HOKKAIDO UNIVERSITY

Title	Studies on New Tractable Indexing Structures and Rapid Compilation Methods for Graph-Substructures
Author(s)	菅谷, 輝治
Degree Grantor	北海道大学
Degree Name	博士(情報科学)
Dissertation Number	甲第14282号
Issue Date	2020-09-25
DOI	https://doi.org/10.14943/doctoral.k14282
Doc URL	https://hdl.handle.net/2115/79538
Type	doctoral thesis
File Information	Teruji_Sugaya.pdf

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

Thesis submitted to obtain the title
Doctor of Information Science
令和２年度　博士学位論文

Studies on New Tractable Indexing Structures and Rapid Compilation Methods for Graph-Substructures

グラフ部分構造を扱う新規索引構造と
高速コンパイル法に関する研究

Teruji Sugaya

菅谷 輝治

Division of Computer Science and Information Technology,
Graduate School of Information Science and Technology
Hokkaido University

北海道大学 大学院情報科学研究科
情報理工学専攻

July 2020

Abstract

Knowledge compilation has been intensely studied in the field of artificial intelligence. Knowledge compilation is, roughly describing, take some data with "certain" property, then change it into a data structure, which supports varieties of queries or transformation. Here, the data structure is called *tractable indexing structure* and the process to change the data into tractable indexing structure is called *compilation*. Among the tractable indexing structures, BDD is the most fundamental and has been studied for a long time.

Recently, there have been two directions in the subsequent studies based on BDDs.

The first direction is to enable succinctness. Darwiche et al. presented a knowledge compilation map. In knowledge compilation map, some superset structures of BDD are proposed. While the index size of BDDs are bounded with path-width of the input CNF, these superset structures in knowledge compilation map is bound with tree-width of the input CNF.

The second direction is the application to graph-substructure. ZDD is a tractable indexing structure that changed the default value of BDD from "do not care" to "does not exist". This property of ZDD called *zero-suppression* makes ZDD to be appropriate to represent a set of family, especially graph-substructures. As an efficient compilation method of ZDD, frontier-based search is known.

An important issue of today is how to redesign ZDD and frontier-based search to achieve more efficient compilation of graph-substructures. ZSDD is a remarkable preceding work to cope with this task. ZSDD is an extension of ZDD and more succinct than ZDD in general. As a result, ZSDD's compilation method runs faster than that of ZDD's. However, since ZSDD is strongly structured, its compilation method become more complicated than ordinal frontier-based search. As a result, the overhead of computation is inevitable. The key issue to make the compilation much faster is how to reduce the overhead of computation.

In this thesis, we introduce a new tractable indexing structure and a compilation method for graph-substructures. Our tractable indexing structure is called *structured Z-d-DNNF* and its compilation method is called *merging frontier-based search*. Since our structure is less structured than ZSDD, the capability of our structure to support

query or transformation is lower than that of ZSDD. However, as trade-off, its simple structure makes the compilation method more efficient. With the theoretical analysis and the experimental result, we show that our compilation method runs faster than the previous method in many cases.

In Chapter 3, we introduce our structure, structured Z-d-DNNF. We introduced zero-suppression to structured Z-d-DNNF same as ZDDs or ZSDDs. In addition, we present a new system of tractable indexing structure called zero-suppressed knowledge compilation map. It is a system of structures that has adopted zero-suppression to the whole structures of knowledge compilation map. Zero-suppressed knowledge compilation map has ZDD as basis instead of BDD.

In Chapter 4 and Chapter 5, we propose our compilation method, merging frontier-based search. Merging frontier-based search is an extension of conventional frontier-based search. Unlike ordinal ZDD, we must "merge" the states of searching branches to compile structured Z-d-DNNF. We apply our compilation method to the independent set in Chapter 4 and *s*-*t* paths in Chapter 5.

Finally, in Chapter 6, we conclude this thesis and discuss the future work.

Contents

1 Introduction **1**

1.1 Background 1
1.2 Our Contributions 3
1.3 Related Work 4
1.3.1 Knowledge Compilation Map 4
1.3.2 Tree Decomposition and Dynamic Programming 5
1.3.3 Frontier-based Search 5
1.3.4 Enumeration of Independent Sets or Cliques 5
1.3.5 Applications of Clique 6
1.3.6 Path Counting and Enumeration 6
1.3.7 Algorithmic metatheorem 6
1.4 Thesis Organization 6

