Regular Paper

Tree Decomposition-based Approach for Compiling Independent Sets

Teruji Sugaya[1,a) Masaaki Nishino[2,b) Norihito Yasuda[2,c) Shin-ichi Minato[3,d)

Received: September 28, 2019, Accepted: April 8, 2020

Abstract: Knowledge compilation is a method for compiling a knowledge base into an appropriate data structure, generally called tractable language. Graph substructure plays an important role in knowledge compilation and frontier-based search is known to be an efficient algorithm, in which computation time is bounded by the path-width of a graph. For some limited classes of graph structures, studies have shown that it can be improved and bounded by the branch-width, however, the redesign of an algorithm for other classes does not appear to be straightforward. In this paper, we focus on the similarity between frontier-based search and dynamic programming on tree decomposition. Dynamic programming on tree decomposition has been intensely studied for varieties of problems on counting or optimization of graph substructures. However, to the best of our knowledge, they are rarely applied to knowledge compilation. Then, we show that dynamic programming for finding the size of the maximum independent set can be, by simple replacement, applied to the compilation of independent sets. Furthermore, we empirically show that our method can compile much faster than conventional frontier-based search in some instances, and it becomes several orders of magnitude faster especially when the tree-width is small compared to the path-width.

Keywords: knowledge compilation, frontier-based search, structured Z-d-DNNF, independent set

1. Introduction

Knowledge compilation is a method for compiling a knowledge base into an appropriate data structure that supports transformations or queries such that they can be applied or answered efficiently [1]. The language of data structures is called a *tractable language* in general. In knowledge compilation, the tractability of the language is a result of the compilation and it can be used to process various queries related to a knowledge base, such as optimization, enumeration, and counting. Knowledge compilation has been intensively studied and has applications in various fields. In Broeck et al.'s study [2], a variety of exemplary applications were cited, such as diagnosis [3], [4], planning [5], probabilistic reasoning [6], [7], [8], probabilistic databases [9], [10], [11], [12], first-order probabilistic inference [13], [14], [15], and learning of tractable probabilistic models [16].

The graph substructure is also an important target of knowledge compilation. Graph substructures are subsets of graph nodes and edges that satisfy specific conditions, such as independent set, *s*-*t* path, and matching. Frontier-based search [17] algorithms, which include Knuth's Simpath [18], are efficient for compiling a graph substructure into a tractable language called zero-suppressed binary decision diagrams (ZDDs) [19]. Frontier-based search algorithms can be used to compile large sets of a graph substructure into ZDDs of a considerably smaller size. Once such ZDDs have been constructed, the sets of a graph substructure can be efficiently enumerated or counted using the operations supported by ZDDs. Currently, frontier-based algorithms for several graph substructures, including *s*-*t* paths, Hamiltonian paths, spanning trees, and matching, have been proposed.

It is known that the computation time and size of the obtained ZDDs can be bounded by the path-width of the graph [20]. Studies were conducted with the objective of improving the efficiency of frontier-based search. Nishino et al. [21] and Sugaya et al. [22] redesigned frontier-based search to run on branch decomposition to compile matching [21] or *s*-*t* paths [21], [22]. The size of their output tractable language was then bounded by the branch-width. However, to expand their method such that it can be applied to other types of graph substructures, one must design a suitable algorithm from scratch, which does not appear to be straightforward.

On the other hand, dynamic programming has been intensively studied as an algorithm that uses tree decomposition. Numerous studies have focused on counting or optimizing graph substructures, e.g., the maximum independent set [23], [24], minimum dominating set [25], coloring [26], and minimum Steiner tree [27]. It is known that the tree-width and branch-width are always within a constant factor of each other [28]. However, to the best of our knowledge, their application was limited to counting or optimizing graph substructures, and few approaches have been

1 Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Hokkaido 060–0808, Japan
2 NTT Communication Science Laboratories, NTT Corporation, Keihanna Science City, Kyoto 619–0237, Japan
3 Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
a) sugaya@ist.hokudai.ac.jp
b) nishino.masaaki@lab.ntt.co.jp
c) yasuda.n@lab.ntt.co.jp
d) minato@i.kyoto-u.ac.jp

Table 1 Frequently used notations.

Notation	Description	Notation	Description
$G = (V, E)$	Simple undirected graph	$g \sqcup h$	Represents $\{a \cup b \mid a \in g \text{ and } b \in h\}$
$v_1, v_2, \ldots, u_1, u_2, \ldots, w_1, w_2, \ldots$	Nodes of G (gnodes)	$\alpha, \beta, \ldots$	Nodes of a vtree (vnodes)
$\lvert V\rvert, \lvert E\rvert$	Size of gnodes and edges of G	E_β	Set of items that correspond to the leaves of a sub-tree the root of which is β
V	Gnode set contained in G	e_α	An item that corresponds to a leaf α of a vtree
$\mathcal{T}$	Tree decomposition	$s(i)$	Size of the maximum independent set of G_i
$T(\mathcal{T})$	Set of nodes of a tree decomposition $\mathcal{T}$	$s(i; P_i, Q_i)$	Size of the maximum independent set of G_i that contains the entire gnode set P_i and no member of the gnode set Q_i
B_i	Bag of a tnode $i \in T(\mathcal{T})$		
W	Tree-width	$\mathcal{S}(i)$	Independent sets of G_i
V_i	Union of tnode B_i and its children tnodes	$\mathcal{S}(i; P_i, Q_i)$	Independent sets of G_i that contain the entire gnode set P_i and no member of the gnode set Q_i
G_i	The subgraph induced by V_i.		
d, d_α	Dnode of structured Z-d-DNNF and dnode that identifies the respecting vnode α	$mate[v]$	Mate of gnode v
$\perp, \epsilon, X, \pm X$	Terminal dnode, each representing $\emptyset, \{\emptyset\}, \{X\}$, and $\{\{X\}, \emptyset\}$		
D_α	Dnode set that respects vnode α		
$\langle d\rangle$	A family of sets represented with dnode d		

done for the knowledge compilation [*1].

In this paper, unlike previous studies, we focus on the similarity between frontier-based search and dynamic programming methods that use tree decomposition. We show that the dynamic programming used for finding the size of the maximum independent set in a given graph [23], [24] can be used in the compilation of independent sets by simple replacement. Additionally, we use structured Z-deterministic decomposable negation normal form (Z-d-DNNF) [22] as a tractable language in our method. Previous studies of dynamic programming on tree decomposition, including that of maximum independent set [23], [24], is conducted on a bottom up search dependent on the types of tree decomposition nodes. As we show in Section 3.4, a tractable language Z-d-DNNF is suitable to apply this type of dynamic programming on tree decomposition towards knowledge compilation.

The time complexity of the proposed method is the same as that of the dynamic programming method for computing the size of a maximum independent set. Moreover, the size of the tractable language is bounded by the tree-width of the input graph, whereas that of frontier-based search is bounded by the path-width. In addition, as in the case of frontier-based search, we can enumerate or count the independent sets using an operation supported by structured Z-d-DNNF. Furthermore, we empirically show that our method can compile independent sets of a graph much faster than the conventional frontier-based search in some instances, and it may become several orders of magnitude faster especially when the tree-width is small compared to the path-width.

Organization.

The organization of this paper is as follows. The relevant terms are explained in Section 2. Our proposed method is described in Section 3. Experimental results are provided in Section 4. Related research is described in Section 5. Section 6 concludes the paper and describes areas of future research.

2. Preliminaries

We consider four types of graphs, which are discussed later: input graph, tree decomposition of the input, vtree, and structured Z-d-DNNF. To avoid confusion, we distinguish them by the name and notation of their nodes: the vertices of the input graph are called *gnodes* and are denoted by $u, w, v_1, v_2, \ldots$, the nodes in tree decompositions are called *tnodes* and are denoted by $B, B_1, B_2, \ldots$, vtree nodes are called *vnodes* and are denoted by $\alpha, \beta, \ldots$, and structured Z-d-DNNF nodes are called *dnodes* and are denoted by $d_1, d_2, \ldots$.

For the reader's convenience, the notations frequently used in this paper are listed in **Table 1**.

2.1 Graph and Tree Decomposition

2.1.1 Graph

Let $G = (V, E)$ be a simple undirected connected graph having a gnode set V and an edge set E. $|V|$ and $|E|$ denote the size of the gnodes and that of the edges of the graph G, respectively. An *independent set* is a set of gnodes in a graph, in which no two gnodes are adjacent. A *maximum independent set* is an independent set of the largest size of a graph G.

2.1.2 Tree Decomposition

Intuitively, a *tree decomposition* of a graph G is a means of representing G as a tree-like structure. Formally, it is defined as follows.

Definition 1 (Tree Decomposition)**.** A tree decomposition of a graph G is a pair $(\mathcal{B}, \mathcal{T})$, where $\mathcal{T}$ is a tree, $T(\mathcal{T})$ is the set of tnodes of $\mathcal{T}$, and $\mathcal{B} = (B_i : i \in T(\mathcal{T}))$ is a family of subsets of V that are assigned to each tnode $i \in T(\mathcal{T})$. B is also called a *bag*. The properties of tree decomposition are as follows [30], [31].

(1) $\bigcup(B_i : i \in T(\mathcal{T})) = V$.

(2) For every edge e of G there exists $i \in T(\mathcal{T})$ such that e has both ends in B_i.

(3) For $i, j, k \in T(\mathcal{T})$, if j is on the path of $\mathcal{T}$ between i and k, then

$$B_i \cap B_k \subseteq B_j. \quad (1)$$

The *width* of the tree decomposition is

$$\max_{i \in T(\mathcal{T})} (|B_i| - 1). \quad (2)$$

We say graph G has *tree-width* W if W is the minimum width of

[*1] For the study of dynamic programming on automaton, refer to Amarilli et al.'s paper [29] which is also shown in Section 5.

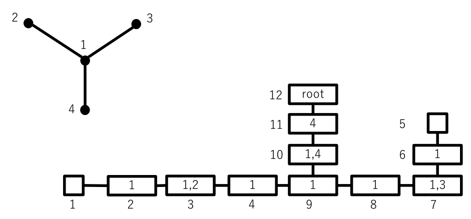

Fig. 1 Example of a graph and its nice tree decomposition.

any tree decomposition of G. When $\mathcal{T}$ of a tree decomposition is a path, the decomposition is called *path decomposition*, and the tree-width derived from the decomposition is called *path-width*.

A tree decomposition is frequently transformed to a *nice tree decomposition* [32] to facilitate the design of dynamic programming.

Definition 2 (Nice Tree Decomposition)**.** A tree decomposition $\mathcal{T}$ is *nice* if there is a root node *root* $\in T(\mathcal{T})$, and each tnode $i \in T(\mathcal{T})$ is one of the following four types:

(1) Leaf: i is a leaf of $\mathcal{T}$ and $|B_i| = 1$.

(2) Introduce: i has one child j such that there is a gnode u with $B_i = B_j \cup \{u\}$, where gnode u is said to be *introduced* in B_i.

(3) Forget: i has one child j such that there is a gnode w with $B_j = B_i \cup \{w\}$, where gnode w is said to be *forgotten* in B_i.

(4) Join: i has two children j and k with $B_i = B_j = B_K$.

From the graph $G(V, E)$ and its tree decomposition with tree-width W, it is known that a nice tree decomposition of tree-width W can be obtained in $O(W^2 \cdot max(|T(\mathcal{T})|, |V|)$ [33]. We show an example of a graph and its nice tree decomposition in **Fig. 1**. For the sake of simplicity, hereafter we represent a (nice) tree decomposition $(\mathcal{B}, \mathcal{T})$ by $\mathcal{T}$.

2.2 Tractable Language

Structured-Z-d-DNNF is a tractable language for hierarchically decomposing given families of sets and representing them with a directed acyclic graph (DAG). In this section, we explain structured-Z-d-DNNF and its related notions.

