Storing Set Families More Compactly with Top ZDDs

Kotaro Matsuda
Graduate School of Information Science and Technology, The University of Tokyo, Japan
kotaro_matsuda@mist.i.u-tokyo.ac.jp

Shuhei Denzumi
Graduate School of Information Science and Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Kunihiko Sadakane
Graduate School of Information Science and Technology, The University of Tokyo, Japan
sada@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 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 size than ZDDs for real data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases top tree, Zero-suppressed Decision Diagram, space-efficient data structure

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.6

Related Version A full version of the paper is available at [9], `https://arxiv.org/abs/2004.04586`.

1 Introduction

Zero-suppressed Binary Decision Diagrams (ZDDs) [10] are data structures which are derived from Binary Decision Diagrams (BDDs) [3] and which represent a family of sets (combinatorial sets) in a compressed form by Directed Acyclic Graphs (DAGs). ZDDs are data structures specialized for processing set families and compress sparse set families well. ZDDs support binary operations between two set families in time polynomial to the ZDD size. Due to these advantages, ZDDs are used for combinatorial optimization and enumeration.

Though ZDDs can store set families compactly, their size may grow for some set families, and we need further compression. DenseZDDs [5] 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 height direction, they cannot be shared.

For not DAGs but trees, there exists a data structure called top DAG compression [2], which can capture repetitive structures in height direction. We extend it for DAGs and apply to compress ZDDs which support the operations on compressed ZDDs.

1.1 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 number of 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 [12] and for bitvectors [14, 8].

We show experimental results on the size and query time of our top ZDDs and existing data structures. The results show that the top ZDDs use less space for most of 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. 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) [10] 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 two types of 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 $v_0 = \mathit{zero}(v)$ $(v_1 = \mathit{one}(v))$. If for any branching node v it holds $\ell(v) < \ell(v_0)$ and $\ell(v) < \ell(v_1)$, 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 $L_1, \ldots, L_{c+1}$ $(i = 1, \ldots, c + 1)$ according to the labels of the nodes. 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.

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 $v_0 = \mathit{zero}(v)$, $v_1 = \mathit{one}(v)$. Then the set family $\mathcal{F}_v$ represented by v is defined 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 $\mathcal{F}_G$.

All the paths from the root to the terminal $\top$ on ZDD G have one-to-one correspondence to all the sets $S = \{a_1, \ldots, a_s\}$ 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 the branching node corresponding to $a_i \in S$ does not exist.

2.2 Succinct data structures

Succinct data structures are data structures whose size match the information theoretic lower bound. 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 the following data structures.

2.2.1 Bitvectors

Bitvectors 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 bitvector. On this bitvector 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 2** ([14]). *For a bitvector 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 bitvector of length n with m ones. For a sparse bitvector, namely, the one with $m = \mathrm{o}(n/ \log n)$, we can obtain a more space-efficient data structure.

▶ **Theorem 3** ([8]). *For a bitvector B of length $n = 2^w$ with m ones, $select_1(B, i)$ is computed in constant time on the word-RAM with word length $\Omega(\log n)$ using a $m(2 + w - \lfloor \log_2 n \rfloor) + \mathrm{O}(m \log\log m/ \log m)$-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. An information-theoretic lower bound of such trees is $2n - \Theta(\log n)$ bits. We want to support the following operations: (1) $parent(x)$: returns the parent of node x, (2) $firstchild(x), lastchild(x)$: returns the first/last child of node x, (3) $nextsibling(x), prevsibling(x)$: returns the next/previous sibling of node x (4) $isleaf(x)$: returns if node x is a leaf or not, (5) $preorder\_rank(x)$: returns the preorder of node x, (6) $preorder\_select(i)$: returns the node with preorder i, (7) $leaf\_rank(x)$: returns the number of leaves whose preorders are smaller than that of node x, (8) $leaf\_select(i)$: returns the i-th leaf in preorder, (9) $depth(x)$: returns the depth of node x, that is, the distance from the root to x, (10) $subtreesize(x)$: returns the number of nodes in the subtree rooted at node x, (11) $lca(x, y)$: returns the lowest common ancestor (LCA) between nodes x and y.

▶ **Theorem 4** ([12]). *On the word-RAM with word length $\Omega(\log n)$, the above operations are done in constant time using a $2n + \mathrm{o}(n)$-bits data structure.*

We call this the BP representation in this paper.