2 Preliminaries **9**

2.1 Graph and its Decomposition 10
2.1.1 Graph 10
2.1.2 Tree Decomposition 10
2.1.3 Dynamic Programming Using Tree Decomposition 11
2.1.4 Branch Decomposition 13
2.2 Tractable Indexing Structure 13
2.2.1 BDDs 14
2.2.2 Knowledge Compilation Map 15
2.2.3 Structured d-DNNF 16
2.2.4 SDDs 19
2.3 Tractable Indexing Structure with Zero-suppression 20
2.3.1 Indicator Function and a Family of Sets 21
2.3.2 ZDDs 21
2.3.3 ZSDDs 22
2.4 Compilation 25
2.4.1 Frontier-based Search 25
2.4.2 Compilation of *s*-*t* paths 26

2.4.3 Compilation of independent sets 27

3 Structured Z-d-DNNF and Zero-suppressed Knowledge Compilation Map 31

3.1 Structured Z-d-DNNF 31
3.2 Query and Transformation supported by Structured Z-d-DNNF 34
3.2.1 Query 34
3.2.2 Transformation 35
3.3 Zero-suppressed Knowledge Compilation Map 35
3.4 Concluding Remarks 37

4 Merging Frontier-Based Search to Compile Independent Sets 39

4.1 Overview 40
4.2 Dynamic Programming 40
4.3 Vtree Construction from a Nice Tree Decomposition 42
4.4 Compilation with Structured Z-d-DNNF 43
4.5 Correctness and Complexity 47
4.6 Experiments 48
4.7 Concluding Remarks 53

5 Merging Frontier-Based Search to Compile *S*-*T* Paths 57

5.1 Overview 57
5.2 Frontier and Configuration 58
5.3 Merging Frontier-based Search for *S*-*T* Paths 60
5.4 Merging mates 62
5.5 Correctness and Complexities 67
5.6 Experiments 68
5.7 Discussion 69
5.8 Concluding Remarks 71

6 Conclusions and Future Work 77

6.1 Conclusions 77
6.2 Future Work 78

Acknowledgements 81

Bibliography 82

Chapter 1

Introduction

1.1 Background

In the academic field of artificial intelligence, *knowledge compilation* has been studied for years. Knowledge compilation is, roughly describing, take some data with "certain" property, then change it into a data structure, which supports varieties of queries or transformation. Here, the data structure is called *tractable indexing structure* and the process to change the data into tractable indexing structure is called *compilation*.

In the history of knowledge compilation, binary decision diagrams (BDDs) has played an important role and become the basis of knowledge compilation. BDD is a data structure to represent a Boolean function compactly based on Shannon decomposition. In practice, as a special type of BDD, reduced ordered binary decision diagram (ROBDD) [Bryant, 1986a] is frequently used. The paper of BOBDD is known as one of the most-cited papers in computer science and it has more than 10,000 citation. [1] In this thesis, we use the term BDD to refer to ROBDD.

Recently, there have been two directions in the subsequent studies based on BDDs.

The first direction is to enable succinctness. Darwiche et al. proposed a series of tractable indexing structures which are collectively called *knowledge compilation map* [Darwiche and Marquis, 2002, Darwiche, 2014]. In a knowledge compilation map, some superset structures of BDDs are proposed. In stead of being based on Shannon decomposition, these structures are based on $(\boldsymbol{X},\boldsymbol{Y})$-Decomposition. $(\boldsymbol{X},\boldsymbol{Y})$-Decomposition is an extension of Shannon decomposition. While Shannon decomposition is a method to decompose Boolean function with linear order, $(\boldsymbol{X},\boldsymbol{Y})$-Decomposition is a method with tree structure. Consequently, the size of Shannon decomposition is bound with path-width of the input CNF, whereas the size of $(\boldsymbol{X},\boldsymbol{Y})$-Decomposition is bound with tree-width of the CNF [Inoue and Minato, 2016].

A Knowledge compilation map is a system of tractable indexing structures, which

[1] Based on Google scholar search in November 4th, 2019.