2.2.1 (X,Y)-Decomposition

(X,Y)-decomposition is a method for decomposing given families of sets into sub-families.

Let f be a family of sets and U be its ground set. Let X, Y be mutually exclusive subsets of U and let $X \cup Y = U$. For $i = 1, \ldots, n$, let $p_i^l(X)$ be a family of sets, whose ground set is X, and let $p_i^r(Y)$ be a family of sets, whose ground set is Y.

The **(X,Y)**-decomposition of a family of sets f is described as shown below; this decomposition is referred to as **(X,Y)**-decomposition.

$$f = [p_1^l(X) \sqcup p_1^r(Y)] \cup \ldots \cup [p_n^l(X) \sqcup p_n^r(Y)]. \tag{3}$$

Here, $\sqcup$ represents the operation *join*, defined as $g \sqcup h = \{a \cup b \mid a \in g \text{ and } b \in h\}$, where g and h are arbitrary families of sets. For example, when $g = \{\{1\}, \{2\}\}$ and $h = \{\{3\}, \{4\}\}$ are given, $g \sqcup h$ is equal to $\{\{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}$.

Any family of sets are **(X,Y)**-decomposable, however in general, the **(X,Y)**-decompositions are not canonical [34]. Additionally, n in Eq. (3) is $O(2^{|X|}2^{|Y|})$ with a naive method. In Section 3.2, we describe how to reduce the number of cases in our method.

2.2.2 Vtree

A *vtree* is a rooted, full, and ordered binary tree. Each node of a vtree is called a *vnode*. In a vtree, each leaf corresponds to a distinct item of the ground set of the families of sets, except the dummy vnode, described below.

Here, the left and right children of the internal vnode α are denoted by α^l and α^r, respectively. The set of items corresponding to the leaves of a vtree rooted at vnode α is E_α, and the item corresponding to a leaf vnode α is e_α. An internal vnode α represents a partition of the set of items E_α into two sets E_{α^l} and E_{α^r}. A vtree is used to recursively **(X,Y)**-decompose a family of sets, starting from the root of the vtree.

In addition, when a vnode α is a dummy, i.e., α has no corresponding item, the corresponding dummy item is represented by $e_\alpha = \phi$. A dummy vnode is necessary for the proposed method, as explained in Section 3. A vtree that includes dummy nodes is called a *dummy-added vtree*.

We show an example of a vtree in Fig. 3. In the figure, the number of each node is the ID of the vnode. Root node 3 corresponds to a **(X,Y)**-decomposition, where $X = \{A, B\}$ and $Y = \{C, D\}$. Similarly, vnode 1 corresponds to a **(X,Y)**-decomposition, where $X = \{A\}$ and $Y = \{B\}$.

2.2.3 Structured Z-d-DNNF

Structured Z-d-DNNF is a tractable representation that represents families of sets [22]. In this study, we used structured Z-d-DNNF as the output from the compilation of independent sets in a given graph. Structured Z-d-DNNF represents a family of sets as a DAG that shows recursive **(X,Y)**-decompositions of the set family.

Structured Z-d-DNNF consists of one or more nodes, called *dnodes*. A dnode d corresponds to a family of sets. Let $\langle d \rangle$ be the set family corresponding to d. Structured Z-d-DNNF respecting a vtree Φ is inductively defined as follows.

Definition 3. A structured Z-d-DNNF that respects vtree Φ is one of the following.

(1) $d = \epsilon$ or $d = \perp$.
These represent families of sets $\{\emptyset\}$ and $\emptyset$, respectively. [*2]

(2) $d = e_\alpha$ or $d = \pm e_\alpha$.
These represent families of sets $\{\{e_\alpha\}\}$ and $\{\{e_\alpha\}, \emptyset\}$, respectively. Here, α is a leaf of vtree Φ and e_α is the item respecting vnode α. To identify the vnode that the dnode respects, it is denoted by d_α.

(3) $d = \{(d_{\beta^l}^1, d_{\beta^r}^1), \ldots, (d_{\beta^l}^n, d_{\beta^r}^n)\}$.
Here, β^l and β^r are the left and right children of the internal vnode β of a vtree Φ, respectively. d represents the families of sets $\bigcup_{i=1}^n \langle d_{\beta^l}^i \rangle \sqcup \langle d_{\beta^r}^i \rangle$, where $d_{\beta^l}^1, \ldots, d_{\beta^l}^n$ are dnodes that respect β^l, $d_{\beta^r}^1, \ldots, d_{\beta^r}^n$ are dnodes that respect β^r, and $\langle d \rangle$ is **(X, Y)**-decomposed with $X = E_{\beta^l}$ and $Y = E_{\beta^r}$.

The former two dnodes are called *terminal dnodes* and the latter is called a *decision dnode*. The ordered pair of dnodes $(d_{\beta^l}^i, d_{\beta^r}^i)$ that appears in a decision dnode is called an *element*. The size of a structured Z-d-DNNF is defined as the number of elements.
A decision node combined with its child elements is called a *decomposition*. An **(X, Y)**-decomposition $\{(d_{\beta^l}^1, d_{\beta^r}^1), \ldots, \}$ is said

[*2] $\emptyset$ is an empty family and $\{\emptyset\}$ is a family having only an empty set.

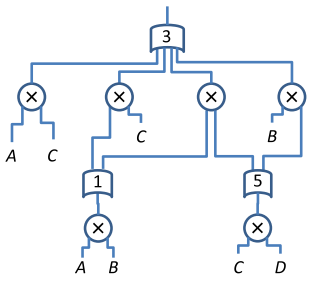

Fig. 2 Structured Z-d-DNNF respecting vtree in Fig. 3, which represents families of sets, $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$.

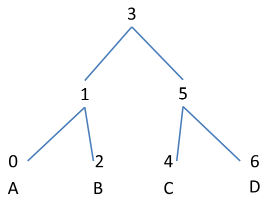

Fig. 3 Example of vtree.

to satisfy *determinism* if $(\langle d^i_{\beta^l}\rangle \sqcup \langle d^i_{\beta^r}\rangle) \cap (\langle d^j_{\beta^l}\rangle \sqcup \langle d^j_{\beta^r}\rangle) = \emptyset$ for any $i \neq j$. An **(X, Y)**-decomposition $\{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, \}$ is said to satisfy *structured decomposability* if $E_{\beta^l} \cap E_{\beta^r} = \emptyset$. Every **(X, Y)**-decomposition that corresponds to a decision dnode of structured Z-d-DNNF satisfies determinism and structured decomposability.

If a structured Z-d-DNNF has no **(X,Y)**-decomposition of the form $\{(\epsilon,\alpha)\}$ or $\{(\alpha,\epsilon)\}$, it is said to be *empty-trimmed*. If a structured Z-d-DNNF has no **(X,Y)**-decomposition of the form $(\perp,\alpha)$ or $(\alpha,\perp)$, it is said to be *empty-removed*. A structured Z-d-DNNF that is empty-trimmed and empty-removed is said to be *reduced*. Note that unlike ZDDs [19], a reduced structured Z-d-DNNF is not always canonical, i.e., the topologies of two structured Z-d-DNNFs that respect the same vtree and represent the same family of sets may differ. Given a structured Z-d-DNNF, some queries can be efficiently answered, including the computation of the size of a family of sets or enumeration of all sets in the family.

In **Fig. 2**, we show an example of a structured Z-d-DNNF that respects the vtree shown in **Fig. 3** and represents a family of sets, $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$. An "or-gate"-shaped node in the figure represents a decision dnode, where the given number represents the ID of the vnode that it respects. Furthermore, a circle node marked with a cross combined with its left and right children represents an element. The size of the structured Z-d-DNNF is six. For example, the leftmost element (A,C) represents a family of sets $\{\{A,C\}\}$, whereas the lower elements (A,B) and (C,D) represent the families of sets, $\{\{A,B\}\}$ and $\{\{C,D\}\}$, respectively. In addition, upper elements other than (A,C) represent the families of sets $\{\{A,B\}\} \sqcup \{\{C\}\} = \{\{A,B,C\}\}$, $\{\{A,B\}\} \sqcup \{\{C,D\}\} = \{\{A,B,C,D\}\}$, $\{\{B\}\} \sqcup \{\{C,D\}\} = \{\{B,C,D\}\}$, respectively. Consequently, the root dnode of Fig. 2 represents families of sets $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$.

2.3 Dynamic Programming on Tree Decomposition to Find the Size of the Maximum Independent Set

In this section, we explain dynamic programming on tree decomposition to compute the size of the maximum independent set of a given graph [23], [24]. As described in Section 3, we apply this method to knowledge compilation.

Before offering a formal explanation, we provide a sketch of the algorithm. In this method, first a depth first search is conducted on the nice tree decomposition of a given graph. In this search procedure, each tnode is traversed from a child to a parent and the maximum independent set candidate is searched. Then, the size of the maximum independent set is computed. To distinguish which gnodes are and are not contained in an independent set, 1 is mapped to the former and 0 to the latter. The former gnodes must not be adjacent according to the definition of an independent set.

Here, the key is that the problem can be solved by locally checking the adjacency of gnodes; that is, it is sufficient to check the adjacency of the gnodes mapped with 1 on each tnode because of the separation Lemma 1, which is explained in Section 3.1.

The method is formally explained here. In this method, a simple undirected connected graph $G = (V,E)$ and its nice tree decomposition $\mathcal{T}$ are the inputs. Then, the size of the maximum independent set is computed. Here, $\mathcal{T}_i$ is the subtree of $\mathcal{T}$ rooted at i. In addition, we define $V_i = \bigcup_{j \in T(\mathcal{T}_i)} B_j$ and let G_i be the subgraph induced by V_i. Furthermore, $s(i)$ is the size of the maximum independent set of G_i. Subsequently, for each bag B_i in $\mathcal{T}$, we define the sets of gnodes, P_i and Q_i, that satisfy $P_i \cup Q_i = B_i, P_i \cap Q_i = \emptyset$. Note that, in the preceding sketch, a set of gnodes that is mapped to 1 in tnode i corresponds to P_i, and one mapped to 0 corresponds to Q_i. Furthermore, $s(i; P_i, Q_i)$ is defined as follows [*3]:

$$s(i;P_i,Q_i) = max \left\{ |S_i| \;\middle|\; \begin{array}{l} S_i \subseteq V_i \\ S_i \text{ is an independent set} \\ P_i \subseteq S_i, Q_i \cap S_i = \emptyset, P_i \cup Q_i = B_i \end{array} \right\}. \quad (4)$$

That is, $s(i;P_i,Q_i)$ is the maximum size among the independent sets of G_i that contain all gnodes of P_i and do not contain any gnodes of Q_i.

In this method, a depth first search is conducted on the tnodes on $\mathcal{T}$ and the process is categorized according to the types of nodes. Next, the following computation according to the type of each tnode is performed. Consequently, $s(root)$ equals the size of the maximum independent set. Here, the child of an introduce or forget tnode i is j, and the children are j and k when i is a join tnode. In addition, it is supposed that $B_i = B_j \cup \{u\}$ when i is an introduce tnode, and that $B_i = B_j \setminus \{w\}$ when i is a forget node.

(1) i is a leaf:
Clearly, $s(i) = 0$.

(2) i is an introduce node:
The process is defined according to three cases:
(I) There exist two adjacent gnodes in P_i,
(II) There do not exist two adjacent gnodes in P_i and $u \in P_i$,
(III) There do not exist two adjacent gnodes in P_i and $u \in Q_i$.

