families More Compactly with Top ZDDs

Kotaro Matsuda, Shuhei Denzumi * and Kunihiko Sadakane

Graduate School of Information Technology and Science, The University of Tokyo, Tokyo 158-8557, Japan; kotaro_matsuda@me2.mist.i.u-tokyo.ac.jp (K.M.); sada@mist.i.u-tokyo.ac.jp (K.S.)
* Correspondence: denzumi@mist.i.u-tokyo.ac.jp

Abstract: Zero-suppressed Binary Decision Diagrams (ZDDs) are data structures for representing set families in a compressed form. With ZDDs, many valuable operations on set families can be done in time polynomial in ZDD size. In some cases, however, the size of ZDDs for representing large set families becomes too huge to store them in the main memory. This paper proposes top ZDD, a novel representation of ZDDs which uses less space than existing ones. The top ZDD is an extension of the top tree, which compresses trees, to compress directed acyclic graphs by sharing identical subgraphs. We prove that navigational operations on ZDDs can be done in time poly-logarithmic in ZDD size, and show that there exist set families for which the size of the top ZDD is exponentially smaller than that of the ZDD. We also show experimentally that our top ZDDs have smaller sizes than ZDDs for real data.

Keywords: top tree; Zero-suppressed Decision Diagram; space-efficient data structure

check for updates

Citation: Matsuda, K.; Denzumi, S.; Sadakane, K. Storing Set Families More Compactly with Top ZDDs. *Algorithms* **2021**, *14*, 172. https://doi.org/10.3390/a14060172

Academic Editors: Frank Werner and Alberto Policriti

Received: 15 March 2021
Accepted: 29 May 2021
Published: 31 May 2021

1. Introduction

Zero-suppressed Binary Decision Diagrams (ZDDs) [1] are data structures which are derived from Binary Decision Diagrams (BDDs) [2] and represent a family of sets (combinatorial sets) in a compressed form using Directed Acyclic Graphs (DAGs). ZDDs are data structures specialized for processing set families and it is known that sparse set families can be compressed well. ZDDs support binary operations between two families of sets in time polynomial in the size of the ZDDs. Because of these advantages, ZDDs are used for combinatorial optimization problems and enumeration problems. Although ZDDs were originally developed for VLSI logic design, it has been shown that they can also be used effectively for graph problems and combinatorial problems because set families are a fundamental concept that appears in various fields [3,4]. Various other application researches related to intelligent information processing have been published, for example, in the fields of data mining [5–7], probabilistic graphical models such as Bayesian network [8,9], and game theory [10].

Though ZDDs can store set families compactly, their size may grow for some set families, and we need further compression. DenseZDDs [11] are data structures for storing ZDDs in a compressed form and supporting operations on the compressed representation. DenseZDDs represent a ZDD by a spanning tree of the DAG representing it, and an array of pointers between nodes on the spanning tree. Therefore its size is always linear to the original size, and to compress more, we need another representation.

Our basic idea for compression is as follows. In a ZDD, the identical sub-structures are shared and replaced by pointers. However identical sub-structures cannot be shared if they appear at different heights in ZDD. As a result, even if the DAG of a ZDD contains repetitive structures in the height direction, they cannot be shared.

For not DAGs but trees, there exists a data structure called top DAG compression [12], which can capture repetitive structures in the height direction. We extend it for DAGs and apply it to compress ZDDs which support the operations on compressed ZDDs.

This paper is an extended version of a conference paper published in SEA 2020 [13]. We rewrote the previous work so that readers not familiar with compression methods and operations on compressed data structure can understand our results easily. We added explanations of the DAG compression and how our algorithms work on compressed data with pseudo-codes. We also complement proof of space complexity.

The organization of the paper is as follows. In Section 2, we introduce our notation and data structures used throughout this paper. In Section 3, we introduce top trees and top DAGs. We show how to construct them and what operations they can perform. In Section 4, we propose our data structure, Top ZDD, and present the detailed components of Top ZDD, the theoretical analysis of the size, and the implementation method of the operations. In Section 5, we show the results of experiments for real and artificial data to evaluate construction time, search time and compactness of DenseZDDs. In Section 5, we show the experimental results on real and artificial data and compare Top ZDD, DenseZDD and standard ZDD to confirm the performance of the proposed method. In Section 6, we summarize the overall contents of this paper and discuss future work.

Our Contribution

We propose top ZDDs, which partition the edges of a ZDD into a spanning tree and other edges called complement edges, and store each of them in a compressed form. For the spanning tree, we use the top DAG compression, which represents a tree by a DAG with fewer nodes. For the complement edges, we store them in some nodes of the top DAG by sharing identical edges. We show that basic operations on ZDDs can be supported in $O(\log^2 n)$ time where n is the number of nodes of the ZDD. For further compression, we use succinct data structures for trees [14] and for bit vectors [15,16].

We show experimental results on the size of our top ZDDs and existing data structures, and query time on them. The results show that the top ZDDs use less space for most of the input data.

2. Preliminaries

Here we explain notations and basic data structures.

Let $C = \{1, \ldots, c\}$ be the universal set. Any set in this paper is a subset of C. The empty set is denoted by $\emptyset$. For a set $S = \{a_1, \ldots, a_s\} \subseteq C$ $(s \geq 1)$, its size is denoted by $|S| = s$. The size of the empty set is $|\emptyset| = 0$. A subset of the power set of C is called a set family. A set family is also called a family of sets, a set of sets, or a combinatorial set. If a set family $\mathcal{F}$ satisfies either $S \in \mathcal{F} \Rightarrow \forall k \in S, S\backslash\{k\} \in \mathcal{F}$ or $S \in \mathcal{F} \Rightarrow \forall k \in C, S \cup \{k\} \in \mathcal{F}$, $\mathcal{F}$ is said to be monotone. If the former is satisfied, $\mathcal{F}$ is monotone decreasing and the latter monotone increasing.

2.1. Zero-Suppressed Binary Decision Diagrams

Zero-suppressed Binary Decision Diagrams (ZDDs) [1] are data structures for manipulating finite set families. A ZDD is a directed acyclic graph (DAG) $G = (V, E)$ with a root node satisfying the following properties. A ZDD has two types of nodes; branching nodes and terminal nodes. There are exactly two terminal nodes; $\bot$ and $\top$. These terminal nodes have no outgoing edges. Each branching node v has an integer label $\ell(v) \in \{1, \ldots, c\}$, and also has two outgoing edges 0-edge and 1-edge. The node pointed to by the 0-edge (1-edge) of v is denoted by $v_0 = zero(v)$ $(v_1 = one(v))$. If for any branching node v it holds $\ell(v) < \ell(v_0)$ and $\ell(v) < \ell(v_1)$, then the ZDD is said to be ordered. In this paper, we consider only ordered ZDDs. For convenience, we assume $\ell(v) = c + 1$ for terminal nodes v. We divide the nodes of the ZDD into layers $L_1, \ldots, L_{c+1}$ Note that if $i \geq j$ there are no edges from layer L_i to layer L_j. The number of nodes in ZDD G is denoted by $|G|$ and called the size of the ZDD. On the other hand, the data size of a ZDD stands for the number of bits used in the data structure representing the ZDD.

In a ZDD, each node represents a set family. When a ZDD is single-rooted, the set family represented by the root is called the set family represented by the ZDD. The set family represented by a ZDD is defined as follows.

Definition 1 (The set family represented by a ZDD)**.** *Let v be a node of a ZDD and $v_0 = zero(v)$, $v_1 = one(v)$. Then the set family $\mathcal{F}_v$ represented by v is defined as follows. ZDD $G = (V, E)$, rooted at node $v \in V$, represents a finite family of sets $\mathcal{F}_v$ on C defined recursively as follows:*

1. *If v is a terminal node: if $v = \top$, $\mathcal{F}_v = \{\emptyset\}$, if $v = \bot$, $\mathcal{F}_v = \emptyset$.*
2. *If v is a branching node: $\mathcal{F}_v = \{S \cup \{\ell(v)\} \mid S \in \mathcal{F}_{v_1}\} \cup \mathcal{F}_{v_0}$.*

For the root node r of ZDD G, $\mathcal{F}_r$ corresponds to the set family represented by the ZDD G. This set family is also denoted by $\mathcal{F}_G$.

A family of sets can be represented by a binary tree as shown in Figure 1. Let $\mathcal{F}$ be a family of sets. For a set S, we traverse the 1-edge at the nodes whose labels are elements in S, and the 0-edge otherwise. Then we reach the $\top$ if $S \in \mathcal{F}$, and the $\bot$ otherwise. Any family of sets can be represented by a binary tree of height c. This is almost equivalent to the bitstring representation of length 2^n. This tree structure can be used for compression, however. A ZDD can be constructed by deleting redundant nodes from the binary tree and merging equivalent nodes. Every path from the root to the terminal $\top$ on ZDD G have a one-to-one correspondence to a set $S = \{a_1, \ldots, a_{|S|}\}$ in the set family represented by G. Consider a traversal of nodes from the root towards terminals so that for each branching node v on the path, if $\ell(v) \notin S$ we go to $v_0 = zero(v)$ from v, and if $\ell(v) \in S$ we go to $v_1 = one(v)$ from v. By repeating this process, if $S \in \mathcal{F}_G$ we arrive at $\top$, and if $S \notin \mathcal{F}_G$ we arrive at $\bot$ or find no branching node corresponding to some $a_i \in S$ during the process. An example is shown in Figure 2. ZDDs also support various set operations, which can be used to obtain new ZDDs from multiple ZDDs, as shown in Figure 3.

2.2. Succinct Data Structures

Succinct data structures are data structures whose size match the information-theoretic lower bound. Formally, a data structure is succinct if any element of a finite set U with cardinality L is encoded in $\log_2(L) + o(\log_2(L))$ bits. In this paper, we use succinct data structures for bit-vectors and trees.