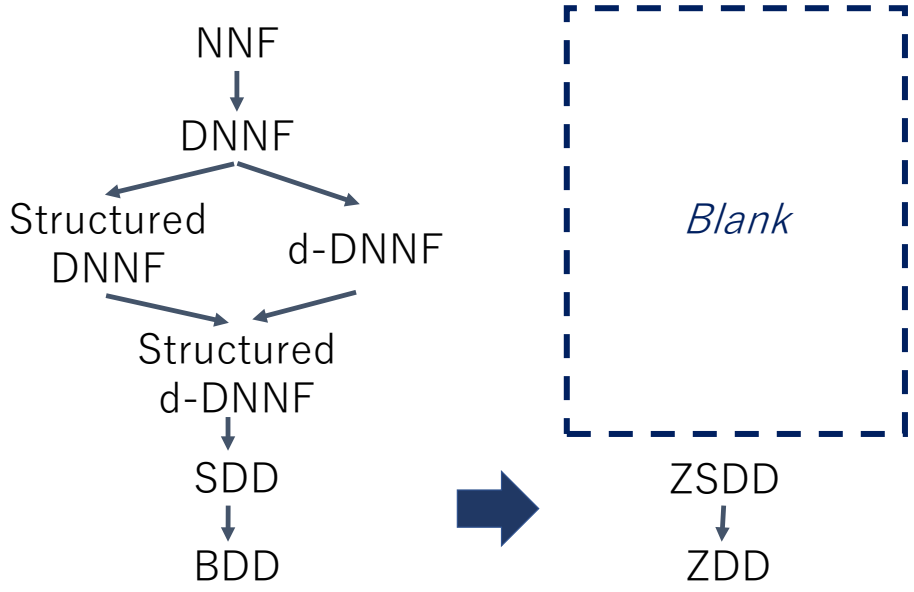


Figure 1.1. Knowledge compilation map, ZDD, and ZSDD.
In the right side, zero-suppressed ones lack some of the equivalent indexing structures.

contains BDDs and its upper layer structures. [2] At the top of the system, NNF is positioned, which is followed by DNNF and its descendants. The system is shown in the left side of Fig. 1.1. As descending the system below, a structure becomes more powerful and support more of queries. Especially, SDD supports varieties of operation called *apply* and it support the same types of operations with BDD. Besides, as descending the system below, a structure become more structured. As a result, the compilation method of the structure tends to be more complicated and the overhead of the calculation is inevitable.

The second direction is the application to a family of sets, especially graph-substructures. BDDs has a variation which is called zero-suppressed decision diagrams (ZDDs) [Minato, 1993]. Minato introduced *zero suppression* to BDDs and proposed a new tractable indexing structure which is suitable to represent a family of set. That is, a tractable indexing structure is said to be zero suppressed when "does not exist" is adopted as a default value, instead of "don't care". In stead of Boolean function, ZDDs are widely applied towards a family of sets, especially graph-substructures. Graph-substructures, such as, *s-t* paths, independent sets, and matching has a broad application and they are important compilation targets with ZDDs [Inoue et al., 2015, Maehara et al., 2017].

[2] In the article of knowledge compilation map, Darwiche studied other than DNNF and its descendants, such as CNF and DNF [Darwiche and Marquis, 2002]. For simplicity, this thesis focus on NNF, DNNF, and its subset when we refer to a knowledge compilation map.

For the compilation of graph-substructure with ZDDs, a method called frontier-based search is proposed by Knuth [Knuth, 2011]. Knuth proposed an ingenious data structure called "mate" to efficiently manage the states of searching branches and consequently realized an efficient compilation method. To compile the *s-t* paths in a 11×11 grid graph, a naive brute force method spends almost 29 billion years, whereas frontier-based search finishes the calculation within a few seconds[3].

However, to the best we know, an approach to exploit tree structure is rarely applied to compile the graph-substructures. The rare exception is Zero-Suppressed Sentential Decision Diagrams (ZSDDs) [Nishino et al., 2016]. Nishino et al., introduced zero-suppression to SDDs and proposed a new tractable indexing structure called ZSDDs. In addition, they proposed a compilation method for ZSDDs called top-down construction [Nishino et al., 2017]. Top-down construction is reported that it runs at most several magnitudes faster than conventional frontier-based search. Since ZSDD is a variation of SDD, it supports the same types of apply operations with SDD or BDD. However, as a trade-off, the method for the compilation become more complicated than conventional frontier-based search with ZDDs, thus the overhead is inevitable. If we can compile with a more simplified and less structured structure than ZSDD, the compilation would become more faster.

1.2 Our Contributions

Our contributions are as follows.

Firstly, we propose a new tractable indexing structure called *structured Z-d-DNNF*. Structured Z-d-DNNF is a variation of structured d-DNNF, which is originally proposed by Pipatsrisawat et al [Pipatsrisawat and Darwiche, 2008]. We introduce zero-suppression to structured d-DNNF, consequently, we create a new tractable indexing structure that is suitable to represent graph-substructure same as ZDD or ZSDD. Also same as ZSDD, structured Z-d-DNNF is based on $(\boldsymbol{X}, \boldsymbol{Y})$-decomposition. Therefore, the size of the structure is bound with the tree-width of an input graph. As a result, it become more succinct than ZDDs. Moreover, with exploiting its simplified and less structured syntax than ZSDD, we can realize a more efficient compilation method than that of ZSDD.

In addition, we propose a new system of tractable indexing structure called *zero-suppressed knowledge compilation map* (Z-KCM). We introduce zero-suppression to the whole system of knowledge compilation map of Darwiche and create a new system of tractable indexing structure suitable for the compilation of graph-substructure. In the map, ZDDs and ZSDDs are positioned at the most lower layer of the system. In

[3] https://www.youtube.com/watch?v=Q4gTV4r0zRs

addition, our structure structured Z-d-DNNF is positioned as a upper layer structure of ZDDs and ZSDDs in a system. We briefly explain the syntax and semantics of the structure system in this thesis. The whole systems of knowledge compilation map is shown in the right side of Fig. 1.1.