2.3 DenseZDD

A DenseZDD [5] is a static representation of a ZDD with attricuted edges [11] by using some succinct data structures. In comparison to the ordinary ZDD, a DenseZDD provides a much faster membership operation and less memory usage for most of 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 internal node v for fast traversal. The spanning tree consisting of all reversed 0-edges is represented by straight forward BP. The DenseZDD is a combination of this BP and other succinct data structures that represent remaining information of the given ZDD.

3 Top Tree and Top DAG

We explain top DAG compression [2] to compress labeled rooted trees.

Top DAG compression is a compression scheme for labeled rooted trees by converting the input tree into top tree [1] and then compress it by DAG compression [4, 6, 7]. DAG compression is a scheme to represent a labeled rooted tree by a smaller DAG obtained by merging identical subtrees of the tree. Top DAG compression can compress repeated sub-structures (not only subtrees). For example, a path of length n with identical labels can be represented by a top DAG with $O(\log n)$ nodes. Also, for a tree with n nodes, accessing a node label, computing the subtree size, and tree navigational operations such as first child and parent are done in $O(\log n)$ time. Here we explain the top tree and its greedy construction algorithm. We also explain operations on top DAGs.

The top tree [1] for a labeled rooted tree T is a binary tree $\mathcal{T}$ representing the merging process of clusters of T defined as follows. 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 as the node set of the new cluster. There are five types of merges, as shown in Figure 1.

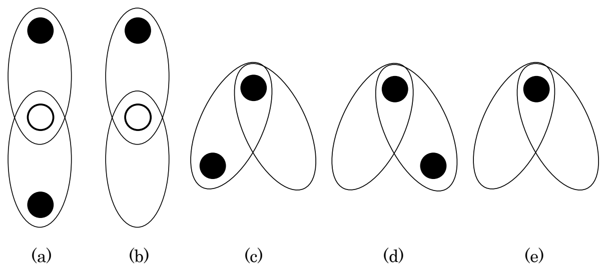

Figure 1 Types of merging of clusters. Ellipses are clusters before merge, black circles are boundary nodes of new clusters, and white circles are not boundary nodes in new clusters.

These five merges are divided into two.

1. (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.
2. (c)(d)(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.

The top tree of the tree T is a binary tree $\mathcal{T}$ satisfying the following conditions.

- 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 1.
- The cluster of the root of the top tree is T itself.

We call the DAG obtained by DAG compression of the top tree $\mathcal{T}$ as top DAG $\mathcal{TD}$, and the operation to compute the top DAG $\mathcal{TD}$ from tree T is called top DAG compression [2].

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 both endpoints of the corresponding edge in T. For an internal vertex of the top tree, its label must have the information about the cluster merge. It is enough to consider three types of merges, not five as in Figure 1. For vertical merges, it is not necessary to store the information that the merged cluster has the 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 $\mathrm{H_L}/\mathrm{H_R}$ if the left/right child cluster has the bottom boundary node, respectively. 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 5** ([2]). *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* $\mathrm{O}(\log n)$.

Consider to support operations on a tree T which is represented by top DAG. From now on, a node x in T stands for the preorder of x in T. By storing additional information to each vertex of the top DAG, many operations can be supported [2]. For example, $\mathit{Access}(x)$ returns the label of x and $\mathit{Decompress}(x)$ returns the subtree $T(x)$ rooted at x. For a tree with n nodes, many operations in [2] are done in $\mathrm{O}(\log n)$ time, and $\mathit{Decompress}(\cdot)$ is done in $\mathrm{O}(\log n + |T(x)|)$ time. A pseudo code of $\mathit{Access}(\cdot)$ is in the full version [9].

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 only compression of ZDDs.