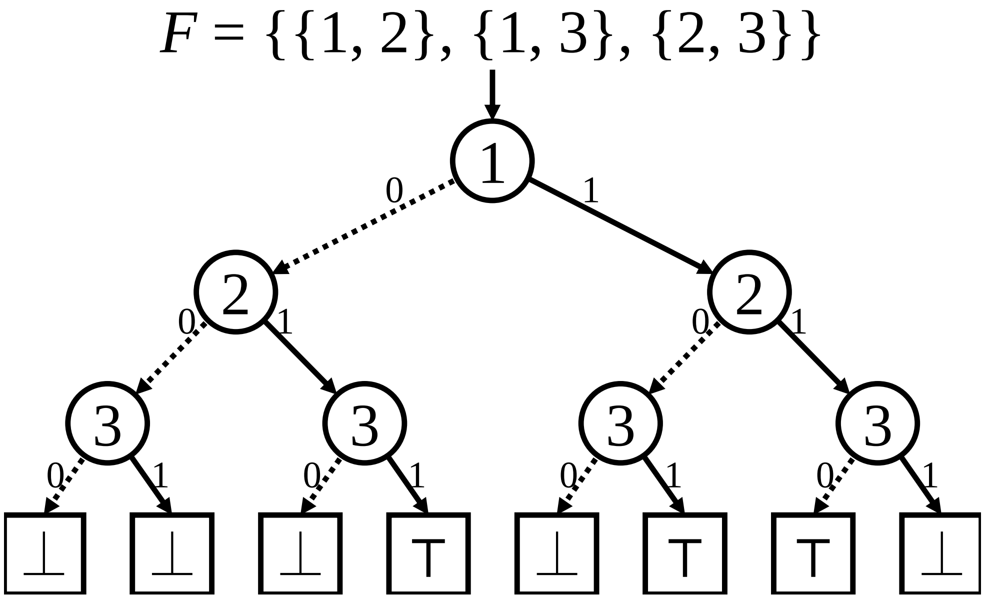

Figure 1. The binary tree that represents a family of sets $F = \{\{1,2\},\{1,3\},\{2,3\}\}$.

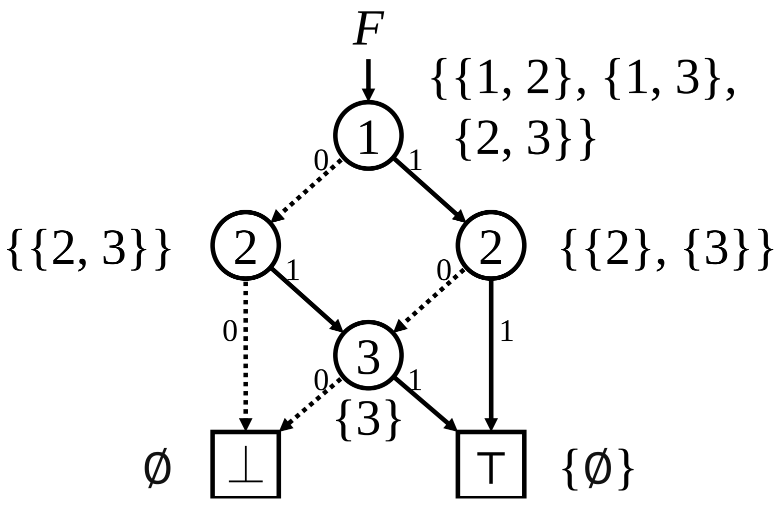

Figure 2. The ZDD obtained from the binary tree in Figure 1 by merging equivalent nodes and deleting redundant nodes. Terminal nodes and branching nodes are depicted by squares and circles, respectively, and 0-edges and 1-edges are depicted by dashed and solid lines, respectively. The family of sets represented by each node is indicated beside it.

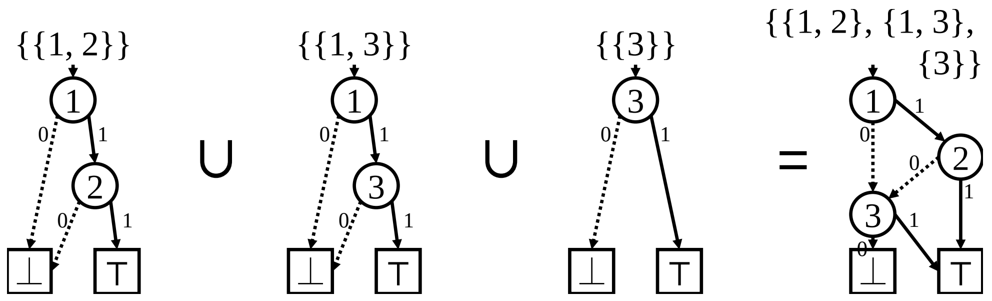

Figure 3. Example of how ZDDs are utilized to obtain a new ZDD from given ZDDs via a set operation.

2.2.1. Bit Vectors

Bit vectors are the most basic succinct data structures. A length-n sequence $B \in \{0,1\}^n$ of 0's and 1's is called a bit vector. On this bit vector we consider the following operations:

- $access(B,i)$ $(1 \le i \le n)$: returns $B[i] \in \{0,1\}$, the i-th entry of B.
- $rank_c(B,i)$ $(1 \le i \le n, c = 0,1)$: returns the number of c in the first i bits of B.
- $select_c(B,j)$ $(1 \le j \le n, c = 0,1)$: returns the position of the j-th occurrence of c in B.

The following result is known.

Theorem 1. ([15]) *For a bit vector of length n, using a $n + \mathrm{O}(n \log\log n/\log n)$-bit data structure constructed in $\mathrm{O}(n)$ time, $access(B,i), rank_c(B,i), select_c(B,j)$ are computed in constant time on the word-RAM with word length $\Omega(\log n)$.*

Consider a bit vector of length n with m 1's. For sparse bit vectors, namely, that consisting of a binary sequence with only $m = \mathrm{o}(n/\log n)$ 1's, we can obtain a more space-efficient data structure.

Theorem 2. ([16]) *For a bit vector B of length $n = 2^w$ with m ones, $select_1(B,i)$ is computed in constant time using a $2m + \mathrm{O}(m \log\log m/\log m)$-bit data structure.*

Note that on this data structure, $rank_0, rank_1, select_0$ takes $\mathrm{O}(\log m)$ time.

2.2.2. Trees

Consider a rooted ordered tree with n nodes. The information-theoretic lower bound on the size of such trees is $2n - \Theta(\log n)$ bits. When we traverse a tree by depth-first search and number each node in the order of the first visit, the number is called the preorder id of the node. When nodes are given ids, we may refer to each node as its number. We want to support the following operations:

- $parent(x)$: returns the parent of node x.
- $firstchild(x), lastchild(x)$: returns the first/last child of node x.
- $nextsibling(x), prevsibling(x)$: returns the next/previous sibling of node x.
- $isleaf(x)$: returns if node x is a leaf or not.
- $preorder\_rank(x)$: returns the preorder id of node x.
- $preorder\_select(i)$: returns the node with preorder id i.
- $leaf\_rank(x)$: returns the number of leaves whose preorder ids are smaller than that of node x.
- $leaf\_select(i)$: returns the i-th leaf in preorder.
- $depth(x)$: returns the depth of node x, that is, the distance from the root to x.
- $subtreesize(x)$: returns the number of nodes in the subtree rooted at node x.
- $lca(x, y)$: returns the lowest common ancestor (LCA) between nodes x and y.

Theorem 3. ([14]) *On the word-RAM with word length $\Omega(\log n)$, the above operations are done in constant time using a $2n + o(n)$-bits data structure.*

We call this the balanced parenthesis (BP) representation in this paper.

2.3. DenseZDD

A DenseZDD [11] is a static representation of a ZDD with attributed edges [17] by using some succinct data structures. In comparison to the standard ZDD, a DenseZDD provides a much faster membership operation and less memory usage for most cases. When we construct a DenseZDD from a given ZDD, dummy nodes are inserted so that $\ell(v_0) = \ell(v) + 1$ holds for each branching node v for fast traversal. A Spanning Tree consisting of all inverted 0-edges is represented by a simple BP. Notice that, each node always has one 0-edge, and ZDDs have no cycles and the terminal node is only $\perp$ if we employ the attributed edge technique. Therefore, if you traverse only the 0-edges from any branching node, you will always end up at the $\perp$. The inverted 0-edges thus form a spanning tree rooted at the $\perp$. The DenseZDD is a combination of this BP and other succinct data structures that represent the remaining information of the given ZDD. Its size is much smaller than the standard ZDD size and it can execute fast membership operations. A DenseZDD represents 0-edges and 1-edges respectively in a different way. For 0-edges, the spanning tree of a ZDD formed by the 0-edges is stored. We give preorder id for each node by a depth-first traversal of the spanning tree. Some dummy nodes are added such that $\ell(v_0) = \ell(v) + 1$ holds for each node v. Spanning tree and dummy node information are represented by BP and Fully Indexable Dictionary (FID), respectively. For 1-edges, an array stores all the preorder ids of 1-children together.

3. Top Tree and Top DAG

We explain DAG compression [18–20] and top DAG compression [12] to compress labeled rooted trees.

3.1. DAG Compression

DAG compression is a scheme to represent a labelled rooted tree by a smaller DAG by sharing identical subtrees appearing repeatedly. See Figure 4 for an example.