Secondly, we propose a new compilation method called *merging frontier based search*. Merging frontier-based search is an extension of conventional frontier-based search. Since structured Z-d-DNNF is based on $(\boldsymbol{X}, \boldsymbol{Y})$-decomposition, we have to "merge" the states of searching branches, which is not seen in conventional frontier-based search. We propose an efficient merging method for the compilation with structured Z-d-DNNF and realize a more efficient compilation method than its preceding algorithms. In this thesis, we apply merging frontier-based search towards the compilation both of *s-t* paths and independent sets in a graph. On both cases, we propose a new merging method necessary for the compilation. For the compilation of the *s-t* paths, we compared our method with top-down construction. The experimental results showed our compilation method runs at most twenty times faster than top-down construction of ZSDDs. On the other hand, for the compilation of the independent sets, we compared our method with conventional frontier-based search. Especially when the tree-width of the input graph is smaller than the path-width, our method runs several magnitude faster than a conventional frontier-based search.

1.3 Related Work

1.3.1 Knowledge Compilation Map

Darwiche et al. developed a system of tractable indexing structures known as knowledge compilation map, which has NNF at the top level [Darwiche and Marquis, 2002, Bordeaux et al., 2014]. The NNF has subsets that have the following conditions: decomposability [Darwiche, 1999, Darwiche, 2001a] and determinism [Darwiche, 2001b]. The NNF that satisfies decomposability is referred to as DNNF, whereas the NNF that satisfies determinism is referred to as d-NNF. In addition, the NNF that satisfies both is referred to as d-DNNF. DNNF(d-DNNF) is referred to as structured DNNF (structured d-DNNF) when it respects a vtree [Pipatsrisawat and Darwiche, 2008]. Vtree is a binary tree to recursively $(\boldsymbol{X}, \boldsymbol{Y})$-decompose a Boolean function. Furthermore, when a vtree represents recursive $(\boldsymbol{X}, \boldsymbol{Y})$-partition, a structure is referred to as SDD [Darwiche, 2011a]. SDD is a superset of OBDD [Bryant, 1986a]. For those subsets of NNF, supported queries and transformations have been studied [Darwiche and Marquis, 2002, Bordeaux et al., 2014].

Various methods for compiling from CNF to each subset of NNF have been proposed. Darwiche proposed a compiling method for d-DNNF using the dtree of the

input CNF [Darwiche, 2001b]. Furthermore, he demonstrated a more efficient compilation using a new method of caching and applying DPLL, which is a method of SAT solver [Darwiche, 2004]. Since structured DNNF supports the conjoin operation, CNF can be compiled in a bottom up manner [Pipatsrisawat and Darwiche, 2008]. Pipatsrisawat et al. demonstrated a top down compilation of structured DNNF using a new caching method [Pipatsrisawat and Darwiche, 2010]. Since apply is an operation in a bottom up manner, the bottom up compilation of SDDs is applicable [Darwiche, 2011a]. To realize a top down compilation of SDDs, Oztok et al. made use of unit resolution and clause learning, which are methods of SAT solver [Oztok and Darwiche, 2015].

1.3.2 Tree Decomposition and Dynamic Programming

Tree decomposition was proposed by Halin [Halin, 1976] and rediscovered by Robertson and Seymour [Robertson and Seymour, 1984]. Dynamic programming on tree decomposition was intensively studied for use in optimization problems of graph-substructures, such as minimum dominant set [Hedetniemi and Laskar, 1985], edge coloring [Mitchell and Hedetniemi, 1979], and Steiner tree [Chimani et al., 2012].

1.3.3 Frontier-based Search

Sekine et al. [Sekine et al., 1995] designed an algorithm that compiles the spanning trees of a given graph into BDDs [Bryant, 1986b]. This method compiles the spanning trees with an exhaustive search on the edges of a given graph. Sekine et al. efficiently reduced 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 [Hayase et al., 1995]. Knuth [Knuth, 2011] independently proposed the same type of method *Simpath*. He proposed an ingenious data structure called "mate" to efficiently compile the *s*-*t* paths or cycles of a given graph into ZDDs [Minato, 1993]. This method is also called a frontier-based search and it is known that it can be also be used to compile matching or Steiner trees [Kawahara et al., 2017]. It is known that the size of the state per frontier is bounded by the path-width of the given graph [Inoue and Minato, 2016]. The top-down construction [Nishino et al., 2017] and merging frontier-based search [Sugaya et al., 2018] methods were proposed for improving this size to be bounded by the branch-width.