[*3] We refer to a Japanese article that divides the gnodes in a tnode into two groups to explain this method, http://dopal.cs.uec.ac.jp/okamotoy/lect/2016/treewidth/lect07.pdf

$$s(i; P_i, Q_i) = \begin{cases} -\infty & (I) \\ s(j; P_i \setminus \{u\}, Q_i) + 1 & (II) \\ s(j; P_i, Q_i \setminus \{u\}) & (III) \end{cases} . \quad (5)$$

(3) i is a forget node:

$$s(i; P_i, Q_i) = max\{s(j; P_i \cup \{w\}, Q_i), s(j; P_i, Q_i \cup \{w\})\}. \quad (6)$$

(4) i is a join node:

$$s(i; P_i, Q_i) = s(j; P_i, Q_i) + s(k; P_i, Q_i) - |P_i|. \quad (7)$$

Here, the size of the states of each tnode is $O(2^W)$. It takes $O(|V|)$ to naively check the adjacency of gnodes. However, a method that achieves the computation in $O(W)$ has been proposed [24]. Thus, the time complexity of this computation for the entire node is $O(2^W|\mathcal{T}|)$.

2.4 Frontier-based Search

Knuth proposed *Simpath* [18], which is a method for compiling graph substructures contained in a given graph, e.g., *s*-*t* paths or cycles, into a tractable language ZDD [19]. This method is also called a *frontier-based search* [17]. In this method, an exhaustive search is conducted on the edges of the given graph successively in a predefined order. ZDDs, where each ZDD node represents the candidate subsets of graph substructures, are simultaneously constructed. However, if the procedure is conducted naively, the number of generated ZDD nodes grows exponentially and the procedure soon becomes intractable. To reduce the number of ZDD nodes during the search procedure, a "label" is assigned to the gnodes. The labels are mapped to the gnodes based on the state of the incident edges and are designed to identify the following two conditions pertaining to the candidate subsets. (1) If two candidates share the same remaining search space, the state of these candidates can be judged as equivalent. Subsequently, the corresponding ZDD nodes can be merged. (2) If the state of a candidate is invalid for the graph substructure, the corresponding ZDD node can be pruned. These labels are called *mates*. Knuth showed that, by designing a suitable *mate*, several types of graph substructures, such as *s*-*t* paths, cycles, or Hamiltonian paths [18], can be compiled by using the frontier-based search. Kawahara et al. showed that it can be used to compile other types of graph substructures, e.g., spanning tree or matching [17].

As a trivial extension of Knuth or Kawahara's approach, the independent sets contained in an input graph can be compiled as follows. Instead of on the edges, an exhaustive search is conducted on the gnodes of the given graph successively in a predefined order. ZDDs, where each ZDD node represents a candidate of independent sets, are simultaneously constructed. In the procedure, $mate[v] = 1$ is assigned to gnodes v contained in the independent set candidate, and $mate[u] = 0$ is assigned to gnodes u not contained in the candidate. Here, gnodes to which no *mate* was assigned are called *unprocessed*. Whereas, if *mates* are assigned to a gnode including all of its adjacent gnodes, the gnode is called *processed*. The remaining gnode set is defined as *frontier gnodes*. By definition, an unprocessed gnode is not adjacent to any processed gnode. Therefore, only the adjacency of the frontier gnodes must be checked. Thus, if the values of the mates assigned to the frontier gnodes are equivalent in two independent set candidates, they share the same remaining search space. Then, the corresponding ZDD nodes can be merged. However, if both values of the *mates* assigned to two adjacent gnodes on a frontier equal to 1, the property of independent sets is violated, then the corresponding ZDD node can be pruned. The assignment of *mates* to the frontier gnodes is called *configuration*.

Note that, frontier gnode set is equal to *vertex separator*, where, the j-th vertex separator F_j on the gnode order $(v_1, \ldots, v_n)$ is defined as $F_j = \{v_i | i \leq j, \exists k > j, \{v_i, v_k\} \in E\}$. *Vertex separation number* is defined as $\max_{1\leq i\leq |V|} |F_i|$. It is known that when we find the gnode order with minimum vertex separation number of a graph, the minimum vertex separation number is equal to the path-width of the graph [35].

3. Our Method

Our proposed method takes a connected undirected graph G, its nice tree decomposition $\mathcal{T}$, and the vtree Φ generated from the nice tree decomposition as inputs. [*4] Furthermore, the independent sets contained in the input graph are compiled and its result becomes the output of our method. Here, the size of the independent sets tends to be very large, and the computational cost is high if it is computed naively.

Thus, we adopted a method to compile the independent sets that uses a structured Z-d-DNNF [22]. Because the size of the output of a structured Z-d-DNNF is much smaller than that of the independent sets that it represents, the process of compilation can efficiently be conducted using this method rather than a naive method.

In the following, we first describe the tree decomposition property called *separation lemma*, which plays an important role in the design of our method (Section 3.1). Then, we explain our method for compiling the independent sets in an input graph. To clarify the comparison of dynamic programming for finding the maximum number of independent sets and the proposed method, we first describe our compilation method as a dynamic programming scheme for computing the independent sets (Section 3.2). Then, we explain the method for constructing the vtree from the nice tree decomposition of an input graph (Section 3.3), additionally the method for compiling the independent sets into structured Z-d-DNNFs using the vtree (Section 3.4). Because the two children nodes of a join node of nice tree decomposition are not mutually exclusive as two children of an internal vnode are, we need to modify a nice tree decomposition before we construct a vtree and structured Z-d-DNNF. Further, we explain the correctness and complexity of the algorithm (Section 3.5), and finally, queries and transformations supported by structured Z-d-DNNF (Section 3.6).

3.1 Separation Lemma

The nodes of a tree decomposition satisfy the property shown in the following Lemma 1, which plays an important role in the design of our method, as shown later in this section.

[*4] We show the method for generating a vtree from a nice tree decomposition in Section 3.3.

Lemma 1 (Separation Lemma)**.** Let $\mathcal{T}$ be the tree decomposition of a graph G and i, j and k be the tnodes of $\mathcal{T}$, where j exists on the path from i to k. Then, an edge that connects a gnode of $u \in B_i \setminus B_j$ to that of $v \in B_k \setminus B_j$ does not exist. In other words, a bag B_j *separates* $B_i \setminus B_j$ and $B_k \setminus B_j$ [33].

3.2 Dynamic Programming

In this section, we explain the use of dynamic programming for compiling the independent sets of an input graph.

Before offering a formal explanation, we provide a sketch of the algorithm. As the same with the algorithm in Section 2.3, we first conduct a depth first search on the nice tree decomposition of a given graph. Here, instead of maximum independent set candidates, we search independent set candidates. Also, as the same in Section 2.3, to distinguish which gnodes are and are not contained in an independent set, 1 is mapped to the former and 0 to the latter. The former gnodes must not be adjacent and it can be solved by locally checking the adjacency of gnodes because of the separation Lemma 1 in Section 3.1.

The method is formally explained as follows. Let $\mathcal{T}_i$ be the subtree of $\mathcal{T}$ rooted at i, and let V_i be $V_i = \bigcup_{j \in T(\mathcal{T}_i)} B_j$ and let $\mathcal{S}(i)$ be the independent sets of G_i.

For each tnode i of $\mathcal{T}$, we define gnode sets P_i, Q_i to satisfy $P_i \cup Q_i = B_i$ and $P_i \cap Q_i = \emptyset$. Furthermore, we define $\mathcal{S}(i; P_i, Q_i)$ as follows:

$$\mathcal{S}(i; P_i, Q_i) = \left\{ S_i \;\middle|\; \begin{array}{l} S_i \subseteq V_i \\ S_i \text{ is an independent set} \\ P_i \subseteq S_i, Q_i \cap S_i = \emptyset, P_i \cup Q_i = B_i \end{array} \right\}. \tag{8}$$

That is, $\mathcal{S}(i; P_i, Q_i)$ is the family of independent sets of G_i, and an independent set $S_i \in \mathcal{S}$ contains all gnodes of P_i and none of Q_i. When we use *mate*, as described in Section 2.4, we have $mate[v] = 1$ for a gnode v that is contained in P_i, and we have $mate[u] = 0$ for u contained in Q_i.

As in the method for finding maximum independent sets described in Section 2.3, we conduct a depth-first search on the tnodes of the tree decomposition, where processing begins from the lower tnode. When we classify the process according to the tnode types, we obtain the recursive expressions in Theorems 4, 5, 6, and 7. Note that, as a consequence of Lemma 1, more than two independent sets that share the same combination of P_i and Q_i share the same subsequent processing after B_i; that is, the combination of P_i and Q_i plays the same role as a configuration in the frontier-based search described in Section 2.4.

Theorem 4 (Leaf Nodes)**.** $\mathcal{S}(i) = \mathcal{S}(i; \emptyset, \emptyset) = \{\emptyset\}$.

Proof. Because a leaf tnode contains a gnode set $\emptyset$, the theorem is proven.

Theorem 5 (Introduce Nodes)**.** Let i be an introduce node and j be the child of i; gnode u not contained in B_j is introduced in B_i. We then classify the process into three cases as described in Section 2.3: (I) There exist more than one adjacent gnodes in P_i,

(II) No two gnodes in P_i are adjacent and $u \in P_i$,

(III) No two gnodes in P_i are adjacent and $u \in Q_i$.

We then have

$$\mathcal{S}(i; P_i, Q_i) = \begin{cases} \{\emptyset\} & \text{(I)} \\ \mathcal{S}(j; P_i \setminus \{u\}, Q_i) \sqcup \{\{u\}\} & \text{(II)} \\ \mathcal{S}(j; P_i, Q_i \setminus \{u\}). & \text{(III)} \end{cases} . \tag{9}$$

Proof. (I) Suppose that $I \cap B_i = P_i$ holds for a gnode set I in G_i, and $u, v \in P_i$ holds for some $\{u, v\} \in E$. Then, I is not an independent set.

(II) Suppose that $I \in \mathcal{S}(j)$, $I \cap B_j = P_j$, and $\{u, v\} \notin E$ for all $v \in P_j$. Then, because all the adjacent gnodes of u in G_i are contained in B_j from Lemma 1, $I \cup \{u\}$ is an independent set. Because $(I \cup \{u\}) \cap B_i = P_j \cup \{u\}$, this case is proven.

(III) Because $u \in B_i$, it holds that $P_i = P_j$. Consequently, if an independent set I satisfies $I \cap B_j = P_j$, it also holds that $I \cap B_i = P_i$. Thus, this case is proven.

Theorem 6 (Forget Nodes)**.** Suppose that i is a forget node, j is the child of i, and gnode w contained in B_j is forgotten in B_i. There can then exist more than one independent set, in which the configuration is different in B_j, whereas it is equal in B_i. Because these independent sets share the same subsequent processing after B_i, a union operation is performed to unify the processing. Consequently, it holds that

$$\mathcal{S}(i; P_i, Q_i) = \mathcal{S}(j; P_i \cup \{w\}, Q_i) \cup \mathcal{S}(j; P_i, Q_i \cup \{w\}). \tag{10}$$

Proof. $\mathcal{S}(i; P_i, Q_i)$ is the union of the two following sets: (1) All independent sets I in G_i, in which it holds that $I \cap B_i = P_i$ and $w \in I$. (2) All independent sets I in G_i, in which it holds that $I \cap B_i = P_i$ and $w \notin I$. Consequently, it holds that $\mathcal{S}(j; P_i \cup \{w\}, Q_i)$ in the former, whereas it holds that $\mathcal{S}(j; P_i, Q_i \cup \{w\}\}$ in the latter.