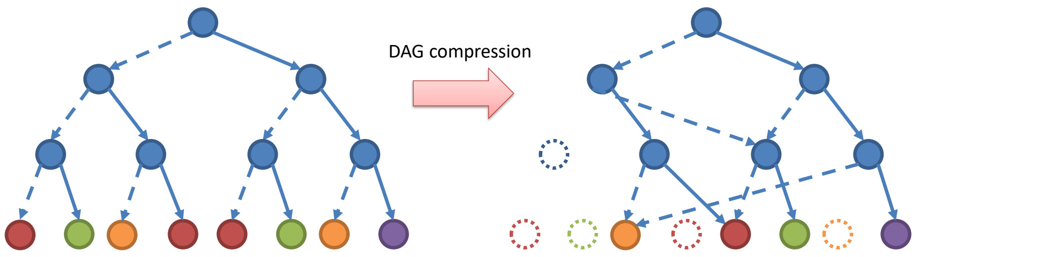

Figure 4. An example of DAG compression. Dashed lines are 0-edges, and solid lines are 1-edges. Leaves of the same colour represent the identical subtrees.

When we repeat sharing common sub-structures until the sub-structure rooted at each node u becomes unique, the resulting structure is called the minimal DAG. The minimal DAG is unique and it can be obtained in $\mathrm{O}(n)$ time for a tree with n nodes [19]. We call representing a tree by the minimal DAG as DAG compression. After DAG compression, the number of nodes may be exponentially smaller than the original tree. On the other hand, in the worst case, the number of nodes never decreases. This is because DAG compression can share only identical subtrees. Here a subtree means the tree consisting of all the descendants of a node. For example, a path in which all nodes have the same label can be represented by just storing the length of the path and the label, DAG compression cannot capture this repeated structure and cannot compress the path.

3.2. *Top DAG Compression*

Top DAG compression is a compression scheme for labelled rooted trees by converting the input tree into top tree [21] and then compress it by DAG compression [18–20]. Top DAG compression can compress repeated sub-structures (not only subtrees). DAG compression can share identical subtrees, but it cannot share similar structures at different heights. This is made possible by top DAG compression. For example, a path of length n with identical labels cannot be compressed at all by DAG compression but can be represented by a top DAG with $\mathrm{O}(\log n)$ nodes. Also, for a tree with n nodes, top DAG supports the following operations in $\mathrm{O}(\log n)$ time: accessing a node label, computing the subtree size, and tree navigational operations such as first child and parent. Here we explain the top tree and its greedy construction algorithm. We also explain operations on top DAGs.

3.2.1. Top Tree

The top tree [21] for a labelled rooted tree T is a binary tree $\mathcal{T}$ representing the merging process of clusters of T defined as follows. A top tree is a meta-binary tree for managing tree structure, and its purpose is to transform a given tree into a somewhat balanced binary tree. We assume that all edges in the tree are directed from the root towards leaves, and an edge (u, v) denotes the edge from node u to node v. Clusters are subsets of T with the following properties.

- A cluster is a subset F of the nodes of the original tree T such that nodes in F are connected in T.
- F forms a tree and we regard the node in F closest to the root of T as the root of the tree. We call the root of F as the *top boundary node*.
- F contains at most one node having directed edges to outside of F. If there is such a node, it is called the *bottom boundary node*.

A boundary node is either a top boundary node or a bottom boundary node.

By merging two adjacent clusters, we obtain a new cluster, where merge means to take the union of node sets of two clusters and make it the node-set of the new cluster. There are five types of merges, as shown in Figure 5. In the figure, ellipses are clusters before merge, black circles are boundary nodes of new clusters, and white circles are not boundary nodes in new clusters.

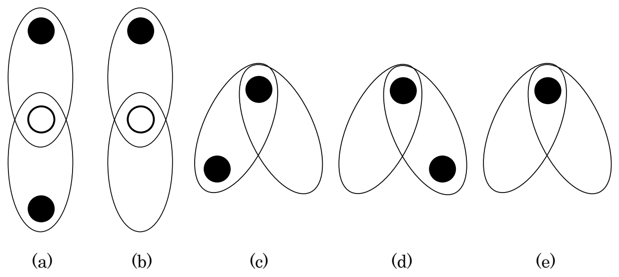

Figure 5. Merging clusters. These five merges are divided into two. (**a**,**b**) Vertical merge: two clusters can be merged vertically if the top boundary node of one cluster coincides with the bottom boundary node of the other cluster, and there are no edges from the common boundary node to nodes outside the two clusters. (**c**–**e**) Horizontal merge: two clusters can be merged horizontally if the top boundary nodes of the two clusters are the same and at least one cluster does not have the bottom boundary node.

Examples of merge are shown in Figures 6 and 7.

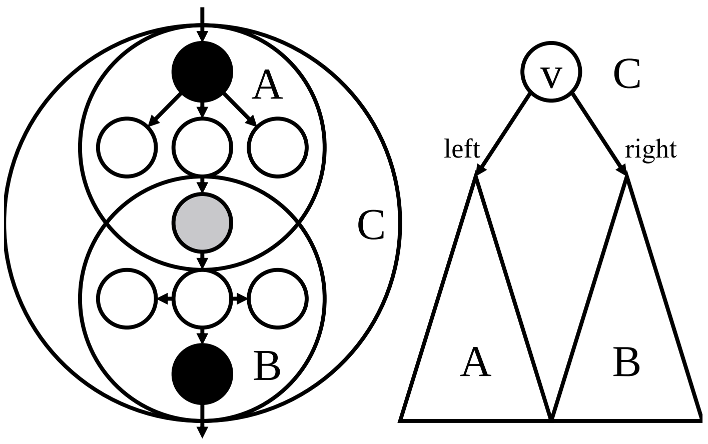

Figure 6. An example of vertical merge of type (a). The left side is the clusters in the given tree and the right side is the corresponding top tree. A black node above is a boundary node of the clusters A and C. A black node below is a boundary node of the clusters B and C. A light gray node is a boundary node of the clusters A and B, but not of C.

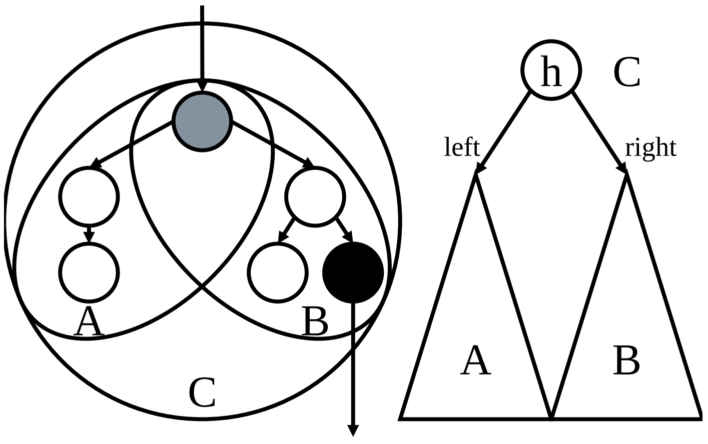

Figure 7. An example of horizontal merge of type (d). The left side is the clusters in the given tree and the right side is the corresponding top tree. A black node is a boundary node of the clusters B and C. A dark gray node is a boundary node of clusters A, B and C.

The top tree of the tree T is a binary tree $\mathcal{T}$ satisfying the following conditions. Note that we use terms nodes and edges for the original tree T, and vertices and arcs for the top tree $\mathcal{T}$.

- Each leaf of the top tree corresponds to a cluster with the endpoints of an edge of T.
- Each internal vertex of the top tree corresponds to the cluster made by merging the clusters of its two children. This merge is one of the five types in Figure 5.
- The cluster of the root of the top tree is T itself.

Figure 8 is an example of the top tree $\mathcal{T}$ for an unlabeled tree T.

V and H in vertices of the top tree represent vertical and horizontal merges, respectively. Letters on the right of vertices show the type of merges in Figure 5. Clusters corresponding to vertices of the top tree are also shown. Red nodes show boundary nodes of clusters.

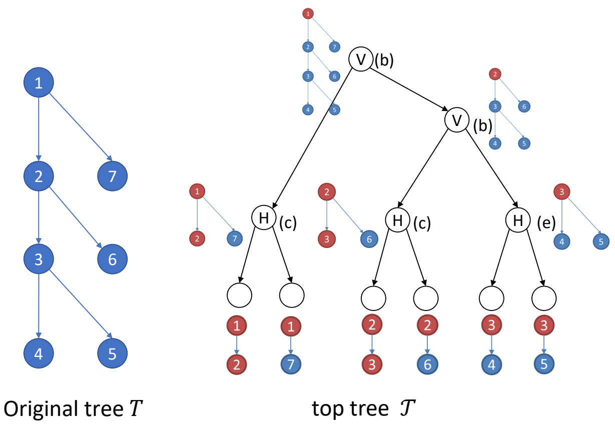

Figure 8. An example of top tree. Alongside each vertex of the top tree, the corresponding cluster is drawn. Red nodes show boundary nodes.

We call the DAG obtained by DAG compression of the top tree $\mathcal{T}$ as top DAG $\mathcal{T}D$, and the operation to compute the top DAG $\mathcal{T}D$ from tree T is called top DAG compression [12].

We define labels of vertices in the top tree to apply DAG compression as follows. For a leaf of the top tree, we define its label as the pair of labels of two endpoints of the corresponding edge in T. For an internal vertex of the top tree, its label must have information about cluster merge. It is enough to consider three types of merges, not five as in Figure 5. For vertical merges, it is not necessary to store the information that the merged cluster has a bottom boundary node or not. For horizontal merges, it is enough to store if the left cluster has a bottom boundary node or not. From this observation, we define labels of internal vertices as follows.

- For vertices corresponding to vertical merge: we set their labels as V.
- For vertices corresponding to horizontal merge: we set their labels as $\mathrm{H_L}$ if the left child cluster has the bottom boundary node, or $\mathrm{H_R}$ if the right child cluster has the bottom boundary node. If both children do not have bottom boundary nodes, the label can be arbitrary.

Top trees created by a greedy algorithm satisfy the following.