A ZDD $G = (V, E)$ is a directed acyclic graph in which nodes have labels $\ell(\cdot)$ (terminal nodes have $\bot$ and $\top$) and edges have labels 0 or 1. We can regard it as a graph with only edges being labeled. For each edge (u, v) of ZDD G, we define its label as a pair (edge label 0/1, $\ell(u) - \ell(v)$) if v is a branching node, or a pair (edge label 0/1, $\bot/\top$) if v is a terminal node. In practice, we can use $c+1$ instead of $\bot$ and $c+2$ instead of $\top$ for the second element, where $c+1 = \ell(\bot) = \ell(\top)$. 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-labeled trees. The difference from node-labeled trees is only how to store the information for single edge clusters. In top trees, 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 preorders in T. If we say node u, it means the node in T with preorder 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 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 $0/1$, $\ell(u) - \ell(v))$. We decide a vertex to store it 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 preorders 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 $\mathcal{T}D$ by DAG compression by sharing identical clusters. To determine identicalness of two clusters, we compare them together with the information of complement edges in the clusters stored in step 3. Complement edges which do not appear in multiple clusters are moved to the root of T.

Figure 2 shows an example of a top ZDD. The left is the original ZDD and the right is the corresponding top ZDD. Red and green edges show edges in the spanning tree and complement edges, respectively. In this figure we show for each vertex of the top DAG, the corresponding cluster, but they are not stored explicitly.

To achieve small space, it is important to use what data structure for representing each information. For example, we explained that each vertex of the top DAG stores the cluster size etc., this is redundant and the 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 $\mathcal{T}D$.
- Information about each vertex of $\mathcal{T}D$. 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. The root or other vertices of $\mathcal{T}D$ store them.

We show space-efficient data structures for storing these information. In theory, we use the succinct bitvector with constant time rank/select support [14]. In practice, we use the SparseArray [13] to compress a bitvector if the ratio of ones in the bitvector is less than $\frac{1}{4}$,

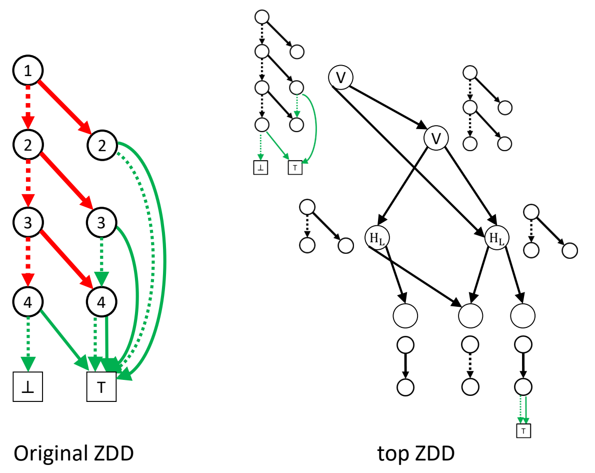

Figure 2 An example of a top ZDD. 0-edges and 1-edges are depicted by dotted and solid lines, respectively. Red edges are spanning tree edges and green edges are complement edges. For each vertex of the top DAG, the corresponding cluster and the information stored in the vertex are shown.

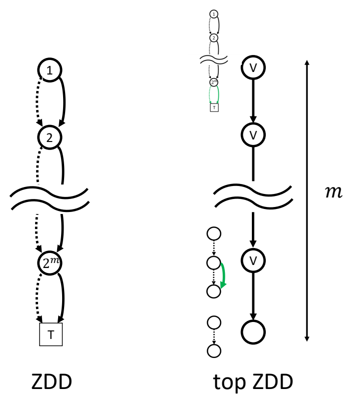

Figure 3 A top ZDD with $O(\log n)$ vertices, where $n = 2^m$.

and use the SparseArray for the bitvector whose 0/1 are flipped if the ratio of zeros 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 internal 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

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_i) 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 of x in T' from which the dummy vertices are removed.

Then we can convert the top DAG to 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 [12]. 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 bitvector. Let m be the number of leaves in T'. We create a bitvector 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 $clsize[k] = \sum_{i=1}^{k}(\text{the number of vertices in the cluster represented by} s_i)$. That is, $clsize[k]$ 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 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 edge corresponding to the i-th leaf, respectively. Then we set $\mathit{label\_span}[i] = \ell(v) - \ell(u)$. The other is 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 bitvector 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 preorders 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 $\mathit{preorder\_diff}[i] =$ (the local preorder of b_i 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: those stored in the root of the top DAG and those stored in other vertices. We represent them in a different way.