1.3.4 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 indepen-

dent sets were intensively studied. Bron and Kerbosch proposed a method to enumerate the maximum cliques of a given graph by using backtracking [Bron and Kerbosch, 1973]. Tsukiyama et al. proposed a first output-polynomial algorithm that enumerates the maximal independent sets in time $O(|E||V|\mu)$ [Tsukiyama et al., 1977], where μ is the size of the maximal cliques. Chiba and Nishizeki improved this result and achieved $O(|E|a\mu)$ [Chiba and Nishizeki, 1985], where a is the arboricity of the given graph. Makino and Uno proposed an algorithm for enumerating the maximal cliques based on matrix multiplication [Makino and Uno, 2004]. 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 [Makino and Uno, 2004]). Tomita et al. used the same type of pruning method as that presented by Bron and Kerbosch [Bron and Kerbosch, 1973] and proposed a method for enumerating the maximal cliques in time $O(3^{|V|/3})$ [Tomita et al., 2006].

1.3.5 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 [Luce and Perry, 1949]. It is also used in the description of similarity searching in 3D databases in chemistry [Rhodes et al., 2003], test set compaction algorithms for combinatorial circuits [Hamzaoglu and Patel, 2000], and comparative modeling of protein structures in bioinformatics [Samudrala and Moult, 1998].

1.3.6 Path Counting and Enumeration

An *s-t* paths counting problem in a graph is known to be #P-complete [Valiant, 1979]. As a study of s-t paths enumeration other than frontier-based search, Uno proposed "Cypath" to enumerate the chordless *s-t* paths and chordless cycles in a graph [Uno, 2003].

1.3.7 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 [Ishihata et al., 2018]. Since Courcelle's theorem [Courcelle, 1990], many algorithmic metatheorems has been shown [Seese, 1996, Bagan, 2006a, Kazana and Segoufin, 2011, Arnborg et al., 1991]. Amarilli et al. [Amarilli et al., 2017] proposed a *monotone d-DNNF circuit in zero-suppressed semantics*, which independently uses a concept similar to that of a structured Z-d-DNNF. 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. This result is the same as the existing results obtained by Bagan [Bagan, 2006b] or Kazana and Segoufin [Kazana and Segoufin, 2013].

1.4 Thesis Organization

In the following Chapter 2, we introduce definitions and notations about a graph, its decomposition, tractable indexing structures, and its compilation methods. Chapter 3 provides a new tractable indexing structure, structured Z-d-DNNF and a new system of structure, zero-suppressed knowledge compilation map (zero-suppressed knowledge compilation map). In the following Chapter 4 and 5, we explain a new method of compilation using structured Z-d-DNNF. Chapter 4 applies our compilation method to the independent sets in a graph, while Chapter 5 applies our method to *s*-*t* paths in a graph. In Chapter 6, we summarize the results in this thesis and explain the future work.

Chapter 2

Preliminaries

In this chapter, we introduce required definitions and notations to describe the contributions in this thesis. Firstly, we introduce graph and its decomposition, both of which are inputs of our method and also the latter is a tool for compilation. Secondly, we introduce tractable indexing structure that is a data structure used in the compilation. Finally, we introduce compilation method, that is, frontier-based search with its two targets, *s-t* paths and independent sets.

For the reader's convenience, the notations frequently used in this paper are listed in

Table 2.1. Frequently used notation.

Notation	Description	Notation	Description
$G=(V,E)$	Simple undirected graph.	d,d_α	Node of a structured Z-d-DNNF (dnode)
$\|V\|,\|E\|$	Size of the vertices and edges of G.		and one respecting vnode α.
$v_1,v_2,\ldots$	Vertices of G (gnodes).	$\bot,\varepsilon,X,\pm X$	Terminal dnodes representing $\emptyset,\{\emptyset\},\{X\},\{\{X\},\emptyset\}$.
$u_1,u_2,\ldots$		$h^i_\alpha=(d^i_{\alpha^l},d^i_{\alpha^r})$	Element with $d^i_{\alpha^l}$ and $d^i_{\alpha^r}$
$G[A]$	Subgraph induced by edge set A of G.	$d_\alpha=\{(d^1_{\alpha^l},d^1_{\alpha^r}),\ldots,(d^n_{\alpha^l},d^n_{\alpha^r})\}$	Internal dnode with n elements
$P(s,t)$	Set of *s-t* paths.	D_α	Set of dnodes respecting vnode α.
BW	Branch-width.	$\langle d\rangle$	Family of sets represented by d.
$\mathcal{T}$	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.
B_i	Bag of a tnode $i\in T(\mathcal{T})$.		that contains the entire gnode set P_i and none of Q_i.
V_i	Union of B_i and its children.	$\mathbb{S}(i)$	Independent sets of G_i.
G_i	The subgraph induced by V_i.	$\mathbb{S}(i;P_i,Q_i)$	Independent sets of G_i that contain the entire P_i.
TW	Tree-width.		and none of Q_i.
$\alpha,\beta,\ldots$	Nodes of a vtree (vnodes)	F_α	Frontier gnode set of a vnode α.
α^l,β^r	Left and right children of vnode α.	$mate[u]$	Mate of gnode u.
e_α	An element corresponding to a leaf α.	$mate_{\langle d\rangle}[u]$	Mate of gnode u in the family of set $\langle d\rangle$.
E_α	Variables corresponding to leaves	$\phi(\langle d_\alpha\rangle)$	Configuration of dnode d_α respecting a vnode α.
	of a vtree rooted from vnode α.		
2^{E_α}	Power set with universe E_α.		