Theorem 4. ([12]) *Let n be the number of nodes of a tree T. Then the height of* top tree $\mathcal{T}$ *created by a greedy algorithm is* $\mathrm{O}(\log n)$.

3.2.2. Operations on Top DAGs

Consider to support operations on a tree T which is represented by top DAG $\mathcal{T}D$. From now on, a node x in T stands for the node with preorder id x in T. By storing additional information to each vertex of the top DAG, the following tree operations can be supported [12].

- $Access(x)$: returns the label of x.
- $Decompress(x)$: returns the subtree $T(x)$ of T rooted at x.
- $Parent(x)$: returns the preorder id of the parent of x.

- $Depth(x)$: returns the depth, i.e., the distance to the root, of x.
- $Height(x)$: returns the height, i.e., the distance to the farthest leaf, of x.
- $Size(x)$: returns the subtree size rooted at x.
- $Firstchild(x)$: returns the preorder id of the first child of x, or reports x is a leaf.
- $NextSibling(x)$: returns the preorder id of the next sibling of x, or reports it does not exist.
- $LevelAncestor(x, i)$: returns the preorder id of the ancestor of x with distance i from x.
- $LCA(x, y)$: returns the preorder id of the lowest common ancestor between x and y.

For a tree with n nodes, all operations except $Decompress(\cdot)$ are done in $O(\log n)$ time, and $Decompress(\cdot)$ is done in $O(\log n + |T(x)|)$ time.

We explain the algorithm for $Access(x)$ based on a recursive function $sub(u, k)$ for a vertex u of the top DAG and an integer k. The function $sub(u, k)$ returns the label of the node with local preorder id k inside the cluster corresponding to vertex u of the top DAG. Note that $Access(x)$ can be achieved by doing $sub(r, x)$ because the cluster corresponding to r is the whole tree. The function $sub(u, k)$ is computed recursively as follows. If u has no outgoing edges, u is a leaf of the top DAG. Thus, u corresponds to a cluster consisting of a single edge. In this case k is either 0 or 1. For such a cluster, only the label of the two endpoints of the edge is stored. Let s and t be the label of the starting point and ending point, respectively. If $k = 0$, it returns $sub(u, k) = s$; if $k = 1$, it returns $sub(u, k) = t$.

If u is a vertex corresponding to horizontal merge, let v and w be the left and the right child of u, respectively, let $cl(u), cl(v), cl(w)$ be the corresponding cluster of u, v, w, respectively, and let $C(u), C(v), C(w)$ be their sizes. If we traverse nodes of $cl(u)$ in preorder, the first one is the node shared by $cl(v)$ and $cl(w)$, then all the nodes of $cl(v)$ appear and finally nodes of $cl(w)$ are traversed. Therefore $sub(u, k)$ is computed as follows.

- If $k = 1$: the corresponding node for $sub(u, k)$ is contained in both $cl(v)$ and $cl(w)$, and it holds $sub(u, k) = sub(v, 1) = sub(w, 1)$.
- If $2 \leq k \leq C(v)$: the corresponding node for $sub(u, k)$ is in $cl(v)$, and it holds $sub(u, k) = sub(v, k)$.
- If $C(v) + 1 \leq k \leq C(v) + C(w) - 1$: the corresponding node for $sub(u, k)$ is in $cl(w)$, and it holds $sub(u, k) = sub(w, k - C(v) + 1)$.

This function can be computed if the cluster size $C(\cdot)$ is known for each vertex of the top DAG.

Finally, if u is a vertex corresponding to vertical merge, define v, w, $cl(u)$, $cl(v)$, $cl(w)$, $C(u)$, $C(v)$, $C(w)$ similarly to the horizontal merge case. We need another value $D(v)$, which is the preorder id of the bottom boundary of $cl(v)$. If we traverse nodes of $cl(u)$ in preorder, we first visit nodes of $cl(v)$ with local preorder id up to $D(v)$, then visit all nodes of $cl(w)$, then finally visit nodes of $cl(v)$ with local preorder id from $D(v) + 1$. Therefore $sub(u, k)$ is computed as follows.

- If $1 \leq k \leq D(v) - 1$: the corresponding node for $sub(u, k)$ is in $cl(v)$, and it holds $sub(u, k) = sub(v, k)$.
- If $k = D(v)$: the corresponding node for $sub(u, k)$ is the bottom boundary node of $cl(v)$, which is also the top boundary node of $cl(w)$, and it holds $sub(u, k) = sub(v, k) = sub(w, 1)$.
- If $D(v) + 1 \leq k \leq D(v) + C(w)$: the corresponding node for $sub(u, k)$ is in $cl(w)$, and it holds $sub(u, k) = sub(w, k - D(v) - 1)$.
- If $D(v) + C(w) + 1 \leq k \leq C(v) + C(w) - 1$: the corresponding node for $sub(u, k)$ is in $cl(v)$, and it holds $sub(u, k) = sub(v, k - C(w) - 1)$.

This function can be computed recursively if $C(\cdot)$ and $D(\cdot)$ are known. Algorithm 1 shows a pseudo code.

Algorithm 1 *Access*(x): computes the label of a node whose preorder id in the tree representing the top DAG is x.

```
Input: Preorder $x$
Output: The label of node $x$
 1: $r \leftarrow$ the root of top DAG
 2: return SUB$(r, x)$
 3: procedure SUB$(u, k)$
 4:     if vertex $u$ corresponds to a cluster with a single edge $e$ then
 5:         if $k = 1$ then
 6:             return (the label $s$ of the starting point of $e$)
 7:         else
 8:             return (the label $s$ of the ending point of $e$)
 9:     else
10:         $v \leftarrow$ (the left child of $u$)
11:         $w \leftarrow$ (the right child of $u$)
12:         $C(v) \leftarrow$ (the size of the cluster of $v$)
13:         $C(w) \leftarrow$ (the size of the cluster of $w$)
14:         if vertex $u$ is horizontal merge then
15:             if $1 \leq k \leq C(v)$ then
16:                 return SUB$(v, k)$
17:             else
18:                 return SUB$(w, k - C(v) - 1)$
19:         else
20:             $D(v) \leftarrow$ (the preorder id of the bottom boundary node of the cluster of $v$)
21:             if $1 \leq k \leq D(v)$ then
22:                 return SUB$(v, k)$
23:             else if $D(v) + 1 \leq k \leq D(v) + C(w)$ then
24:                 return SUB$(w, k - D(v) - 1)$
25:             else
26:                 return SUB$(v, k - C(w) - 1)$
```

4. Top ZDD

We explain our top ZDD, which is a representation of ZDD by top DAG compression. Though it is easy to apply our compression scheme for general rooted DAGs, we consider the only compression of ZDDs.

A ZDD $G = (V, E)$ is a directed acyclic graph in which branching nodes have labels $\ell(\cdot)$ and edges have labels 0 or 1. We can regard it as a graph with only edges being labelled. For each edge (u, v) of ZDD G, we define its label as a pair (edge label $0/1$, $\ell(v) - \ell(u)$) if v is a branching node, or a pair (edge label $0/1$, $\bot/\top$) if v is a terminal node. Note that in the second case, we are not taking the difference of labels. We only care whether the terminal is $\top$ or $\bot$. In practice, we can use $c + 1$ instead of $\bot$, and $c + 2$ instead of $\top$ for the second element of the second case. It is enough to distinguish the above two cases because $\ell(v) - \ell(u)$ is always less than c for any u, v. Below we assume ZDDs have labels for only edges, and 0-edge comes before 1-edge for each node.

Next, we consider top trees for edge-labelled trees. The difference from node-labelled trees is only how to store the information for single edge clusters. In Section 3.2.1, we stored labels for both endpoints of edges. We change this for storing only edge labels.

The top ZDD is constructed from a ZDD $G = (V, E)$ as follows.

1. We perform a depth-first traversal from the root of G and obtain a spanning tree T of all branching nodes. During the process, we do not distinguish 0-edges and 1-edges, and terminal nodes are not included in the tree. Nodes of the tree are identified with their preorder ids in T. If we say node u, it means the node in T with preorder id u. We call edges of G not included in T as *complement edges*.
2. We convert the spanning tree T to a top tree $\mathcal{T}$ by the greedy algorithm.

3. For each complement edge (u, v), we store its information in a node of $\mathcal{T}$ as follows. If v is a terminal, let a be the vertex of the top tree corresponding to the cluster of a single edge between u and its parent in T. Note that a is uniquely determined. Then we store a triple $((u, v)$, edge label 0/1, $\bot/\top)$ in a. If v is a branching node, we store the information of the complement edge to a vertex of T corresponding to a cluster containing both u and v. The information to store is a triple $((u, v)$, edge label 0/1, $\ell(u) - \ell(v))$. Each vertex stores such information as follows. Let a, b be the vertices of the top tree corresponding to the clusters of single edges towards u, v in T, respectively. Then we store the triple in the lowest common ancestor $\mathrm{lca}(a, b)$ in T. Here the information (u, v) represents local preorder ids inside the cluster corresponding to $\mathrm{lca}(a, b)$. Note that $\mathrm{lca}(a, b)$ may not be the minimal cluster including both u and v.
4. We create a top DAG $\mathcal{TD}$ by DAG compression by sharing identical clusters. To determine the identity of two clusters, we compare them together with the information of complement edges in the clusters stored in step 3. Complement edges that do not appear in multiple clusters are moved to the root of T.

Figure 9 shows an example of how our algorithm works. In this figure, we show the clusters corresponding to each node. But, they are not stored explicitly.