Theorem 7 (Join Nodes)**.** Suppose that i is a join node and j and k are the two children of i. Then, we have

$$\mathcal{S}(i; P_i, Q_i) = \mathcal{S}(j; P_i, Q_i) \sqcup \mathcal{S}(k; P_i, Q_i). \tag{11}$$

Proof. For all $I_j \in \mathcal{S}(j; P_i, Q_i)$ and all $I_k \in \mathcal{S}(k; P_i, Q_i)$, it holds that $I_j \cap B_j = I_k \cap B_k = I_j \cap B_i = I_k \cap B_i = P_i$, because $B_i = B_j = B_k$. Here, from Lemma 1, it holds that $\{v, u\} \notin E$ for $\forall v \in I_j \setminus P_i$ and $\forall u \in I_k \setminus P_i$. Consequently, gnode sets $I_j \setminus P_i$ and $I_k \setminus P_i$ do not share adjacent gnodes. Thus, it holds that $I_i = I_j \cup I_k$ is an independent set of G_i for $\forall I_j \in \mathcal{S}(j; P_i, Q_i)$ and $\forall I_k \in \mathcal{S}(k; P_i, Q_i)$.

3.3 Vtree Construction from a Nice Tree Decomposition

Next, we explain the construction of a dummy-added vtree from a given nice tree decomposition. For simplicity, it is called a vtree described below. The problem here is the difference between the properties of an internal node of a vtree and those of a join tnode of a nice tree decomposition. Namely, at each internal node of a vtree, the sets corresponding to the leaves of the left and right sub-vtrees are disjoint, whereas, in a join tnode i and its two children j and k of a nice tree decomposition, V_j and V_k are not always mutually exclusive.

Therefore, we first transform a given nice tree decomposition $\mathcal{T}$ such that the gnodes contained in the left and right children of each internal node are disjoint. The transformation method is as follows. For every join tnode i and its two children j and k in a nice tree decomposition $\mathcal{T}$, we can specify j as the left child and k as the right child without loss of generality. For j and k, we let V_m be $V_m = V_j \cap V_k$. Furthermore, for each tnode k included in

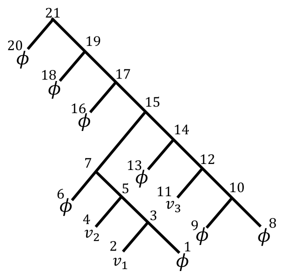

Fig. 4 Example of a dummy-added vtree generated from a nice tree decomposition in Fig. 1.

the sub-vtree on the right, we change it to $B'_k = B_k \setminus V_m$ and limit the gnode contained in V_k and its child tnodes to B'_k. We conduct this transformation on each join tnode from the bottom to the root and let the obtained tree be $\mathcal{T}'$ [*5].

Then, we conduct a depth-first search on the tree $\mathcal{T}'$, giving priority to the left side, and generate a vnode corresponding to each node of the tree from a child to a parent.

Here, we inductively define the process of vnode generation. A node of tree $\mathcal{T}'$ is denoted as $i, j, k, \ldots$.

(1) When a node i has no child, we generate the corresponding vnode α that is a dummy $e_\alpha = \phi$.

(2) When a node i has one child j for which $B'_i = B'_j \cup \{u\}$ holds, there exists a vnode α, where $E_\alpha = V_j$ from the inductive assumption. We generate a vnode β and let $\beta^r = \alpha$. Furthermore, we generate a vnode β^l, where $e_{\beta^l} = u$.

(3) When a node i has one child j for which $B'_i \cup \{w\} = B'_j$ holds, as in (2), we generate a vnode β and let $\beta^r = \alpha$. Then, we generate a dummy vnode β^l, where $e_{\beta^l} = \phi$.

(4) When a node i has one child j for which $B'_i = B'_j$ holds, we have the same vnode generation as that described in (3).

(5) When a node i has two children j and k, there exists a vnode α, where $E_\alpha = V_j$, and a vnode β, where $E_\beta = V_k$ from the inductive assumption. We generate a vnode γ and let $\gamma^l = \alpha$ and $\gamma^r = \beta$.

In **Fig. 4**, we show an example of a vtree generated from a nice tree decomposition of Fig. 1.

3.4 Compilation with Structured Z-d-DNNF

In this section, we present a method for representing the dynamic programming scheme described in Section 3.2 with a structured Z-d-DNNF. The problem here involves the structured decomposability of a structured Z-d-DNNF. That is, in each element of a structured Z-d-DNNF, the families of sets represented by the left and right ordered pairs are mutually exclusive, whereas the ground sets of two children of $\mathcal{S}(i)$ are not necessarily disjoint. As a result, a structured Z-d-DNNF cannot be used to represent the hierarchy of $\mathcal{S}(i)$ as it is.

Therefore, we limit $\mathcal{S}(i)$ to the tree $\mathcal{T}'$ described in Section 3.3. Suppose that we have a join tnode i and its left and right children j and k, where $V_m = V_j \cap V_k$ holds. Then, for the right child and its descendant r, we exchange $\mathcal{S}(r)$ with $\mathcal{S}'(r) = \{S \setminus V_m | S \in \mathcal{S}(r)\}$. Each $S(i; P_i, Q_i)$ we exchange with $S'(i; P'_i, Q'_i)$, where $P'_i = P_i \setminus B_m$, $Q'_i = Q_i \setminus B_m$. We conduct this process for each join tnode from the root to the bottom. Even after performing this process, the families of sets represented by the root $\mathcal{S}'(root)$ equal $\mathcal{S}(root)$, and all the independent sets of the input graph are represented.

Then, we construct a dnode according to the types of node i, as described in detail below, and we let it have one-to-one correspondence with a certain $\mathcal{S}'(i; P'_i, Q'_i)$. As shown in Section 3.5, the hierarchy of these nodes meets the definition of a structured Z-d-DNNF.

In the following, we inductively define the method for composing a dnode with regard to node type i.

(1) When i has no child, we let the corresponding dnode be ϵ, because the vnode to which it corresponds is a dummy vnode.

(2) When a node i has one child j for which $B'_i = B'_j \cup \{u\}$ holds, $S'(j)$ has its corresponding dnodes based on the inductive assumption. When gnode u is contained in P'_i, we construct an element with the ordered pair of dnodes, one of which is a dnode that represents $\{\{u\}\}$ and the other is a decision dnode that corresponds to $\mathcal{S}'(j; P'_i \setminus \{u\}, Q'_i)$. Subsequently, we let the element be a child of a decision dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$. On the other hand, when gnode u is contained in Q'_i, we construct an element with the ordered pair of dnodes, one of which is a terminal dnode representing $\{\epsilon\}$ and the second is a decision dnode corresponding to $\mathcal{S}'(j; P'_i, Q'_i \setminus \{u\})$. Subsequently, we let the element be a child of a decision dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$.

(3) When a node i has one child j for which $B'_i \cup \{w\} = B'_j$ holds, $S'(j)$ has its corresponding dnodes based on the inductive assumption. When both $\mathcal{S}'(j; P'_i \cup \{w\}, Q'_i)$ and $\mathcal{S}'(j; P'_i, Q'_i \cup \{w\})$ exist, we construct an element using an ordered pair with a terminal dnode $\{\epsilon\}$ and the corresponding dnode of each. These are (ϵ, d^1_j) and (ϵ, d^2_j), where d^1_j corresponds to $\mathcal{S}'(j; P'_i \cup \{w\}, Q'_i)$ and d^2_j corresponds to $\mathcal{S}'(j; P'_i, Q'_i \cup \{w\})$. Subsequently, we let the element be the child of a decision dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$. When one of $\mathcal{S}'(j; P'_i \cup \{w\}, Q'_i)$ and $\mathcal{S}'(j; P'_i, Q'_i \cup \{w\})$ does not exist, a dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$ has only one child.

(4) When a node i has one child j for which $B'_i = B'_j$ holds, $S'(j)$ has its corresponding dnodes based on the inductive assumption. We construct an element with the ordered pair of dnodes, one of which is a terminal dnode that represents $\{\epsilon\}$ and the second is a decision dnode that corresponds to $\mathcal{S}'(j; P'_j, Q'_j)$. Subsequently, we let the element be a child of a decision dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$, where $P'_i = P'_j$ and $B'_i = B'_j$.

(5) When a node i has two children j and k, j and k have their corresponding dnodes based on the inductive assumption. We then construct an element with the ordered pair, where the dnodes share the same configuration $\mathcal{S}'(j; P'_j, Q'_j)$ and $\mathcal{S}'(k; P'_k, Q'_k)$, and where $P'_j = P'_k, B'_j = B'_k$. Subsequently, we let the element be the child of a dnode corresponding to $\mathcal{S}'(i; P'_i, Q'_i)$.

Example. We describe an example of the method for constructing a structured Z-d-DNNF. We take the graph of Fig. 1 and its

[*5] $\mathcal{T}'$ cannot always be a tree decomposition, because it does not always satisfy (2) in Definition 1.

vtree Fig. 4 as inputs. Then, we have **Fig. 5** as the result of compiling all the independent sets contained in the input graph. In Fig. 5, the decision dnode respecting the same vnode α is said to be $d^1_\alpha, d^2_\alpha, \ldots$ from the left. For example, the decision nodes with 3 shown at the bottom of the figure are d^1_3, d^2_3 from the left, respectively. Furthermore, **Table 2** shows the state at each tnode and vnode during processing.

In tnode 1, the family of independent sets is only $\mathcal{S}(1; \emptyset, \emptyset) = \{\emptyset\}$. Thus, a terminal dnode representing $\{\emptyset\}$ is constructed that respects a vnode 1. In tnode 2, there are two families of independent sets: one is $\mathcal{S}(2; \emptyset, \{1\}) = \{\emptyset\}$, which does not contain gnode 1 in B_2, and the second is $\mathcal{S}(2; \{1\}, \emptyset) = \{\{1\}\}$, which does contain gnode 1 in B_2. The corresponding elements are (ϵ, ϵ) and (v_1, ϵ), which are contained in the two decision dnodes d^1_3 and d^2_3, respectively. Vnode 3 is respected by these decision dnodes. In tnode 3, because gnodes 1 and 2 are not contained in the same independent set, we have the families of independent sets $\mathcal{S}(3; \emptyset, \{1, 2\}) = \{\emptyset\}$, $\mathcal{S}(3; \{2\}, \{1\}) = \{\{2\}\}$, and $\mathcal{S}(3; \{1\}, \{2\}) = \{\{1\}\}$. The corresponding elements are (ϵ, d^1_3), (v_2, d^1_3), and (ϵ, d^2_3), which are contained in the decision dnodes d^1_5, d^2_5, and d^3_5. A vnode 5 is respected by these decision dnodes. In tnode 4, we have the families of independent sets $\mathcal{S}(4; \emptyset, \{1\}) = \{\emptyset, \{2\}\}$ and $\mathcal{S}(4; \{1\}, \emptyset) = \{\{1\}\}$, because gnode 2 is forgotten. The corresponding elements are (ϵ, d^1_5), (ϵ, d^2_5), and (ϵ, d^3_5). The configurations of these elements are $\{mate[1] = 0, mate[2] = 0\}$, $\{mate[1] = 0, mate[2] = 1\}$, and $\{mate[1] = 1, mate[2] = 0\}$, respectively. Because $B_4 = \{1\}$, the first two are merged and contained in the same decision dnode d^1_7, and the last is contained in decision dnode d^2_7; both of these dnodes respect vnode 7.

In tnode 5, we have the family of independent sets $\mathcal{S}(5; \emptyset, \emptyset) = \{\emptyset\}$. Thus, a terminal dnode representing $\{\emptyset\}$ is constructed that respects vnode 8. In tnode 6, we have the families of independent sets $\mathcal{S}(6; \emptyset, \{1\}) = \{\emptyset\}$ and $\mathcal{S}(6; \{1\}, \emptyset) = \{\{1\}\}$. Because, vnode 9 is a dummy node that has no corresponding gnode, we construct an element (ϵ, ϵ) from the vnodes corresponding to tnode 6 and give them the configurations of $\mathcal{S}(6; \emptyset, \{1\})$ and $\mathcal{S}(6; \{1\}, \emptyset)$, respectively. In addition, these are contained in the decision dnodes d^1_{10} and d^2_{10}, respectively, both of which respect vnode 10.