2.1 Graph and its 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. *Path* is a trail of vertices in which an edge exists between a vertex and the next. *Simple path* is a path that contains no repeated vertices. 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. For $A \subseteq E$, we represent a subgraph induced by A as $G[A]$.

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 2.1.1 (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 [Halin, 1976, Robertson and Seymour, 1984].*

(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

$$B_i \cap B_k \subseteq B_j. \tag{2.1}$$

The width of the tree decomposition is

$$\max_{i \in T(\mathcal{T})} (|B_i| - 1). \tag{2.2}$$

We say graph G has tree-width TW, if TW is the minimum width of 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* [Kloks, 1994] to facilitate the design of dynamic programming.

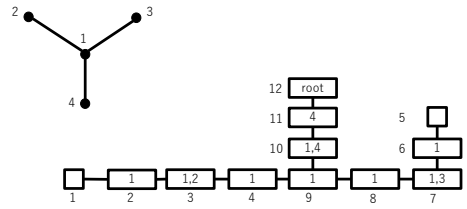


Figure 2.1. Example of a graph and its nice tree decomposition.

Definition 2.1.2 (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 TW, a nice tree decomposition of tree-width TW can be obtained in $O(W^2 \cdot max(|T(\mathcal{T})|, |V|))$ [Cygan et al., 2015]. We show an example of a graph and its nice tree decomposition in Fig. 2.1. For the sake of simplicity, hereafter we represent a (nice) tree decomposition $(\mathcal{B}, \mathcal{T})$ by $\mathcal{T}$.

2.1.3 Dynamic Programming Using Tree Decomposition

In this section, we explain dynamic programming using tree decomposition. Dynamic programming using tree decomposition has been intensely studied and numerous studies have focused on counting or optimizing graph-substructures. For example, the maximum independent set [Bodlaender and Koster, 2008, Bodlaender et al., 2013], minimum dominating set [Hedetniemi and Laskar, 1985], coloring [Mitchell and Hedetniemi, 1979], and minimum Steiner tree [Chimani et al., 2012] have been studied. Here, we focus on the algorithm to find the maximum independent set. That is because, as described later in Section 4.2, we apply this method to the compilation of the independent sets in a graph.

The nodes of a tree decomposition satisfy the property shown in the following Lemma 2.1.1, which plays an important role in the design of dynamic programming using tree decomposition.

Lemma 2.1.1 (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 $B_k \setminus B_j$ [Cygan et al., 2015].*

Before offering a formal explanation of dynamic programming, 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. Consequently, 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 independent set.

Here, the key is that the problem can be solved by locally checking the adjacency of gnodes. That is, because of the separation Lemma 2.1.1, it is sufficient to check the adjacency of the gnodes mapped with 1 on each tnode

The method is formally explained as follows. 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:[1]

$$s(i; P_i, Q_i) = max \left\{ |S_i| \;\middle|\; \begin{array}{l} S_i \subseteq V_i \\ S_i \text{ is an independent set} \\ P_i \subseteq S_i, Q_i \cap S_i = \emptyset, P_i \cup Q_i = B_i \end{array} \right\}. \tag{2.3}$$

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 $\mathcal{T}$, which process is categorized according to the types of nodes. Consequently, $s(root)$ equals the size of the maximum independent set. Here, the child of an introduce or forget tnode i is j, whereas 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, whereas supposed that $B_i = B_j \setminus \{w\}$ when i is a forget node.

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

[1] 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`

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 $u \in Q_i$.

$$s(i;P_i,Q_i) = \begin{cases} -\infty & (I) \\ s(j;P_i \setminus \{u\},Q_i)+1 & (II) \\ s(j;P_i,Q_i \setminus \{u\}) & (III) \end{cases} . \tag{2.4}$$

3. i is a forget node:

$$s(i;P_i,Q_i) = max\{s(j;P_i \cup \{w\},Q_i), s(j;P_i,Q_i \cup \{w\})\}. \tag{2.5}$$

4. i is a join node:

$$s(i;P_i,Q_i) = s(j;P_i,Q_i) + s(k;P_i,Q_i) - |P_i|. \tag{2.6}$$

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 [Bodlaender et al., 2013]. Thus, the time complexity of this computation for the entire node is $O(2^W|\mathcal{T}|)$.