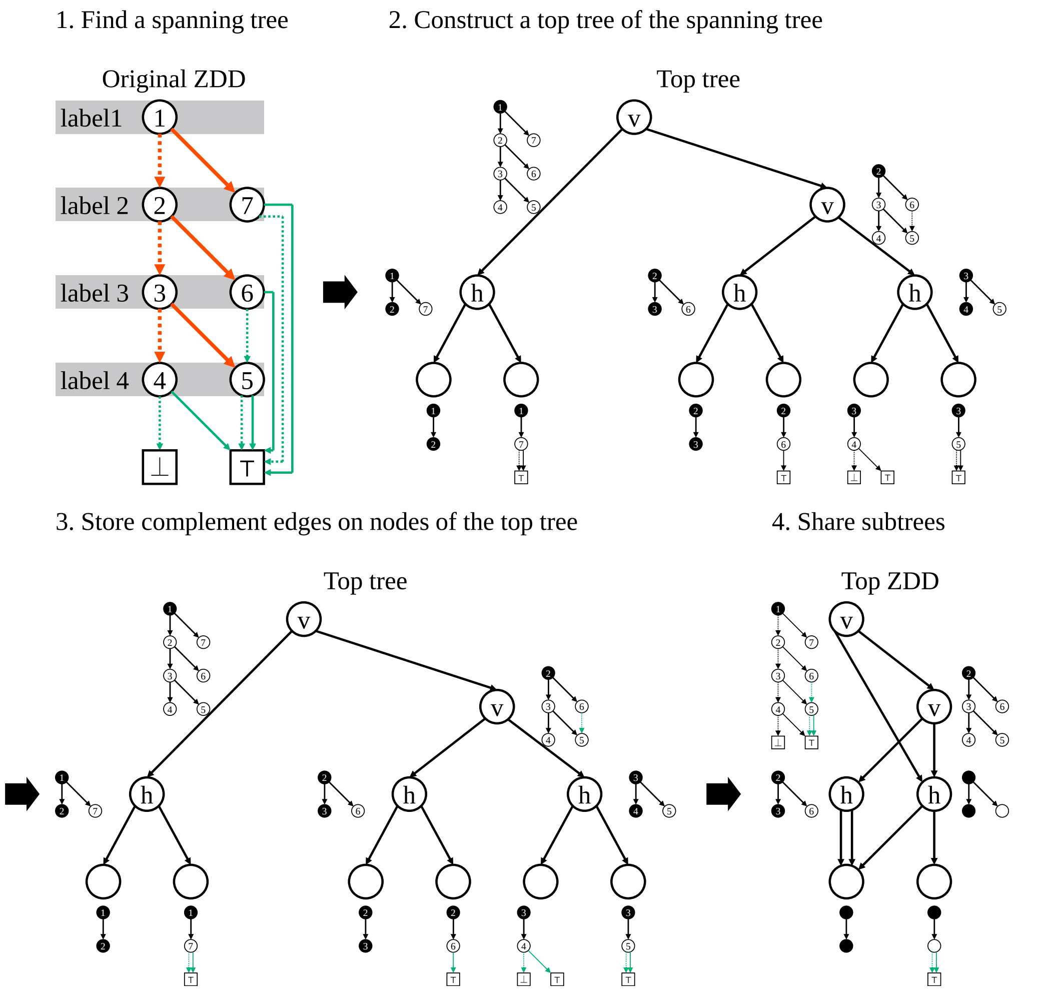

Figure 9. Running example of the Top ZDD construction algorithm. In each branching node of the original ZDD, its preorder id is denoted instead of its label. Red edges are spanning tree edges and green edges are complement edges. For each vertex of the top tree, the corresponding cluster and the complement edges stored are shown. Boundary nodes of each cluster are denoted as black nodes.

It is crucial to choose an appropriate data structure to store each information to achieve smaller space consumption. For example, in Section 3.2.2, we explained that each vertex of the top DAG stores the cluster size etc., this is redundant and space can be reduced. Next, we explain our space-efficient data structure which is enough to support efficient queries in detail.

4.1. Details of the Data Structure

We need the following information to recover the original ZDD from a top ZDD.

- Information about the structure of top DAG $\mathcal{TD}$.
- Information about each vertex of $\mathcal{TD}$. There are three types of vertices: vertices corresponding to a leaf of the top tree, vertices representing vertical merge, and vertices representing horizontal merge. For each type, we store different information.
- Information about complement edges.

We show space-efficient data structures for storing this information. The data structure to be used is chosen depending on the ratio of 1's. If the ratio of 1's is less than $\frac{1}{4}$, then we use the SparseArray [22] to compress a bit vector. Or if the ratio of 1's is between $\frac{1}{4}$ and $\frac{3}{4}$, then we use the succinct bit vector with constant time rank/select support [15].

And, we use the SparseArray for the bit vector whose 0/1 are flipped if the ratio of 0's is less than $\frac{1}{4}$. To store an array of non-negative integers, we use $\lfloor \log_2 m \rfloor$ bits for each entry where m is the maximum value in the array. Let n denote the number of branching nodes of a ZDD. We use $n+1, n+2$ to represent terminals $\bot, \top$, respectively.

4.1.1. The Data Structure for the Structure of Top DAG $\mathcal{TD}$

We store top DAG $\mathcal{TD}$ after converting it to a tree. We make tree T' by adding dummy vertices to $\mathcal{TD}$. For each vertex x of $\mathcal{TD}$ whose in-degree is two or more, we do the following.

1. Let $a_1, \cdots, a_t$ be the vertices of $\mathcal{TD}$ from which there are edges towards x. Note that there may exist identical vertices among them corresponding to different edges. We create $t-1$ dummy vertices $d_1, \cdots, d_{t-1}$.
2. For each $1 \le i \le t-1$, remove edge (a_i, x) and add edge (a_i, d_i).
3. For each dummy vertex d_i, we store information about a pointer to x. In our implementation, we store the preorder id of x in T' from which the dummy vertices are removed.

Then we can represent the structure of the top DAG by the tree T' and the pointers from the dummy vertices.

Next, we explain how to store T' and the information about the dummy vertices. The structure of T' is represented by the BP sequence [14]. There are two types of leaves in T': those which exist in the original top DAG, and those for the dummy vertices. To distinguish them, we use a bit vector. Let m be the number of leaves in T'. We create a bit vector B_{dummy} of length m whose i-th bit corresponds to the i-th leaf of T' in preorder. We set $B_{dummy}[i] = 1$ if the i-th leaf is a dummy vertex, and we set $B_{dummy}[i] = 0$ otherwise.

We add additional information to dummy vertices to support efficient queries. We define an array *clsize* of length D where D is the number of dummy vertices. For the i-th dummy vertex in preorder, let s_i be the vertex pointed to by the dummy vertex. We define $\mathit{clsize}[k] = \sum_{i=1}^{k}$(the number of vertices in the cluster represented by s_i). That is, $\mathit{clsize}[k]$ stores the cumulative sum of cluster sizes up to k. This array is used to compute the cluster size for each vertex efficiently.

4.1.2. Information on Vertices

We explain how to store information on vertices of T' except for dummy vertices. Each vertex corresponding to a leaf in the original top tree is a cluster for a single edge in the spanning tree, and it is a non-dummy leaf in T'. We sort these vertices in preorder in T' and store information on edges towards them in the following two arrays. One is an array *label_span* to store differences of levels between endpoints of edges. Let u and v be the starting and the ending points of the single edge of the cluster corresponding to the i-th leaf, respectively. Then we set $\mathit{label\_span}[i] = \ell(v) - \ell(u)$. The other is an array *type_span* to store if an edge is 0-edge or 1-edge. We set $\mathit{type\_span}[i] = 0$ if the edge corresponding to the i-th vertex is a 0-edge, and $\mathit{type\_span}[i] = 1$ otherwise.

Each vertex of T' corresponding to vertical merge or horizontal merge is an internal vertex. We sort internal vertices of T' in their preorder. Then we make a bit vector B_H so that $B_H[i] = 0$ if the i-th vertex stands for vertical merge, and $B_H[i] = 1$ if it stands for horizontal merge. For vertices corresponding to horizontal merge, we do not store additional information. For vertices corresponding to vertical merge, we use arrays *preorder_diff* and *label_diff* to store the differences of preorder ids and levels between the top and the bottom boundary nodes of the merged cluster. Let x_i be the i-th vertex in preorder corresponding to vertical merge, cl_i be the cluster corresponding to x_i, t_i be the top boundary node of cl_i, and b_i be the bottom boundary node of cl_i. Note that t_i and b_i are nodes of the ZDD. Then we set $\mathit{preorder\_diff}[i] = (\text{the local preorder id of } b_i \text{ inside cluster } cl_i)$ and $\mathit{label\_diff}[i] = \ell(b_i) - \ell(t_i)$.

4.1.3. Information on Complement Edges

Complement edges are divided into two groups: those stored in the root of the top DAG and those stored in other vertices. We represent them differently.

First, we explain the data structure for storing complement edges in the root of the top DAG. Let E_{root} be the set of all complement edges stored in the root. We sort edges of E_{root} in the preorder of their starting point. Orders between edges with the same starting point are arbitrary.

For complement edges stored in the root, we store the preorder ids of their starting point using a bit vector B_{src_root}, the preorder ids of their ending point using an array *dst_root*, and edge labels 0/1 using an array *type_root*. The cluster corresponding to the root of the top DAG is the spanning tree of the ZDD. For each node v of the spanning tree, we represent the number of complement edges in E_{root} whose starting point is v, using a unary code. Unary code is a method of representing a natural number by the length of continuous 1's. We concatenate and store them in preorder in the bit vector B_{src_root}. For edges in E_{root} sorted in preorder of the starting points, we store the preorder id of the ending point of the i-th edge in $\mathit{dst\_root}[i]$, and set $\mathit{type\_root}[i] = 0$ if the i-th edge is a 0-edge, and set $\mathit{type\_root}[i] = 1$ otherwise.

Next, we explain the data structure for storing complement edges in vertices other than the root. Let E_{in} be the set of those edges. We sort the edges as follows.