Next, we conduct this process up to tnode 8 in the same manner. In tnode 9, we construct two elements with the ordered set of the decision dnodes of 7 and 14, which share the same configuration. We then let them be contained in decision dnodes d^1_{15} and d^2_{15}, both of which respect vnode 15.

In the root tnode 12, because the frontier gnode is an empty set, all of the lower dnodes are contained in one decision dnode d_{21}, which represents $\mathcal{S}(12; \emptyset, \emptyset) = \{\emptyset, \{2\,\{3\}, \{4\}, \{2, 4\}, \{3, 4\}, \{1\}\}$ and respects vnode 21.

We show pseudo-code for the proposed method in Algorithm 1. Construct takes sets of decision dnodes D at each tnode and tnode id i as inputs. The output is D, which is added to the computation result. This algorithm conducts an exhaustive search on

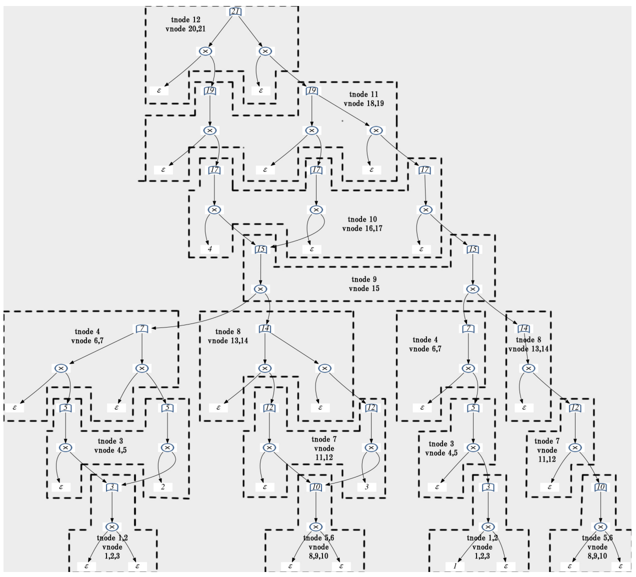

Fig. 5 Compilation result of the graph in Fig. 1.

Algorithm 1 Construct(D, i)
```
1: // $D[i]$ is a decision dnode corresponding to tnode $i$
2: if $i$ is introduce then
3:     $D[\mathrm{Child}(i)] \leftarrow \mathrm{Construct}(D, \mathrm{Child}(i))$
4:     $D[i] \leftarrow \mathrm{Introduce}(D, \mathrm{Child}(i))$
5: else if $i$ is forget then
6:     $D[\mathrm{Child}(i)] \leftarrow \mathrm{Construct}(D, \mathrm{Child}(i))$
7:     $D[i] \leftarrow \mathrm{Forget}(D, \mathrm{Child}(i))$
8: else if $i$ is join then
9:     $D[\mathrm{ChildLeft}(i)] \leftarrow \mathrm{Construct}(D, \mathrm{ChildLeft}(i))$
10:    $D[\mathrm{ChildRight}(i)] \leftarrow \mathrm{Construct}(D, \mathrm{ChildRight}(i))$
11:    $D[i] \leftarrow \mathrm{Join}(D, \mathrm{ChildLeft}(i), \mathrm{ChildRight}(i))$
12: else if $i$ is leaf then
13:    $D[i] \leftarrow \mathrm{Leaf}()$
14: end if
15: return $D$
```

Table 2 Process with inputs Fig. 1 and Fig. 4.

Tnode	Vnodes Dnodes	BAG	Type	$\mathcal{S}(i; P_i, Q_i)$	Families of Independent Sets
1	1	$\emptyset$	Leaf	$\mathcal{S}(1; \emptyset, \emptyset)$	$\{\emptyset\}$
2	2,3	$\{1\}$	Introduce	$\mathcal{S}(2; \emptyset, \{1\}), \mathcal{S}(2; \{1\}, \emptyset)$	$\{\emptyset\}$,$\{\{1\}\}$
3	4,5	$\{1, 2\}$	Introduce	$\mathcal{S}(3; \emptyset, \{1, 2\}), \mathcal{S}(3; \{1\}, \{2\}), \mathcal{S}(3; \{2\}, \{1\})$	$\{\emptyset\}$,$\{\{1\}\}$,$\{\{2\}\}$
4	6,7	$\{1\}$	Forget	$\mathcal{S}(4; \emptyset, \{1\}), \mathcal{S}(4; \{1\}, \emptyset)$	$\{\emptyset, \{2\}\}$,$\{\{1\}\}$
5	8	$\emptyset$	Leaf	$\mathcal{S}(5; \emptyset, \emptyset)$	$\{\emptyset\}$
6	9,10	$\{1\}$	Introduce	$\mathcal{S}(6; \emptyset, \{1\}), \mathcal{S}(6; \{1\}, \emptyset)$	$\{\emptyset\}$,$\{\{1\}\}$
7	11,12	$\{1, 3\}$	Introduce	$\mathcal{S}(7; \emptyset, \{1, 3\}), \mathcal{S}(7; \{1\}, \{3\}), \mathcal{S}(7\{3\}, \{1\})$	$\{\emptyset\}$,$\{\{1\}\}$,$\{\{3\}\}$
8	13,14	$\{1\}$	Forget	$\mathcal{S}(8; \emptyset, \{1\}), \mathcal{S}(8; \{1\}, \emptyset)$	$\{\emptyset, \{3\}\}$,$\{\{1\}\}$
9	15	$\{1\}$	Join	$\mathcal{S}(9; \emptyset, \{1\}), \mathcal{S}(9; \{1\}, \emptyset)$	$\{\emptyset, \{2\}, \{3\}, \{2, 3\}\}$,$\{\{1\}\}$
10	16,17	$\{1, 4\}$	Introduce	$\mathcal{S}(10; \emptyset, \{1, 4\}), \mathcal{S}(10; \{1\}, \{4\}), \mathcal{S}(10; \{4\}, \{1\})$	$\{\emptyset, \{2\}, \{3\}, \{4\,\{2, 4\}, \{3, 4\}\}$,$\{\{1\}\}$
11	18,19	$\{4\}$	Forget	$\mathcal{S}(11; \emptyset, \{4\}), \mathcal{S}(11; \{4\}, \emptyset)$	$\{\emptyset, \{2\}, \{3\}, \{4\}, \{2, 4\}, \{3, 4\}\}$,$\{\{1\}\}$
12	20,21	$\emptyset$	Forget	$\mathcal{S}(12; \emptyset, \emptyset)$	$\{\emptyset, \{2\}, \{3\}, \{4\}, \{2, 4\}, \{3, 4\}, \{1\}\}$

the tnodes of a tree decomposition from the bottom, where the process is divided based on the types of tnodes. In Algorithm 1, Introduce, Forget, Join, and Leaf are functions that return the result corresponding to the types of tnode, where Child(i), ChildLeft(i), and ChildRight(i) return the sole child, the left child, and the right child of i, respectively.

3.5 Correctness and Complexity

Theorem 8. The proposed method correctly compiles the independent sets contained in the input graph.

Proof. In the hierarchy of nodes presented in Section 3.4, it is obvious that each node corresponds to a certain $\mathcal{S}'(i)$, and the root dnode corresponds to $\mathcal{S}'(root) = \mathcal{S}(root)$. Therefore, each node meets conditions (1), (2), and (3) in Definition 3, and it is sufficient to prove that the hierarchy meets the requirements of structured decomposability and determinism. First, we prove that the hierarchy of nodes obtained in the previous section meets the requirement of structured decomposability. Because the ground sets of the left and right subtree are mutually exclusive on each internal vnode β, each internal vnode represents a certain **(X,Y)**-decomposition. Thus, the hierarchy of Section 3.4 satisfies structured decomposability. Next, we prove that the hierarchy of Section 3.4 meets the requirement of determinism. We assume that $I_1, I_2 \in \mathcal{S}(i; P_i, Q_i)$ are limited to $I'_1, I'_2 \in \mathcal{S}'(i; P'_i, Q'_i)$ and that they satisfy $I'_1 = I'_2$ for the sake of contradiction. Let $I' = I'_1 = I'_2$. For $\forall v \in I_1 \setminus I'$, i is a forget tnode of v or its ancestor, because I_1 and I_2 are merged into $\mathcal{S}(i; P_i, Q_i)$. However, $v \in B_i$, because $I_1, I_2 \in \mathcal{S}(i; P_i, Q_i)$ are limited to $I'_1, I'_2 \in \mathcal{S}'(i; P'_i, Q'_i)$, which is a contradiction. This also applies to $\forall v \in I_2 \setminus I'$.

Theorem 9. The time complexity of the proposed method is $O(2^W|T(\mathcal{T})|)$, and the size of the compilation result of a structured Z-d-DNNF is $O(2^W|T(\mathcal{T})|)$.

Proof. The size of the states of each tnode i is $O(2^W)$, which is the same as that of the method for calculating the size of the maximum independent set described in Section 2.3. Because the time required to construct each dnode corresponding to each state is constant, the time complexity for constructing all the dnodes is $O(2^W|T(\mathcal{T})|)$, which is the same as the time required by the method described in Section 2.3. Because the size of the elements of each decision dnode is proportional to that of the states in each tnode, the size of the compilation result of structured Z-d-DNNF is $O(2^W|T(\mathcal{T})|)$.

3.6 Query and Transformation with Structured Z-d-DNNF

In this subsection, we explain some of queries and transformations supported by Structured Z-d-DNNF. First, we take queries. The size of the independent sets represented with a structured Z-d-DNNF is calculated in time $O(n)$. Here, n is the size of the output of the structured Z-d-DNNF. This query is said to be *Model Counting*. The enumeration of independent sets represented by a structured Z-d-DNNF is completed in time $O(p(n, m))$. Here, m is the size of the independent sets and $p(n, m)$ is a polynomial of degree n, m. This query is said to be *Model Enumeration* [22].

Next, we take *Random Sampling* as an example of transformation. After the compilation of independent sets of a given graph, we sometimes need to take some samples of all the independent sets. However, if the total size of the independent sets is very large, a naive enumeration method is impractical in order to select a few of them. For this purpose, random sampling of independent sets on the compilation result is useful.

We assume that the independent sets of a graph is already compiled and reduced, additionally the size of independent sets are already calculated. In addition, the cumulative sum of the child elements at each dnodes is already calculated, which is necessary for random walk described below. Henceforth, Random sampling is conducted by a following random walk: We start from the root node. When we are at a non-terminal node d, we randomly move to a node $(d^i_l, d^i_r) \in d$ with probability $\frac{\mathrm{Card}(d^i_l)\mathrm{Card}(d^i_r)}{\mathrm{Card}(d)}$, where Card($d$) represents the cardinality of the independent sets represented by d. We repeat this procedure until we reach a terminal, which yields a random sampling of the independent sets represented by d. The time complexity is proportional to $\sum_{i\in\{1,\dots,H\}} \log(\mathrm{Ave}[|d(i)|])$, where H denotes the height of the vtree that the compilation result respects to, Ave[.] denotes the average, and $|d(i)|$ denotes the size of the children of a dnode at i-th height. The pseudo code of the procedure is shown in Algorithm 2.