2.1.4 Branch Decomposition

Branch decomposition is a pair (T,τ), where T is a ternary tree and τ is a bijection from the set of leaves of T to the set of edges of G [Robertson and Seymour, 1991]. When excluding a certain edge e from T, T is divided into two connected components. Let E_1 and E_2 be the edge sets included in one component and the other, respectively. Then, G is divided into two subgraphs, $G[E_1]$ and $G[E_2]$, where $G[E_i]$ is the subgraph induced by edge set E_i of G. This partition is called an *e-separation*, and the set of vertices shared by these two subgraphs is called *e-separator*. The maximum size of e-separator is called the *width* of branch decomposition. The minimum width among all branch decompositions of a given graph G is called *branch-width*.

2.2 Tractable Indexing Structure

In this section, we explain tractable indexing structure. Firstly, we explain BDDs which are most commonly used tractable indexing structure in knowledge compilation. Secondly, we explain a knowledge compilation map, which is a system of tractable indexing structure and consists of BDDs and its upper layer structures. Thirdly, we explain structured d-DNNF, which is a tractable indexing structure that is the basis of our proposed structure. Finally, we explain SDDs, which is more powerful but more structured structure than structured d-DNNF.

2.2.1 BDDs

Binary decision diagram (BDD) is a tractable indexing structure that represents a Boolean function. BDDs represent a Boolean function as a DAG that shows recursive Shannon decomposition of the function.

BDDs consist of one or more nodes. Each node corresponds to a Boolean function. To a node b, we let $\langle b \rangle$ be the Boolean function corresponding to the node.

Prior to the definition of BDD, a linear order $P = (x_1, x_2, \ldots, x_n)$ is defined with a set of propositional variables $\{x_1, x_2, \ldots, x_n\}$. Here, $P_i := (x_i, x_{i+1}, \ldots, x_n)$ is denoted by $i \geq 1$. BDDs respecting a linear order P is inductively defined as follows [Bryant, 1986a].

Definition 2.2.1. *BDDs that respect a linear order P is one of the following.*

1. $b = \top$ *or* $b = \bot$.
   Semantics: $\top = true$ *and* $\bot = false$.

2. $b = (x_i, b_j^{\mathrm{LO}}, b_j^{\mathrm{HI}})$, *where* $i \geq 1$. *And* $b^{\mathrm{LO}}(b^{\mathrm{HI}})$ *is a BDD that respects a linear order* P_j, *where* $j > i$.
   Semantics: $\langle b \rangle = (x_i \wedge \langle b^{\mathrm{HI}} \rangle) \vee \langle b^{\mathrm{LO}} \rangle$.

The former is called a leaf node and the latter is called an internal node. When BDDs meets the following, they are said to be reduced.

- *In arbitrary two distinct internal nodes* $(x_i, b_j^{\mathrm{LO}}, b_j^{\mathrm{HI}})$ *and* $(x_k, b_l^{\mathrm{LO}}, b_l^{\mathrm{HI}})$, $(b_j^{\mathrm{LO}}, b_j^{\mathrm{HI}}) \neq (b_l^{\mathrm{LO}}, b_l^{\mathrm{HI}})$.
- $b_j^{\mathrm{LO}} \neq b_j^{\mathrm{HI}}$ *in all internal node* $(x_i, b_j^{\mathrm{LO}}, b_j^{\mathrm{HI}})$.

The former rule is called node sharing and the latter is called node deletion.

In the reduced BDDs, $\langle b \rangle = \langle b' \rangle \Rightarrow b = b'$. That is, semantically equal BDDs are also syntactically identical. As we explain later in Section 2.3.2, the rule of node deletion of ZDDs is different from BDDs.

The outline of a diagram of BDDs is shown in Fig. 2.2. A root bdd node x_i has two children nodes, b_j^{LO} and b_j^{HI}. Fig. 2.3 shows an example of BDDs that represent a Boolean function $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee x_3$. The outline of node sharing and node deletion are shown in Fig. 2.4 and Fig. 2.5 respectively.

BDDs support varieties of operations that is called “apply”. Apply offers basic operations, such as conjunction, disjunction, and negation, which can be realized by polynomial-time graph manipulation.

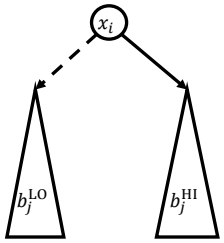


Figure 2.2. The outline of a diagram of BDDs.