1. We assign each edge of E_in to a cluster such that it contains both its starting and ending points. Then, we divide E_in into several groups, each corresponding to a certain cluster. We sort these groups by the preorder id of the top tree vertex corresponding to the cluster.
2. Inside each cluster $cl(x)$, we sort the edges of E_{in} in preorder of starting points of the edges. For edges with the same starting point, their order is arbitrary.

We store the sorted edges of E_{in} using a bit vector B_{edge} and three arrays *src_in*, *dst_in*, and *type_in*. The bit vector B_{edge} stores the numbers of complement edges in vertices of T' by unary codes. The arrays *src_in*, *dst_in*, and *type_in* are defined as: $\mathit{src\_in}[i] = (\text{the local preorder id of the starting point of the } i\text{-th edge inside the cluster})$, $\mathit{dst\_in}[i] = (\text{the local preorder id of the ending point of the } i\text{-th edge inside the cluster})$, $\mathit{type\_in}[i] = 0$ if the i-th edge is a 0-edge, and $\mathit{type\_in}[i] = 1$ otherwise.

Table 1 summarizes the components of the top ZDD.

Table 1. Components of the top ZDD.

bp	BP sequence representing the structure of T'
B_{dummy}	bit vector showing i-th leaf is a dummy vertex or not
$clsize$	array storing cumulative sum of cluster sizes of the first to the i-th dummy leaves
$label\_span$	array storing differences of labels of ending points of i-th non-dummy leaf
$type\_span$	array showing the edge corresponding to the i-th non-dummy leaf is 0-edge or not
B_H	bit vector showing i-th internal vertex is a vertical merge or not
$preorder\_diff$	array storing differences of preorder ids between the top and the bottom boundary nodes of the vertex corresponding to i-th vertical merge
$label\_diff$	array storing differences of labels between the top and the bottom boundary nodes of the vertex corresponding to i-th vertical merge
B_{src_root}	bit vector storing in unary codes the number of complement edges from each vertex
dst_root	array storing preorder ids of ending points of the i-th complement edge stored in root
$type\_root$	array showing the i-th complement edge stored in the root is a 0-edge or not
B_{edge}	bit vector storing in unary codes the number of complement edges from each vertex stored in the root
src_in	array storing local preorder ids of starting points of i-th complement edge stored in non-root
dst_in	array storing local preorder ids of ending points of i-th complement edge stored in non-root
$type\_in$	array showing the i-th complement edge stored in non-root is 0-edge or not

4.2. Size of Top ZDDs

The size of top ZDDs heavily depends on not only the number of vertices in the spanning tree after top DAG compression but also the number of complement edges for which we store some information. As a result, the size of top ZDDs becomes small if the number of nodes is reduced by top DAG compression and many common complement edges are shared.

In the best case, top ZDDs are exponentially smaller than ZDDs.

Theorem 5. *There exists a ZDD with n nodes to which the corresponding top ZDD has* $O(\log n)$ *vertices.*

Proof. A ZDD storing a power set with $n = 2^m$ elements satisfies the claim. Figure 10 shows this ZDD and top ZDD. A ZDD representing a power set have a linear chain-like shape because no matter how we traverse 0-edges or 1-edges, we will always end up at the $\top$ terminal. The spanning tree of the ZDD is a path consisting of 2^m many 0-edges. Its top tree has a leaf corresponding to a 0-edge of length 1, and internal vertices form a complete binary tree with height m. If we apply DAG compression to this top tree, we obtain the DAG of length m as shown in Figure 10. Sharing complement edges also works very well. The k-th vertex below representing vertical merge stores a 1-edge connecting a node with local preorder id $2^{k-1}+1$ inside a cluster and a node with local preorder id $2^{k-1}+2$. The root of the top DAG stores 0-edge and 1-edge to the terminal $\top$. Because the height of the top DAG is $O(\log n)$, the claim holds. □

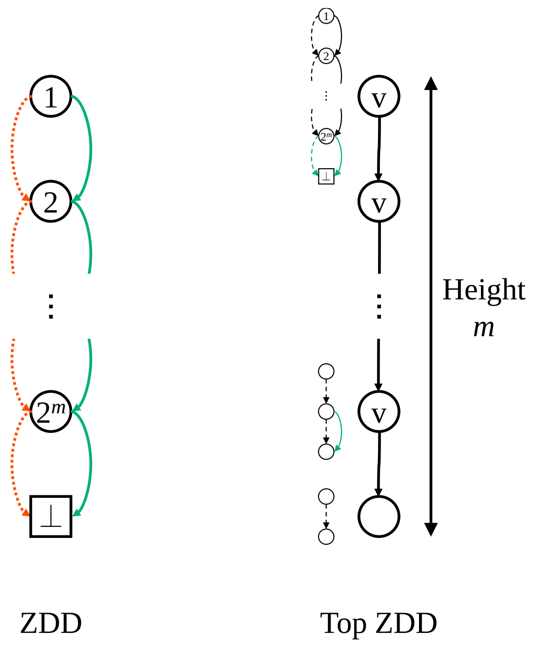

Figure 10. A top ZDD with $O(\log n)$ vertices, where $n = 2^m$. We omitted the middle of the graph by dots, but the height differs exponentially. In this ZDD, the spanning tree is a path consisting only of 0-edges, represented in red. The complement edges are indicated in green. In the top ZDD, the cluster corresponding to each node is shown in a smaller size next to it.

4.3. Operations on Top ZDDs

We give algorithms for supporting operations on the original ZDD using the top ZDD. We consider the following three basic operations. We identify each node x of the ZDD by its preorder id in the spanning tree T used to construct the top ZDD. An example of ZDD nodal numbering based on this rule is shown in the upper left corner of Figure 9.

- $\ell(x)$: returns the label of a branching node x.
- $zero(x)$: returns the preorder id of the node pointed to by the 0-edge of x or returns $\perp$ or $\top$ if the node is a terminal.
- $one(x)$: returns the preorder id of the node pointed to by the 1-edge of x or returns $\perp$ or $\top$ if the node is a terminal.

We show $\ell(x)$ is done in $O(\log n)$ time and other operations are done in $O(\log^2 n)$ time where n is the number of nodes of the ZDD. Below we denote the vertex of T' stored in the top ZDD with preorder id x by "vertex x of T'".

First we explain how to compute $\ell(x)$ in $O(\log n)$ time. We can compute $\ell(x)$ recursively using an algorithm similar to those on the top DAG. A difference is that in Section 3.2.2 we assumed that each vertex of the top DAG stores the cluster size, while in the top ZDD it is not stored to reduce the space requirement. Therefore we have to compute it using the information in Table 1.

To work the recursive computation, we need to compute the cluster size $size(x')$ represented by vertex x' of T' efficiently. We can compute $size(x')$ by the number of non-dummy leaves in the subtree of T' rooted at x', and the sizes of the clusters corresponding to dummy leaves in the subtree rooted at x'. If we merge two clusters of size a and b, the resulting cluster has size $a + b - 1$. Therefore if we merge k clusters whose total size is S, the resulting cluster after $k - 1$ merges has size $S - k + 1$. These values can be computed from the BP sequence bp of T', the array $clsize$, and the bit vector B_{dummy}. By using bp, we can compute the interval $[l, r]$ of leaf ranks in the subtree rooted at x'. Then, using B_{dummy}, we can find the number c of non-dummy leaves and the interval $[l', r']$ of non-dummy leaf ranks, in the subtree of x'. Because *clsize* is the array for storing cumulative sums of cluster sizes for dummy leaves, the summation of sizes of clusters corresponding to l'-th to r'-th dummy leaves is obtained from $clsize[r'] - clsize[l' - 1]$. Since the size of a cluster for a non-dummy leaf is always 2, the summation of cluster sizes for non-dummy leaves is also obtained. Algorithm 2 gives a pseudo-code for computing $size(x')$. This can be done in constant time.

Algorithm 2 $size(x')$: the size of the cluster corresponding to vertex x' of T'.

```
Input: Preorder $x'$
Output: The size of the cluster for $x'$
1: $l \leftarrow leaf\_rank(leftmost\_leaf(x))$
2: $r \leftarrow leaf\_rank(rightmost\_leaf(x))$
3: $l' \leftarrow rank_1(B_{dummy}, l - 1) + 1$
4: $r' \leftarrow rank_1(B_{dummy}, r)$
5: $k \leftarrow r - l + 1$
6: $c \leftarrow (r - l + 1) - (r' - l')$
7: if $l' = 0$ then
8:     return $clsize[r'] + 2c - k$
9: else
10:    return $clsize[r'] - clsize[l' - 1] + 2c - k$
```

Using the function $size(x')$, we can compute a recursive function similar to Algorithm 1. Instead of $D(\cdot)$ in Algorithm 1, we use *preorder_diff*. When we arrive at a dummy leaf, we use a value in *dst_dummy* to move to the corresponding internal vertex of T' and restart the recursive computation. Then for the vertex of the original ZDD whose preorder id in T is x, we can obtain the leaf of T' corresponding to the cluster of a single edge containing x.

To compute $\ell(x)$, we traverse the path from the root of T' to the leaf corresponding to the cluster containing x. First we set $s = 1$. During the traversal, if the current vertex is for vertical merge and the next vertex is its right child, that is, the next cluster is in the bottom, we add the *label_diff* value of the top cluster to s. The index of *label_diff* is computed from B_H and bp. When we reach the leaf p' of T', if x is its top boundary node, it holds $\ell(x) = s$, otherwise, let $k = leaf\_rank(p')$, then we obtain $\ell(x) = s + label\_span[k - rank_1(B_{dummy}, k)]$. Because each operation is done in constant time and the height of the top DAG is $O(\log n)$, $\ell(x)$ is computed in $O(\log n)$ time.