Algorithm 2 RandomSample(d)

Let Card(d) be the size of independent sets represented by d. We assume that d is already reduced and Card(d) is calculated.

```
 1: if $d$ is terminal then
 2:     if $d = \epsilon$ or $d = e_\alpha$ for an item $e_\alpha$ then
 3:         return $d$
 4:     else if $d = \pm e_\alpha$ for an item $e_\alpha$ then
 5:         Assign $e_\alpha$ or $\epsilon$ to $d'$ with probability $\frac{1}{2}$ respectively
 6:         return $d'$
 7:     end if
 8: else
 9:     Assign $(d^i_l, d^i_r)$ to $(d'_l, d'_r)$ with probability $\frac{\mathrm{Card}(d^i_l)\mathrm{Card}(d^i_r)}{\mathrm{Card}(d)}$ respectively.
10:     $d'_l \leftarrow$ RandomSample($d'_l$)
11:     $d'_r \leftarrow$ RandomSample($d'_r$)
12:     return $d' = \{(d'_l, d'_r)\}$
13: end if
```

4. Experiments

In this section, we describe the results of our experimental evaluation. To compare the performance of the proposed method, we conducted frontier-based search. That is, we compiled all of the independent sets in the input graph with our method and the comparison method respectively and compared their performance. Additionally, we took Model Counting and Random Sampling as an example of query and transformation respectively. The detail of the frontier-based search to compile the independent sets in an input graph is described in Section 2.4. We implemented the frontier-based search based on the implementation of TdZdd [*6].

The experiment was performed on a computer with an Intel Xeon CPU E7-8837 (2.67 GHz), 1.5 TB main memory, and Linux 4.4.104-39-default. The proposed methods were implemented in C++ and built with g++ 4.85.

For the input graphs, we used the data set ex-instances-

[*6] https://github.com/kunisura/TdZdd

Table 3 Instances for the ten smallest and ten largest of the path-width/tree-width ratio.

Instance	|V|	|E|	TW	PW	PW/TW
ex069	235	441	13	9	0.69[*10]
ex091	193	336	11	10	0.91[*10]
ex095	220	555	12	11	0.92[*10]
ex143	130	660	36	35	0.97[*10]
ex063	103	582	36	36	1.00
ex077	237	423	11	11	1.00
ex103	237	419	12	12	1.00
ex117	77	181	14	14	1.00
ex145	48	96	12	12	1.00
ex195	216	382	12	12	1.00
ex197	303	1,158	17	41	2.41
ex019	291	752	16	41	2.56
ex073	712	1,085	8	21	2.63
ex015	177	669	15	42	2.80
ex153	772	11,654	56	170	3.04
ex155	758	11,580	55	168	3.05
ex081	188	638	6	19	3.17
ex011	465	1,004	10	33	3.30
ex129	737	2,826	17	60	3.53
ex149	698	2,604	13	51	3.92

PACE2017-public-2016-12-02 [*7], which was provided for a 2017 contest for tree decomposition implementation (PACE). As the prepossessing of our method, we applied tree decomposition to each of the input graphs, transformed them into nice tree decompositions and created vtrees. Consequently, we used them as inputs to the proposed method.

We executed tree decomposition using flow-cutter-pace17 [*8], a heuristics method that won the second place in PACE heuristic tree decomposition challenge in 2017. It was decided as a rule of the contest that the current best tree decomposition must immediately print in standard output and then halt once the calculation process receives the SIGTERM signal [*9]. Thus, we uniformly terminated the calculation for each graph in two seconds.

On the other hand, as the prepossessing of comparative method, we conducted beam search to find the appropriate order of the gnodes of an input graph, which equals to the path-width [35]. We used our own implementation for the beam search with following the method detailed in Inoue et al. [20]. The beam search width we adopted was 3000. Among the data set, we excluded only ex115 because it took more than ten hours to obtain the result. Except this instance, the computation to find the order of the gnodes took two seconds to three hours per instance.

In **Table 3**, we show the instances for the ten smallest and ten largest of the path-width/tree-width ratio. Further, in **Table 4** and **Table 5**, we show the experimental results of construction, reduction. Also, in **Table 6**, we show the results of query and transformation. The rest of the instances is omitted due to lack of space.

In Table 3, the columns headed Instance, $|V|$, $|E|$, TW, PW, and PW/TW show the name of the input graph, the size of gnodes, the size of edges, the tree-width, the path-width of the graph, and the path-width/tree-width ratio respectively. We considered that the

[*7] https://people.mmci.uni-saarland.de/~hdell/pace17/ex-instances-PACE2017-public-2016-12-02.tar.bz2
[*8] https://github.com/kit-algo/flow-cutter-pace17
[*9] https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/
[*10] In these instances, path-width is smaller than tree-width because we used heuristics to calculate both values. In the following three tables, Tables 4–6, we took the path decomposition as the tree decomposition i.e., PW/TW = 1.00.

Table 4 Experimental results of the proposed and the comparative method (Construction).

	Construction				
	Our Method		Frontier-Based Search		
Instance	time (ms)	nodes	time (ms)	nodes	PW/TW
ex069	43	70,385	8	26,433	1.00
ex091	65	114,200	12	41,759	1.00
ex095	57	88,263	7	34,261	1.00
ex143	1,172	1,708,633	125	663,759	1.00
ex063	655	946,529	48	278,210	1.00
ex077	23	38,119	12	40,127	1.00
ex103	134	226,304	22	107,677	1.00
ex117	95	152,518	16	81,573	1.00
ex145	246	393,670	8	33,572	1.00
ex195	119	195,948	33	150,371	1.00
ex197	85	137,419	796,979	2,330,290,142	2.41
ex019	426	684,402	>1 h		2.56
ex073	34	57,840	28,748	94,456,354	2.63
ex015	16	22,818	351,867	1,048,684,198	2.80
ex153	19,195	22,775,996	>1 h		3.04
ex155	10,002	12,799,222	>1 h		3.05
ex081	5	8,491	183	993,796	3.17
ex011	27	46,499	391,209	1,152,363,020	3.30
ex129	200	321,957	>1 h		3.53
ex149	75	123,749	>1 h		3.92

Table 5 Experimental results of the proposed and the comparative method (Reduction).

	Reduction				
	Our Method		Frontier-Based Search		
Instance	time (ms)	nodes	time (ms)	nodes	PW/TW
ex069	29	27,385	1	8,794	1.00
ex091	61	46,738	1	12,048	1.00
ex095	41	30,300	1	5,278	1.00
ex143	1,760	575,335	23	86,983	1.00
ex063	766	56,081	8	14,974	1.00
ex077	16	13,685	1	15,506	1.00
ex103	159	84,846	4	40,242	1.00
ex117	103	55,144	2	17,549	1.00
ex145	291	66,361	1	3,680	1.00
ex195	126	70,287	5	57,224	1.00
ex197	84	34,483	193,147	23,241,232	2.41
ex019	651	192,182	>1 h		2.56
ex073	24	20,137	5,266	35,742,140	2.63
ex015	7	3,992	71,982	11,887	2.80
ex153	30,160	1,923,582	>1 h		3.04
ex155	15,000	998,317	>1 h		3.05
ex081	2	1,315	31	290,033	3.17
ex011	19	14,627	79,808	92,324,582	3.30
ex129	244	94,446	>1 h		3.53
ex149	71	42,867	>1 h		3.92

vertex separation number, which is gained with the gnode order in the last paragraph, is equal to the path-width (cf., Section 2.4.).

On the other hand, in the columns headed Our Method and Frontier-Based Search of Table 4 show the time required to compile the independent sets and the size of the output, respectively. The instances that did not finish running within one hour are marked as ">1 h". Further, the columns headed Reduction of Table 5 show the time required for reduction of the compilation output and the size of the output in the reduced form, respectively.

Additionally, in the column headed MC of Table 6, the time required for model counting is shown. Further, the columns headed Random of Table 6 show the time required for the random sampling. We tried random sampling on our method and the comparative method for one hundred times and recorded the average time required. The column Rand Init show the time required for the calculation of the cumulative sum of the child elements at each dnodes of our method. This step is required only once as the

Table 6 Experimental results of the proposed and the comparative method (Query and Transformation).

	Our Method			Frontier-Based Search		
	MC	Rand Init	Random	MC	Random	
Instance	time (ms)	time (ms)	time (ms)	time (ms)	time (ms)	PW/TW
ex069	15	20	3	0	1	1.00
ex091	27	34	2	0	1	1.00
ex095	17	23	2	0	1	1.00
ex143	457	591	4	2	0	1.00
ex063	40	47	2	0	0	1.00
ex077	9	10	3	1	1	1.00
ex103	67	82	3	1	1	1.00
ex117	37	46	1	0	0	1.00
ex145	43	56	0	0	0	1.00
ex195	49	62	3	2	1	1.00
ex197	25	28	2	1,617	24	2.41
ex019	161	208	5	>1 h		2.56
ex073	13	16	10	2,192	37	2.63
ex015	2	3	2	0	0	2.80
ex153	1,978	2,332	67	>1 h		3.04
ex155	919	1,116	35	>1 h		3.05
ex081	0	1	1	9	0	3.17
ex011	10	12	6	5,917	91	3.30
ex129	80	96	8	>1 h		3.53
ex149	31	36	8	>1 h		3.92

prepossess of binary searches in a random walk (cf., Section 3.6.).

We omitted the cardinality of the independent sets in the table, because it tends to be extremely large. For example, the largest cardinality is that of ex073, which is equal to

$$13,564,085,714,942,$$
$$581,386,021,576,849,347,911,369,251,429,$$
$$160,021,307,529,166,008,896,322,192,959,$$
$$972,367,896,454,404,330,774,718,193,622,$$
$$700,383,259,154,902,400,967,511,932,927.$$

This also shows the effectiveness of our method in the application because a naive enumeration method is impractical in order to select a few of them for the purpose of random sampling.

Unless there is a risk of misunderstanding, hereinafter, we refer to the time required for compilation and reduction as simply the time required for compilation. The compilation of the proposed method finished within 5 minutes, while, the comparative method could not finish the process within 1 hour at 7 data. However, when the path-width/tree-width ratio is small, comparative method outperforms our proposed method both for the time needed for the compilation and the size of the output. As we pointed in Section 3, the two children nodes of a join node of nice tree decomposition are not mutually exclusive, it causes overlap in the process of the compilation of our method. By contrast, the path-width/tree-width ratio is large, the efficiency of our method prevails the overlap.

On the other hand, for both of proposed and comparative method, the time required for model counting and random sampling is relatively small compared to that of compilation. Though proposed method needs the cumulative sum calculation for the prepossess of random sampling, the time required for main process of random sampling is comparable to that of comparative method. In addition, time required for nice tree decomposition and vtree creation is very short and within 50 milli seconds.

In **Fig. 6**, we compared the time required for compilation by

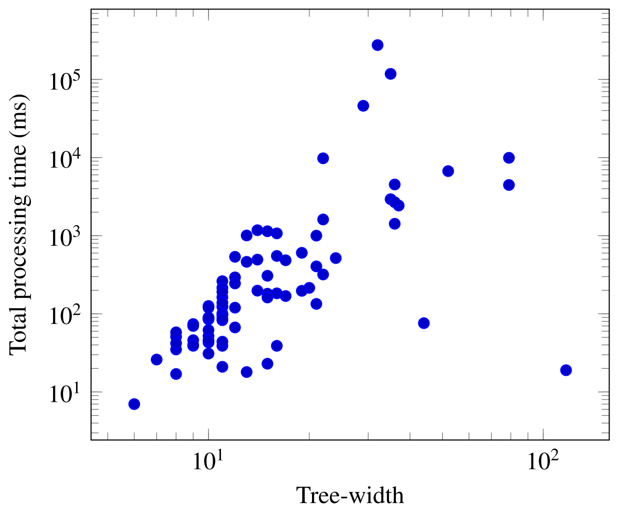

Fig. 6 Compilation time of proposed method vs. Tree-width.

proposed method with the tree-width of the input graphs. In the same way, we compared the time required for compilation by comparative method with and path-width of the input graphs in **Fig. 7**, the size of the output of proposed method with tree-width of the input graphs in **Fig. 8**, and the size of the output of compared method with path-width of the input graphs in **Fig. 9**. Former two graphs show that proposed method and comparative method are executed in proportion to the tree-width and path-width, respectively. On the other hand, latter two graphs show that the size of the output of those two methods are in proportion to the tree-width and path-width, respectively. Figure 6 and Fig. 8 supports the theoretical time complexity value discussed in Section 3.5.