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 preorders of their starting point using a bitvector B_{src_root}, the preorders 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 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. We concatenate and store them in preorder in the bitvector B_{src_root}. For edges in E_{root} sorted in preorder of the starting points, we store the preorder 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 divide the edges of E_{in} into groups based on the clusters having the edges. These groups are sorted in preorder of vertices for the clusters.
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 bitvector B_{edge} and three arrays *src_in*, *dst_in*, and *type_in*. The bitvector 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: $\mathit{src\_in}[i] =$ (the local preorder of the starting point of the i-th edge inside the cluster), $\mathit{dst\_in}[i] =$ (the local preorder of the ending point of the i-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.

A top ZDD is composed of the following data structures:

- *bp*: a BP sequence representing the structure of T'
- B_{dummy}: a bitvector showing i-th leaf is a dummy vertex or not
- *clsize*: an array storing cumulative sum of cluster sizes of the first to the i-th dummy leaves
- *label_span*: an array storing differences of labels of ending points of i-th non-dummy leaf
- *type_span*: an array showing the edge corresponding to the i-th non-dummy leaf is 0-edge or not
- B_H: a bitvector showing i-th internal vertex is a vertical merge or not
- *preorder_diff*: an array storing differences of preorders between the top and the bottom boundary nodes of the vertex corresponding to i-th vertical merge
- *label_diff*: an 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}: a bitvector storing in unary codes the number of complement edges from each vertex
- *dst_root*: an array storing preorders of ending points of the i-th complement edge stored in root
- *type_root*: an array showing the i-th complement edge stored in the root is a 0-edge or not
- B_{edge}: a bitvector storing in unary codes the number of complement edges from each vertex stored in the root
- *src_in*: an array storing local preorders of starting points of i-th complement edge stored in non-root
- *dst_in*: an array storing local preorders of ending points of i-th complement edge stored in non-root
- *type_in*: an array showing the i-th complement edge stored in non-root is 0-edge or not

4.2 The 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. Therefore 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 6.** *There exists a ZDD with n nodes to which the corresponding top ZDD has $O(\log n)$ vertices.*

A detailed proof is in the full version [9]. A ZDD for a power set with $n = 2^m$ elements satisfies the claim. Figure 3 shows such a ZDD.

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 a node x of the ZDD with the vertex in the spanning tree T used to create the top ZDD whose preorder is x.

- $\ell(x)$: returns the label of a branching node x.
- $zero(x)$: returns the preorder of x_0, or returns $\perp$ or $\top$ if the node is a terminal.
- $one(x)$: returns the preorder of x_1, 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 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 a similar algorithm to those on the top DAG. A difference is that 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 components of the top ZDD.

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 bitvector 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 $clsize[r'] - clsize[l'-1]$. Because the size of a cluster for a non-dummy leaf is always 2, the summation of cluster sizes for non-dymmy leaves is also obtained. This can be done in constant time. A pseudo code for computing $size(x')$ is in the full version [9].

Using the function $size(x')$, we can compute a recursive function similar to the algorithm of $Access(\cdot)$. Instead of $D(\cdot)$ in Algorithm 1 [9], 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 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 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', 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 $O(1)$ time and the height of the top DAG is $O(\log n)$, $\ell(x)$ is $O(\log n)$ 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 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 preorders 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 of y inside the cluster. By going back to the root, we obtain the global preorder 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 all components of a top ZDD.
- DenseZDD [5]: data structures for representing a ZDD 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: ZDDs that are implemented naively. We store for each node its label and two pointers corresponding to a 0-edge and a 1-edge. The 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 universal set.

We constructed ZDDs for various data with different settings. The results are shown in Table 1. The unit of size is bytes.

We found that for all data sets, the top ZDD uses less space than the naive representation of the standard ZDD. We also confirmed that the data 1, 2, and 3 can be compressed very well by top ZDDs. For any settings on the data 4, the top ZDD uses less space than the DenseZDD, and for some cases the memory usage of the top ZDD is almost $\frac{1}{2}$–$\frac{2}{3}$ of that of the DenseZDD. For the data 5 and 6, There are a few case that the DenseZDD uses less space than the top ZDD.

The results above are for monotone set families, that is, any subset of a set a the family also exists in the family. The data 7 and 8 are non-monotone set families. For the families of paths on $n \times n$ grid graphs, the top ZDD uses less space than the DenseZDD, and for $n = 9$, the top ZDD uses about $\frac{1}{3}$ the memory of DenseZDD. On the other hand, for the data 8, the top ZDD uses about 10 % more space than the DenseZDD. From these experiments we confirmed that the top ZDD uses less space than the DenseZDD for many set families.