Next, we show how to compute $y = zero(x)$. We can compute $one(x)$ in a similar way. We do a recursive computation as operations on top DAG, a difference is how to process complement edges. There are two cases: if the 0-edge from x is in the spanning tree or not. If the 0-edge from x is in the spanning tree, the edge is stored in a cluster with a single edge (x, y). The top boundary node of such a cluster is x. Therefore we search clusters whose top boundary node is x. If the 0-edge from x is not in the spanning tree, it is a complement edge and it is stored in some vertex on the path from a cluster C with a single edge whose bottom boundary node is x to the root. Therefore we search for C.

First, we recursively find a non-dummy leaf of T' whose top boundary node is x. During this process, if there is a vertex whose top boundary is x and its cluster contains more than one edge and corresponds to horizontal merge, we move to the left child, because the 0-edge from x must exist in the left cluster. If we find a non-dummy leaf of T' which corresponds to a cluster with a single edge and its top boundary node is x, its bottom boundary node is $y = zero(x)$. We climb up the tree until the root to compute the global preorder id of y. If there does not exist such a leaf, the 0-edge from x is not in the spanning tree. We find a cluster with a single edge whose bottom boundary node is x. From the definition of the top ZDD, the 0-edge from x is stored in some vertices visited during the traversal. Because complement edges stored in a cluster are sorted in local preorder ids inside the cluster of starting points, we can check if there exists a 0-edge whose starting point is x in $O(\log n)$ time. If it exists, we obtain the local preorder id of y inside the cluster. By going back to the root, we obtain the global preorder id of y. Note that complement edges for all clusters are stored in one array, and therefore we need to obtain the interval of indices of the array corresponding to a cluster. This can be done using B_{edge}. In the worst case, we perform a binary search in each cluster on the search path. Therefore the time complexity of $zero(x)$ is $O(\log^2 n)$.

5. Experimental Comparison

We compare our top ZDD with existing data structures. We implemented top ZDD with C++ and measured the required space for storing the data structure. For comparison, we used the following three data structures.

- top ZDD (proposed): we measured the space for storing the data structures in Table 1.
- DenseZDD [11]: data structures for representing ZDDs using succinct data structures. Two data structures are proposed; one support constant time queries and the other has $O(\log n)$ time complexity. We used the latter that uses less space.
- a standard ZDD: a data structure which naively represents ZDDs. We store for each node its label and two pointers corresponding to a 0-edge and a 1-edge. Space is $2n\lfloor \log n \rfloor + n\lfloor \log c \rfloor$ bits where n is the number of nodes of a ZDD and c is the size of the universe of a set family.

We constructed ZDDs of the following set families.

- The power set of a set $\{1, \dots, A\}$ with A elements.
- For the set $\{1, \dots, A\}$ with A elements, the family of all the set S satisfying $(\text{The maximum value of } S) - (\text{The minimum value of } S) \leq B$.
- For the set $\{1, \dots, A\}$ with A elements, the family of all the sets with cardinality at most B.
- Knapsack set families with random weights. That is, for i-th element in a set $(1 \leq i \leq A)$, we define its weight w_i as a uniformly random integer in $[1, W]$, then sort the elements in decreasing order of weights, and construct a set family consisting of all sets with weight at most C. Because of the randomness of the input, we ran the experiment 1024 times for each setting and calculate the t-value and p-value between top ZDD and DenseZDD.
- The family of edge sets which are a matching of a given graph. As for graphs, we used the 8×8 grid graph, the complete graph with 12 vertices K_{12}, and a real communication network *"Interoute"*.
- Set families of frequent itemsets.
- Families of edge sets which are paths from the bottom left vertex to the top-right vertex in $n \times n$ grid graph, for $n = 6, 7, 8, 9$.
- Families of solutions of the n-queen problem (The n-queen problem is to find an arrangement of n queens on the $n \times n$ chessboard such that no two queens threaten each other. Here, all possible arrangements are stored in a ZDD), for $n = 11, 12, 13$.

We used several values for the parameters A, B, C, W. The results are shown in Tables 2–9. The unit of size is a byte.

Table 2. The power set of $\{1, \dots, A\}$.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD** $(2n\lfloor \log n \rfloor + n\lfloor \log c \rfloor)/8$
$A = 1000$	**2297**	4185	3750
A = 50,000	**2507**	178,764	300,000

Table 3. For the set $\{1, \ldots, A\}$ with A elements, the family of all the set S satisfying (The maximum value of S) $-$ (The minimum value of S) $\leq B$.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD $(2n\lfloor\log n\rfloor + n\lfloor\log c\rfloor)/8$**
$A = 500, B = 250$	**2471**	227,798	321,594
$A = 1000, B = 500$	**2551**	321,594	1,440,375

Table 4. For the set $\{1, \ldots, A\}$ with A elements, the family of all the sets with cardinality at most B.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD $(2n\lfloor\log n\rfloor + n\lfloor\log c\rfloor)/8$**
$A = 100, B = 50$	**3863**	9544	9882
$A = 400, B = 200$	**13,654**	146,550	206,025
$A = 1000, B = 500$	**43,191**	966,519	1,440,375

Table 5. Knapsack set families with random weights. A is the number of elements, W is the maximum weight of an element, C is the capacity of the knapsack.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD**	**t**	**p**
$A = 100, W = 1000,$ $C = 10,000$	**1,630,136**	1,736,228	2,453,695	49	0
$A = 200, W = 100,$ $C = 5000$	**976,577**	1,409,626	2,026,484	175	0
$A = 1000, W = 100,$ $C = 1000$	**2,078,731**	2,919,211	4,475,977	1124	0
$A = 5000, W = 100,$ $C = 200$	**1,138,778**	1,731,654	2,869,185	1024	0
$A = 1000, W = 10,$ $C = 1000$	**1,404,838**	2,627,430	4,003,302	951	0

Table 6. The family of edge sets which are matching of a given graph.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD $(2n\lfloor\log n\rfloor + n\lfloor\log c\rfloor)/8$**
8×8 grid	**12,246**	16,150	18,014
complete graph K_{12}	23,078	**16,304**	25,340
"Interoute"	**30,844**	39,831	50,144

Table 7. Set families of frequent item sets.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD $(2n\lfloor\log n\rfloor + n\lfloor\log c\rfloor)/8$**
"mushroom" $(p = 0.001)$	104,774	**91,757**	123,576
"retail" $(p = 0.00025)$	**59,894**	65,219	62,766
"T40I10D100K" $(p = 0.005)$	**177,517**	188,400	248,656

Table 8. Families of paths in $n \times n$ grid graph.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD** $(2n\lfloor \log n \rfloor + n\lfloor \log c \rfloor)/8$
$n = 6$	**17,194**	28,593	37,441
$n = 7$	**49,770**	107,529	143,037
$n = 8$	**157,103**	401,251	569,908
$n = 9$	**503,265**	1,465,984	2,141,955

Table 9. Families of solutions of the n-queen problem.

	Top ZDD	**DenseZDD**	**Theoretical Size of ZDD** $(2n\lfloor \log n \rfloor + n\lfloor \log c \rfloor)/8$
$n = 11$	40,792	**35,101**	45,950
$n = 12$	183,443	**167,259**	229,165
$n = 13$	866,749	**799,524**	1,126,295

We found that for all data sets, the top ZDD uses less space than the theoretical size of the standard ZDD. We also confirmed that the data sets in Tables 2–4 can be compressed very well by top ZDDs. Table 5 shows the results on the sets of solutions of knapsack problems. Top ZDD uses less space than DenseZDD in all cases, and in some cases, the memory usage of top ZDD is almost half that of DenseZDD. Tables 6 and 7 show the results for families of matching in a graph and frequent itemsets, respectively. There are a few cases where the DenseZDD uses less space than the top ZDD.

The results above are for monotone set families, that is, any subset of the set of the family also exists in the family. Tables 8 and 9 show results on non-monotone set families. For the set of edges on the path from the bottom left corner to the top right corner of an $n \times n$ grid graph, the top ZDD uses less space than the DenseZDD, and for $n = 9$, the top ZDD uses about $1/3$ the memory of DenseZDD. On the other hand, for the sets of all the solutions of the n-queen problem, the top ZDD uses about 10% more space than the DenseZDD. From these experiments, we confirmed that top ZDD uses less space than DenseZDD for many set families.

Next, we show the construction time and edge traversal time of the top ZDD and the DenseZDD in Tables 10–17. For edge traversal time, we traversed from the root of a ZDD towards terminals by randomly choosing 0- or 1-edge 65,536 times, and took the average. When we arrived at a terminal, we restarted from the root.

Table 10. The power set of $\{1, \ldots, A\}$.

	Construction Time (s)		**Traversal Time (µs)**	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$A = 1000$	0.006	0.004	13.546	0.458
$A = 50{,}000$	0.217	0.116	11.768	0.198

Table 11. For the set $\{1, \ldots, A\}$ with A elements, the family of all the set S satisfying (The maximum value of S) $-$ (The minimum value of S) $\leq B$.