In **Fig. 10**, we compared the ratio of compilation time and tree/path-width. That is, we plotted the ratio of the compilation time of the proposed method and the comparative method on the vertical axis, while the ratio of tree-width and path-width of the input graph on the horizontal axis. The figure shows that, when the tree-width of an input graph is small compared to the path-

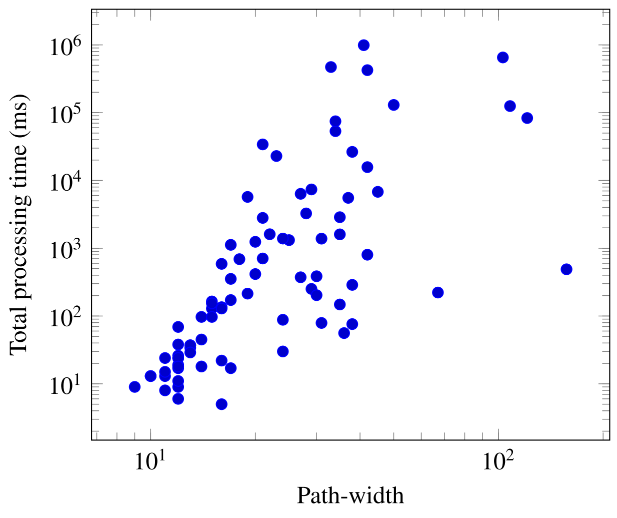

Fig. 7 Compilation time of comparative method vs. Path-width.

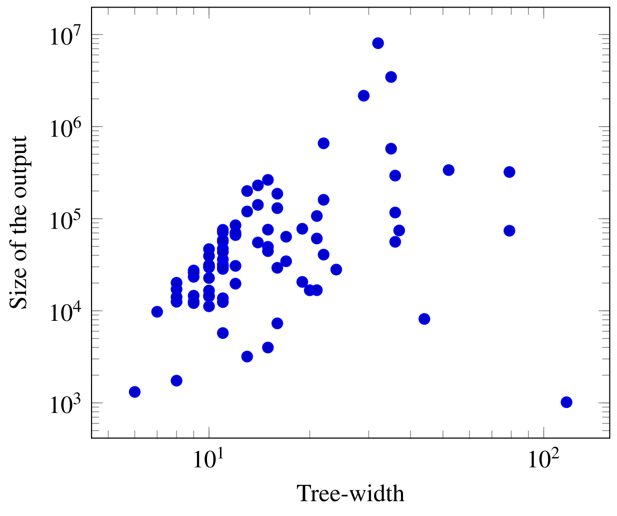

Fig. 8 Size of the output of proposed method vs. Tree-width.

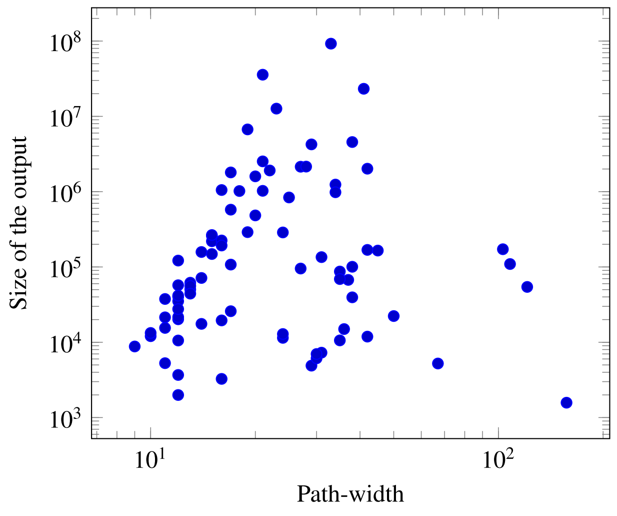

Fig. 9 Size of the output of comparative method vs. Path-width.

width, the compilation time of the proposed method is orders of magnitude smaller than that of comparative method. In the same manner, we compared the ratio of the size of the output of those two methods and tree/path-width in **Fig. 11**. The figure also shows that, when the tree-width of an input graph is small compared to the path-width, the size of the node of proposed method

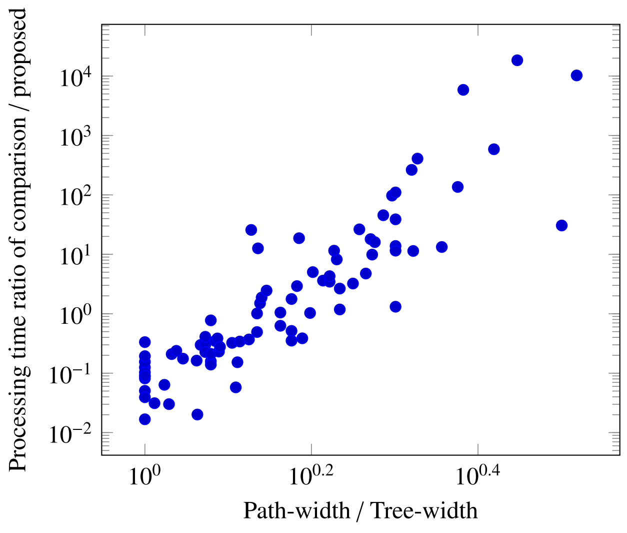

Fig. 10 Compilation time ratio of comparison method to proposed method vs. Ratio of tree-width to path-width.

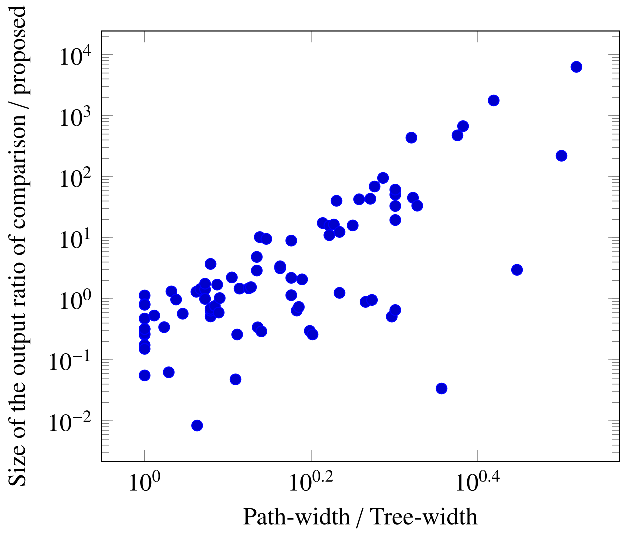

Fig. 11 Size of the output ratio of comparison method to proposed method vs. Ratio of tree-width to path-width.

has a tendency to be orders of magnitude smaller than that of comparative method.

5. Related Work

Tree Decomposition and Dynamic Programming

Tree decomposition was proposed by Halin [30] and rediscovered by Robertson and Seymour [31]. Dynamic programming on tree decomposition was intensively studied for use in optimization problems of graph substructures, such as minimum dominant set [25], edge coloring [26], and Steiner tree [27].

Algorithmic Metatheorem

Algorithmic metatheorem claims that if a graph substructure is described in a certain logic, then some problems of the graph substructure can be solved in a certain amount of delay [36]. Since Courcelle's theorem [37], many algorithmic metatheorems have been shown [38], [39], [40], [41]. Amarilli et al. [29] independently proposed a *monotone d-DNNF circuit in zero-suppressed semantics*, which uses a concept similar to that of a structured Z-d-DNNF. Then, they re-proved that the enumeration of the set of assignments in a given Monadic second order (MSO) formula with free first-order variables on a bounded tree-width structure can be performed with a constant delay between two consecutive outputs, a result which is the same as the existing results obtained by Bagan [42] or Kazana and Segoufin [43].

Frontier-based Search

Sekine et al. [44] designed an algorithm that compiles the spanning trees of a given graph into BDDs [45]. This method enumerates spanning trees with an exhaustive search on the edges of a given graph and efficiently reduces the searching space by merging or pruning the searching branches on the "elimination frontier." This method is also used to compile the maximal independent sets of the given graph [46]. Knuth [18] independently proposed the same type of method *Simpath*. He proposed an ingenious data structure called "mate" to efficiently enumerate the *s*-*t* paths or cycles of a given graph in ZDDs [19]. This method is also called a frontier-based search and it is known that it can also be used to compile matching and Steiner trees [17]. It is known that the size of the state per frontier is bounded by the path-width of the given graph [20]. The top down construction [21] and merging frontier-based search [22] methods were proposed for improving this size such that it is bounded by the branch-width.

Enumeration of Independent Sets or Cliques

The independent sets of a given graph correspond to the cliques of its complement graph. Recently, fast enumeration algorithms for maximal cliques or maximal independent sets were intensively studied. Bron and Kerbosch proposed a method to enumerate the maximum cliques of a given graph by using backtracking [47]. Tsukiyama et al. proposed a first output-polynomial algorithm that enumerates the maximal independent sets in time $O(|E||V|\mu)$ [48], where μ is the size of the maximal cliques. Chiba and Nishizeki improved on this result, achieving $O(|E|a\mu)$ [49], where a is the arboricity of the given graph. Makino and Uno proposed an algorithm for enumerating the maximal cliques based on matrix multiplication [50]. One of their algorithms can be used to enumerate the maximal cliques of G in time $O(\Delta^4\mu)$, where Δ is the maximal degree of G (Theorem 2 in Ref. [50]). Tomita et al. used the same type of pruning method as that presented by Bron and Kerbosch [47] and proposed a method for enumerating the maximal cliques in time $O(3^{|V|/3})$ [51].

Applications of Clique

A clique is known to have various applications. The term "clique" was first used in the study of close communities in social networks [52]. It is also used in the description of similarity searching in 3D databases in chemistry [53], test set compaction algorithms for combinatorial circuits [54], and comparative modeling of protein structures in bioinformatics [55].

6. Conclusion and Future Work

In this paper, we proposed a method for compiling the independent sets of a given graph. In the proposed method, a tractable language called structured Z-d-DNNF is used. By applying a previous method for tree decomposition to compute the size of the maximum independent set of a given graph, the independent sets are computed with a time complexity that is equivalent to that of the previous method. After the compilation, the sizes of the independent sets can be efficiently computed and the independent sets can be counted with dynamic programming on a structured Z-d-DNNF. In addition, the experimental results show that our algorithm runs several orders of magnitude faster than conventional frontier-based search, especially when the tree-width is small compared to the path-width.

Because methods for solving optimization problems on graph substructures are widely applied to problems other than maximum independent sets, our future work will focus on applying them to compilation. For example, it is known that an independent set is maximum if and only if it is dominant [46]. Therefore, by applying a previous method for computing the size of the minimum dominant set of a given graph [25], we could expand our method to compile the maximum independent sets of a graph. It is also our interesting future work to analyze the relationship between our method using structured Z-d-DNNF and other recent tractable language such as Zero-suppressed SDD (ZSDD) [56]. ZSDD is a subset tractable language of structured Z-d-DNNF and supports more operations than the latter. Because ZSDD supports operations that are called *apply*, which covers a variety of transformations or queries, the compilation output from ZSDD could facilitate broader applicable research than structured Z-d-DNNF.

Acknowledgments We thank Jun Kawahara for answering our question on the use of frontier-based search for compiling independent sets. We thank Yoichi Iwata for answering our question on contests for tree decomposition implementation.