The experiments to compare construction time and edge traversal time of top ZDDs and DenseZDDs are conducted in the full version [9]. The results show that DenseZDDs are several times faster on construction and several dozens faster on traversal than top ZDDs. Note that the traversal time is $\Theta(\log n)$ on top ZDDs, but is linear to a query size on DenseZDDs.

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 proved that the size of a top ZDD can be logarithmic of that

Table 1 Results of experiments.

Data	Setting	top ZDD	DenseZDD	ZDD
1. The power set of $\{1,\dots,A\}$	$A = 1000$	**2,297**	4,185	3,750
	$A = 50000$	**2,507**	178,764	300,000
2. For $C = \{1,\dots,A\}$, $\{S \subseteq C \mid \max_{a\in S} a - \min_{a\in S} a \leq B\}$	$A = 500, B = 250$	**2,471**	227,798	321,594
	$A = 1000, B = 500$	**2,551**	321,594	1,440,375
3. For $C = \{1,\dots,A\}$, the family of sets $\{S \subseteq C \mid \lvert S\rvert \leq B\}$	$A = 100, B = 50$	**3,863**	9,544	9,882
	$A = 400, B = 200$	**13,654**	146,550	206,025
	$A = 1000, B = 500$	**43,191**	966,519	1,440,375
4. Knapsack set families with random weights. A is the number of elements, W is the maximum weight of an element, C is the capacity. (Setting is given as (A, W, C).)	$(100, 1000, 10000)$	**1,659,722**	1,730,401	2,444,405
	$(200, 100, 5000)$	**1,032,636**	1,516,840	2,181,688
	$(1000, 100, 1000)$	**2,080,965**	2,929,191	4,491,025
	$(5000, 100, 200)$	**1,135,653**	1,740,841	2,884,279
	$(1000, 10, 1000)$	**1,383,119**	2,618,970	3,990,350
	$(1000, 100, 1000)$	**565,740**	656,728	1,056,907
5. The family of edge sets which are matching of a given graph	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
6. Set families of frequent item sets with a minimum frequency	*mushroom* (0.1%)	104,774	**91,757**	123,576
	retail (0.025%)	**59,894**	65,219	62,766
	T40I10D100K (0.5%)	**177,517**	188,400	248,656
7. Families of edge sets in $n \times n$ grid graph which are paths from the bottom left to the top right	$n = 6$	**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
8. Families of solutions of the n-queen problem	$n = 11$	40,792	**35,101**	45,950
	$n = 12$	183,443	**167,259**	229,165
	$n = 13$	866,749	**799,524**	1,126,295

of the ZDD. We also showed that navigational operations on a top ZDD are done in time polylogarithmic to the original ZDD size. Experimental results show that the top ZDD is always smaller than the ZDD, and uses less space than the DenseZDD for most of the data.

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. Next, 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. 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.

References

1 Stephen Alstrup, Jacom Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005. `doi:10.1145/1103963.1103966`.

2 Philip Bille, Inge Li Gørtz, Gad M.Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015. `doi:10.1016/j.ic.2014.12.012`.

3 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986. `doi:10.1109/TC.1986.1676819`.

4 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 141–152. Morgan Kaufmann, 2003. doi:10.1016/B978-012722442-8/50021-5.

5 Shuhei Denzumi, Jun Kawahara, Koji Tsuda, Hiroki Arimura, Shin ichi Minato, and Kunihiko Sadakane. Densezdd: A compact and fast index for families of sets. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 187—198. Springer Verlag, 2014. doi:10.1007/978-3-319-07959-2_16.

6 Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of ACM*, 27(4):758–771, 1980. doi:10.1145/322217.322228.

7 Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees. In *Proceedings of 18th Annual IEEE Symposium of Logic in Computer Science*, LICS 2003, pages 188–197. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210058.

8 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

9 Kotaro Matsuda, Shuhei Denzumi, and Kunihiko Sadakane. Storing set families more compactly with top zdds, 2020. arXiv:2004.04586.

10 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277. ACM, 1993. doi:10.1145/157485.164890.

11 Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 52–57, 1990. doi:10.1145/123186.123225.

12 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3), 2014. doi:10.1145/2601073.

13 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*, 2007. doi:10.1137/1.9781611972870.6.

14 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43–es, 2007. doi:10.1145/1290672.1290680.