	Construction Time (s)		**Traversal Time (µs)**	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$A = 500, B = 250$	0.264	0.078	9.082	0.244
$A = 1000, B = 500$	0.776	0.412	10.419	0.229

Table 12. For the set $\{1, \ldots, A\}$ with A elements, the family of all the sets with cardinality at most B.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$A = 100, B = 50$	0.011	0.006	10.892	0.320
$A = 400, B = 200$	0.269	0.123	16.019	0.534
$A = 1000, B = 500$	2.013	0.878	20.101	0.412

Table 13. Knapsack set families with random weights. A is the number of elements, W is the maximum weight of an element, C is the capacity of the knapsack. In this experiment, the t-values ranged from 630 to 915 for all settings, and the p-values were 0.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$A = 100, W = 1000, C = 10{,}000$	2.974	1.210	16.716	0.259
$A = 200, W = 100, C = 5000$	2.033	1.019	23.215	0.290
$A = 1000, W = 100, C = 1000$	7.010	1.481	21.698	0.534
$A = 5000, W = 100, C = 200$	2.084	0.954	7.365	0.519
$A = 1000, W = 10, C = 1000$	2.597	1.712	14.127	0.244

Table 14. The family of edge sets which are matching of a given graph.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
8×8 grid	0.030	0.020	11.678	1.053
complete graph K_{12}	0.019	0.009	14.864	0.290
"*Interoute*"	0.028	0.016	15.588	0.397

Table 15. Set families of frequent item sets.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
"*mushroom*" $(p = 0.001)$	0.093	0.037	14.100	0.198
"*retail*" $(p = 0.00025)$	0.099	0.134	12.857	0.702
"*T40I10D100K*" $(p = 0.005)$	0.198	0.117	13.788	0.183

Table 16. Families of paths in $n \times n$ grid graph.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$n = 6$	0.022	0.011	15.491	0.793
$n = 7$	0.082	0.036	12.039	1.022
$n = 8$	0.536	0.153	12.229	1.144
$n = 9$	1.821	0.944	14.233	1.404

Table 17. Families of solutions of the n-queen problem.

	Construction Time (s)		Traversal Time (µs)	
	Top ZDD	**DenseZDD**	**Top ZDD**	**DenseZDD**
$n = 11$	0.038	0.015	17.184	0.778
$n = 12$	0.335	0.065	21.581	0.900
$n = 13$	1.722	0.419	20.173	1.099

In almost all cases, the Top ZDD construction took 1.5 to 2.5 times longer than the DenseZDD construction. The first data in Table 11 and the third data in Table 13 took particularly a long time, suggesting that DenseZDD is faster for families of sets where the size of the included sets is small compared to the total number of variables. The same reason is expected for the overall slowness of Top ZDD for n-queen problems. The second data of Table 15 is the only case that Top ZDD is constructed faster than DenseZDD. This is probably because this data is hardly compressed and DenseZDD is worse than the theoretical size of ordinary ZDD. In terms of traversal time, Top ZDD is 10 to 80 times slower than DenseZDD, which is a natural result since traversal time in DenseZDD is $O(\log n)$ time, while in Top ZDD it is $O(\log^2 n)$ time. In addition, Top ZDD uses more succinct data structures for computation than DenseZDD, so it takes a larger constant coefficient. Basically, in DenseZDD, the traversal time increases with the number of nodes in the input ZDD, but this is not the case in Top ZDD. For example, there is a large variation in the ratio of the traversal speed of Top ZDD to that of DenseZDD in Tables 13–15. The reason for this is that when we execute traverse, Top ZDD performs recursive operations with a maximum $O(\log n)$ depth $O(\log n)$ times, but the actual number of times it is called depends on each entity.

6. Concluding Remarks

We have proposed top ZDD to compress a ZDD by regarding it as a DAG. We compress a spanning tree of a ZDD by the top DAG compression and compress other edges by sharing them as much as possible. We showed that the size of a top ZDD can be logarithmic of that of the standard ZDD. We also showed that navigational operations on a top ZDD are done in time polylogarithmic to the size of the original ZDD. Experimental results show that the top ZDD always uses less space than the standard ZDD, and uses less space than the DenseZDD for most of the data. In previous work on decision diagrams, it was not possible to share and compress substructures of different heights even if they were similar. In addition, the existing methods for top trees could not handle DAGs. In trees, it is easy to number nodes linearly and to divide the whole tree, but this is not the case in DAGs. These are the main reasons for the novelty of this research.

Future work will be as follows. First, in the current construction algorithm, we create a spanning tree of ZDD by a depth-first search, but this may not produce the smallest top ZDD. For example, if we choose all 0-edges, we obtain a spanning tree whose root is the terminal $\top$, and this might be better. Second, in this paper, we considered only traversal operations and did not give advanced operations such as choosing the best solution among all feasible solutions based on an objective function. Third, we will apply our method to other methods that use ZDDs. It would be useful to compare the performance of Top ZDD and DenseZDD in a real case study to deepen our knowledge. For example, we can make the ZDDs smaller that represents Fault Trees for reliability assessment of systems [23]. Lastly, we considered only compressing ZDDs, but our compression algorithm can be used for compressing any DAG. We will find applications of our compression scheme.

Author Contributions: Conceptualization, K.M. and K.S.; methodology, K.M., S.D. and K.S.; software, K.M.; validation, K.M.; formal analysis, K.M.; investigation, K.M.; resources, K.M. and S.D.; data curation, K.M.; writing—original draft preparation, K.M.; writing—review and editing, S.D. and K.S.; visualization, K.M.; supervision, K.S.; project administration, K.S.; funding acquisition, K.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Minato, S. Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In Proceedings of the 30th International Design Automation Conference, Dallas, TX, USA, 14–18 June 1993; pp. 272–277. [CrossRef]
2. Bryant, R.E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* **1986**, *35*, 677–691. [CrossRef]
3. Coudert, O. Solving Graph Optimization Problems with ZBDDs. In Proceedings of the European Design and Test Conference, Paris , France, 17–20 March 1997; pp. 224–228. [CrossRef]
4. Okuno, H.G.; Minato, S.; Isozaki, H. On the properties of combination set operations. *Inf. Process. Lett.* **1998**, *66*, 195–199. [CrossRef]
5. Loekito, E.; Bailey, J. Fast Mining of High Dimensional Expressive Contrast Patterns Using Zero-Suppressed Binary Decision Diagrams. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; pp. 307–316. [CrossRef]
6. Minato, S.; Arimura, H. Frequent Pattern Mining and Knowledge Indexing Based on Zero-Suppressed BDDs. In Proceedings of the 5th International Workshop on Knowledge Discovery in Inductive Databases, Berlin, Germany, 18 September 2006; pp. 152–169.
7. Minato, S.; Uno, T.; Arimura, H. LCM over ZBDDs: Fast Generation of Very Large-Scale Frequent Itemsets Using a Compact Graph-Based Representation. In Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Osaka, Japan, 20–23 May 2008; pp. 234–246.
8. Ishihata, M.; Kameya, Y.; Sato, T.; Minato, S. Propositionalizing the EM algorithm by BDDs. In Proceedings of the 18th International Conference on Inductive Logic Programming, Prague, Czech Republic, 10–12 September 2008; pp. 44–49.
9. Minato, S.; Satoh, K.; Sato, T. Compiling Bayesian Networks by Symbolic Probability Calculation Based on Zero-suppressed BDDs. In Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, 6–12 January 2007; pp. 2550–2555.
10. Sakurai, Y.; Ueda, S.; Iwasaki, A.; Minato, S.; Yokoo, M. A Compact Representation Scheme of Coalitional Games Based on Multi-Terminal Zero-Suppressed Binary Decision Diagrams. In Proceedings of the 14th International Conference on Principles and Practice of Multi-Agent Systems, Wollongong, Australia, 16–18 November 2011; pp. 4–18.
11. Denzumi, S.; Kawahara, J.; Tsuda, K.; Arimura, H.; Minato, S.; Sadakane, K. DenseZDD: A Compact and Fast Index for Families of Sets. *Algorithms* **2018**, *11*, 128. [CrossRef]
12. Bille, P.; Gørtz, I.L.; M.Landau, G.; Weimann, O. Tree Compression with Top Trees. In Proceedings of the Automata, Languages, and Programming—40th International Colloquium Part I, Riga, Latvia, 8–12 July 2013; pp. 160–171.
13. Matsuda, K.; Denzumi, S.; Sadakane, K. Storing Set Families More Compactly with Top ZDDs. In Proceedings of the 18th International Symposium on Experimental Algorithms, Catania, Italy, 16–18 June 2020; Volume 160, pp. 6:1–6:13. [CrossRef]
14. Navarro, G.; Sadakane, K. Fully Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms* **2014**, *10*, 1–39. [CrossRef]
15. Raman, R.; Raman, V.; Satti, S.R. Succinct Indexable Dictionaries with Applications to Encoding K-Ary Trees, Prefix Sums and Multisets. *ACM Trans. Algorithms* **2007**, *3*, 43. [CrossRef]
16. Grossi, R.; Vitter, J.S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* **2005**, *35*, 378–407. [CrossRef]
17. Minato, S.; Ishiura, N.; Yajima, S. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In Proceedings of the 27th ACM/IEEE Design Automation Conference, Orlando, FL, USA, 24–28 June 1990; pp. 52–57. [CrossRef]
18. Buneman, P.; Grohe, M.; Koch, C. Path Queries on Compressed XML. In Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 9–12 September 2003; pp. 141–152.
19. Downey, P.J.; Sethi, R.; Tarjan, R.E. Variations on the Common Subexpression Problem. *J. ACM* **1980**, *27*, 758–771. [CrossRef]
20. Frick, M.; Grohe, M.; Koch, C. Query evaluation on compressed trees. In Proceedings of the 18th Annual IEEE Symposium of Logic in Computer Science, Vienna, Austria, 14–18 July 2003; pp. 188–197. [CrossRef]
21. Alstrup, S.; Holm, J.; de Lichtenberg, K.; Thorup, M. Maintaining Information in Fully Dynamic Trees with Top Trees. *ACM Trans. Algorithms* **2005**, *1*, 243–264. [CrossRef]

22. Okanohara, D.; Sadakane, K. Practical Entropy-Compressed Rank/Select Dictionary. In Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments, New Orleans, LA, USA, 6 January 2007. [CrossRef]
23. xing Duan, R.; lin Zhou, H. A New Fault Diagnosis Method Based on Fault Tree and Bayesian Networks. *Energy Procedia* **2012**, *17*, 1376–1382. [CrossRef]