References

[1] Darwiche, A. and Marquis, P.: A knowledge compilation map, *J. Artif. Intell. Res.*, Vol.17, pp.229–264 (2002).

[2] Van den Broeck, G. and Darwiche, A.: On the role of canonicity in knowledge compilation, *AAAI*, pp.1641–1648 (2015).

[3] Elliott, P. and Williams, B.C.: Dnnf-based belief state estimation. *Proc. AAAI*, pp.36–41 (2006).

[4] Huang, J. and Darwiche, A.: On compiling system models for faster and more scalable diagnosis, *Proc. AAAI*, pp.300–306 (2005).

[5] Huang, J.: Combining knowledge compilation and search for conformant probabilistic planning, *Proc. ICAPS*, pp.253–262 (2006).

[6] Chavira, M., Darwiche, A. and Jaeger, M.: Compiling relational bayesian networks for exact inference, *International Journal of Approximate Reasoning*, Vol.42, No.1-2, pp.4–20 (2006).

[7] Chavira, M. and Darwiche, A.: On probabilistic inference by weighted model counting, *Artificial Intelligence Ournal*, Vol.172, No.6-7, pp.772–799 (2008).

[8] Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B. and De Raedt, L.: Inference in probabilistic logic programs using weighted cnf's, *Proc. CoRR*, Vol.abs/1202.3719 (2012).

[9] Suciu, D., Olteanu, D., Ré, C. and Koch, C.: *Probabilistic Databases*, Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2011).

[10] Jha, A.K. and Suciu, D.: Knowledge compilation meets database theory: Compiling queries to decision diagrams, *Theory of Computing Systems*, Vol.52, No.3, pp.403–440 (2013).

[11] Rekatsinas, T., Deshpande, A. and Getoor, L.: Local structure and determinism in probabilistic databases, *Proc. SIGMOD*, pp.373–384 (2012).

[12] Beame, P., Li, J., Roy, S. and Suciu, D.: Lower bounds for exact model counting and applications in probabilistic databases, *UAI* (2013).

[13] Van den Broeck, G., Taghipour, N., Meert, W., Davis, J. and De Raedt, L.: Lifted probabilistic inference by first-order knowledge compilation, *IJCAI*, pp.2178–2185 (2011).

[14] Van den Broeck, G.: On the completeness of first-order knowledge compilation for lifted probabilistic inference, *NIPS*, pp.1386–1394

(2011).
[15] Van den Broeck, G.: *Lifted Inference and Learning in Statistical Relational Models* (*Eerste-orde inferentie en leren in statistische relationele modellen*), PhD thesis, Katholieke Universiteit Leuven, Belgium (2013).
[16] Kisa, D., Van den Broeck, G., Choi, A. and Darwiche, A.: Probabilistic sentential decision diagrams, *Proc. KR* (2014).
[17] Kawahara, J., Inoue, T., Iwashita, H. and Minato, S.: Frontier-based search for enumerating all constrained subgraphs with compressed representation, *IEICE Trans.*, Vol.100-A, No.9, pp.1773–1784 (2017).
[18] Knuth, D.E.: *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*, Addison-Wesley Professional (2011).
[19] Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems, *DAC*, pp.272–277 (1993).
[20] Inoue, Y. and Minato, S.: Acceleration of zdd construction for subgraph enumeration via path-width optimization, Technical Report TCS-TR-A-16-80 (2016).
[21] Nishino, M., Yasuda, N., Minato, S. and Nagata, M.: Compiling graph substructures into sentential decision diagrams, *AAAI*, pp.1213–1221 (2017).
[22] Sugaya, T., Nishino, M., Yasuda, N. and Minato, S.: Fast compilation of graph substructures for counting and enumeration, *Behaviormetrika*, pp.1–28 (2018).
[23] Bodlaender, H.L. and Koster, A.M.C.A.: Combinatorial optimization on graphs of bounded treewidth, *Comput. J.*, Vol.51, No.3, pp.255–269 (2008).
[24] Bodlaender, H.L., Bonsma, P.S. and Lokshtanov, D.: The fine details of fast dynamic programming over tree decompositions, *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4–6, 2013, Revised Selected Papers*, pp.41–53 (2013).
[25] Hedetniemi, S.T. and Laskar, R.: Domination in trees: Models and algorithms, *Graph Theory with Applications to Algorithms and Computer Science*, pp.423–442, John Wiley & Sons, Inc. (1985).
[26] Mitchell, S. and Hedetniemi, S.: Linear algorithms for edge-coloring trees and unicyclic graphs, *Information Processing Letters*, Vol.9, No.3, pp.110–112 (1979).
[27] Chimani, M., Mutzel, P. and Zey, B.: Improved steiner tree algorithms for bounded treewidth, *J. Discrete Algorithms*, Vol.16, No.Supplement C, pp.67–78 (2012). Selected papers from *the 22nd International Workshop on Combinatorial Algorithms* (*IWOCA 2011*).
[28] Robertson, N. and Seymour, P.D.: Graph minors. X. Obstructions to tree-decomposition, *J. Comb. Theory, Ser. B*, Vol.52, No.2, pp.153–190 (1991).
[29] Amarilli, A., Bourhis, P., Jachiet, L. and Mengel, S.: A circuit-based approach to efficient enumeration, *ICALP LIPIcs*, Vol.80, pp.111:1–111:15, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017).
[30] Halin, R.: S-functions for graphs, *Journal of Geometry*, Vol.8, No.1-2, pp.171–186 (1976).
[31] Robertson, N. and Seymour, P.D.: Graph minors. III. planar tree-width, *Journal of Combinatorial Theory, Ser. B*, Vol.36, No.1, pp.49–64 (1984).
[32] Kloks, T.: *Treewidth: Computations and Approximations*, Vol.842, Springer Science & Business Media (1994).
[33] Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M. and Saurabh, S.: *Parameterized Algorithms*, Springer (2015).
[34] Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases, *IJCAI*, pp.819–826 (2011).
[35] Kinnersley, N.G.: The vertex separation number of a graph equals its path-width, *Information Processing Letters*, Vol.42, No.6, pp.345–350 (1992).
[36] Ishihata, M., Maehara, T. and Rigaux, T.: Algorithmic meta-theorems for monotone submodular maximization, *CoRR*, Vol.abs/1807.04575 (2018).
[37] Courcelle, B.: The monadic second-order logic of graphs. i. recognizable sets of finite graphs, *Information and Computation*, Vol.85, No.1, pp.12–75 (1990).
[38] Seese, D.: Linear time computable problems and first-order descriptions, *Mathematical Structures in Computer Science*, Vol.6, No.6, pp.505–526 (1996).
[39] Bagan, G.: MSO queries on tree decomposable structures are computable with linear delay, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25–29, 2006, Proceedings*, pp.167–181 (2006).
[40] Kazana, W. and Segoufin, L.: First-order query evaluation on structures of bounded degree, *Logical Methods in Computer Science*, Vol.7, No.2 (2011).
[41] Arnborg, S., Lagergren, J. and Seese, D.: Easy problems for tree-decomposable graphs, *J. Algorithms*, Vol.12, No.2, pp.308–340 (1991).
[42] Bagan, G.: Mso queries on tree decomposable structures are computable with linear delay, *International Workshop on Computer Science Logic*, pp.167–181, Springer (2006).
[43] Kazana, W. and Segoufin, L.: Enumeration of monadic second-order queries on trees, *ACM Trans. Computational Logic*, Vol.14, No.4, pp.25:1–25:12 (2013).
[44] Sekine, K., Imai, H. and Tani, S.: Computing the tutte polynomial of a graph of moderate size, *ISAAC*, pp.224–233 (1995).
[45] Bryant, R.E.: Graph-based algorithms for boolean function manipulation, *IEEE Trans. Computers*, Vol.35, No.8, pp.677–691 (1986).
[46] Hayase, K., Sadakane, K. and Tani, S.: Output-size sensitiveness of obdd construction through maximal independent set problem, *International Computing and Combinatorics Conference*, pp.229–234, Springer (1995).
[47] Bron, C. and Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph, *Comm. ACM*, Vol.16, No.9, pp.575–577 (1973).
[48] Tsukiyama, S., Ide, M., Ariyoshi, H. and Shirakawa, I.: A new algorithm for generating all the maximal independent sets, *SICOMP*, Vol.6, No.3, pp.505–517 (1977).
[49] Chiba, N. and Nishizeki, T.: Arboricity and subgraph listing algorithms, *SICOMP*, Vol.14, No.1, pp.210–223 (1985).
[50] Makino, K. and Uno, T.: New algorithms for enumerating all maximal cliques, *SWAT*, Vol.3111, pp.260–272, Springer (2004).
[51] Tomita, E., Tanaka, A. and Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments, *Theor. Comput. Sci.*, Vol.363, No.1, pp.28–42 (2006).
[52] Luce, R.D. and Perry, A.D.: A method of matrix analysis of group structure, *Psychometrika*, Vol.14, No.2, pp.95–116 (1949).
[53] Rhodes, N., Willett, P., Calvet, A., Dunbar, J.B. and Humblet, C.: Clip: Similarity searching of 3D databases using clique detection, *J. Chem. Inf. Comput. Sci.*, Vol.43, No.2, pp.443–448 (2003).
[54] Hamzaoglu, I. and Patel, J.H.: Test set compaction algorithms for combinational circuits, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.19, No.8, pp.957–963 (2000).
[55] Samudrala, R. and Moult, J.: A graph-theoretic algorithm for comparative modeling of protein structure, *J. Mol. Biol.*, Vol.279, No.1, pp.287–302 (1998).
[56] Nishino, M., Yasuda, N., Minato, S. and Nagata, M.: Zero-suppressed sentential decision diagrams, *AAAI*, pp.1058–1066 (2016).

Teruji Sugaya is currently a PH.D. student at the Graduate School of Information Science and Technology, Hokkaido University, Japan. He received his B.S. degree in Public Law from the University of Tokyo in 1995 and his M.S. degree in Informatics from the Open University of Japan in 2015. His research interests include knowledge compilation, graph algorithms, and probabilistic graphical models. He is a member of IPSJ and IEICE.

Masaaki Nishino is a Research Scientist at Nippon Telegraph and Telephone Corporation (NTT) Communication Science Laboratories. He received his B.E., M.E., and Ph.D. degrees in informatics from Kyoto University in 2006, 2008, and 2014, respectively. He joined NTT in 2008. His current research interests include data structures, natural language processing, and combinatorial optimization. He is a member of IPSJ, JSAI, and ANLP.

Norihito Yasuda received his B.S. degree in integrated human studies and his M.S. degree in human and environmental studies from Kyoto University, Kyoto, Japan, and the D.Eng. degree in computational intelligence and system science from the Tokyo Institute of Technology, Japan, in 1997, 1999, and 2011, respectively. He is a Senior Researcher with Nippon Telegraph and Telephone Corporation Laboratories, Tokyo, Japan, where he has worked since 1999. He was a Research Associate Professor with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan in 2015. His current research interests include discrete algorithms and natural language processing.

Shin-ichi Minato is a Professor at the Graduate School of Informatics, Kyoto University. He received his B.E., M.E., and D.E. degrees in information science from Kyoto University in 1988, 1990, and 1995, respectively. He worked for NTT Laboratories from 1990 until 2004. He was a Visiting Scholar at the Computer Science Department of Stanford University in 1997. He joined Hokkaido University as an Associate Professor in 2004 and has been a Professor since October 2010. He became a Professor at Kyoto University in April 2018 (present position). His research interests include efficient representations and manipulation algorithms for large-scale discrete structures, such as Boolean functions, sets of combinations, sequences, and permutations. He served as a Research Director of JST ERATO MINATO Discrete Structure Manipulation System Project from 2009 to 2016. He is a senior member of IEICE and IPSJ and is a member of IEEE and JSAI.