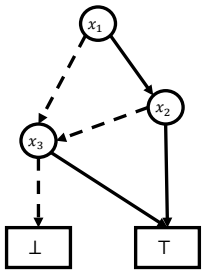


Figure 2.3. An example of BDDs that represent a Boolean function $(x_1 \wedge x_2), \vee (x_1 \wedge x_3) \vee x_3$.

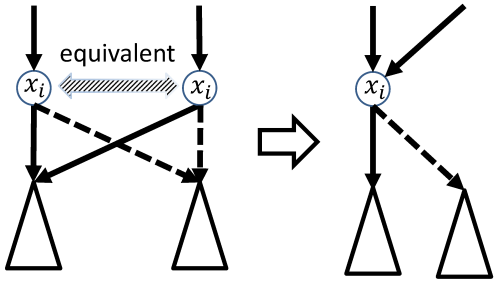


Figure 2.4. Node sharing.

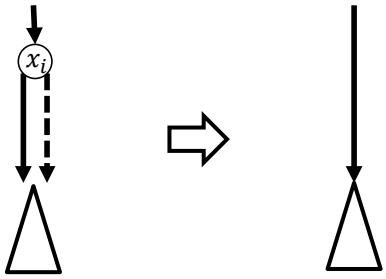


Figure 2.5. Node deletion of BDDs.

2.2.2 Knowledge Compilation Map

Darwiche [Darwiche and Marquis, 2002] presented a knowledge compilation map. A Knowledge compilation map is a system of tractable indexing structure, which contains of BDDs and its upper layer structures. The upper layer structures of BDDs in the map respect a tree instead of a linear order. The motivation is follows. When we represent a Boolean function with respecting a linear order, we have to process the propositional variables in the predefined order. However, to represent a Boolean function more succinctly, it would be more efficient to process some of the variables together. Moreover, processing these variables hierarchically may enable more efficient processing.

Most less structured tractable indexing structure in a compilation map is *NNF*. NNF is defined as follows:

Definition 2.2.2. *NNF is DAG, which leaves are labeled with true, false, X and ¬X, where X is included in a denumerable set of propositional variables. And its inner*

nodes are labeled with $\wedge$ *or* $\vee$.

However, to support efficient queries and transformations, a structure is necessary to be more structured. In the following two sections, we explain structured d-DNNF and SDDs as examples of structures that support sufficiently efficient queries and transformations.

2.2.3 Structured d-DNNF

In this section, we explain structured d-DNNF and its related notions.

$(\boldsymbol{X},\boldsymbol{Y})$-Decomposition

$(\boldsymbol{X},\boldsymbol{Y})$-decomposition is a method for decomposing given Boolean functions into subfunctions.

In this section, we use an upper case letter to denote a variable (e.g., X) and a lower case letter (e.g., x) to denote its instantiation (assignment). Moreover, we use a bold upper case letter to denote a set of variables (e.g., $\boldsymbol{X}$) and a bold lower case letter (e.g., $\boldsymbol{x}$) to denote their instantiations.

Let f be a Boolean function and $\boldsymbol{U}$ be its propositional variables. The conditioning of f on instantiation $\boldsymbol{x}$, written $f|\boldsymbol{x}$, is a subfunction that results from setting variables $\boldsymbol{X}$ to their values in $\boldsymbol{x}$. A function f is called *essentially depends* on variables in $\boldsymbol{X}$ if for any variable $y \notin \boldsymbol{X}$, we have $f|y = f|\neg y$. We write $f(\boldsymbol{X})$ to indicate that f essentially depends on variables $\boldsymbol{X}$. Let $\boldsymbol{X},\boldsymbol{Y}$ be mutually exclusive subsets of $\boldsymbol{U}$ and let $\boldsymbol{X} \cup \boldsymbol{Y} = \boldsymbol{U}$. For $i = 1,\ldots,n$, let $p_i^l(\boldsymbol{X})$ be a Boolean function, which essentially depends on variables in $\boldsymbol{X}$, and let $p_i^r(\boldsymbol{Y})$ be a Boolean function, which essentially depends on variables in $\boldsymbol{Y}$.

The $(\boldsymbol{X},\boldsymbol{Y})$-decomposition of a Boolean function f is described as follows:

$$f = [p_1^l(\boldsymbol{X}) \wedge p_1^r(\boldsymbol{Y})] \vee \ldots \vee [p_n^l(\boldsymbol{X}) \wedge p_n^r(\boldsymbol{Y})]. \tag{2.7}$$

Vtree

A *vtree* is a rooted, full, and ordered binary tree. Vtree is used to represent how to recursively $(\boldsymbol{X},\boldsymbol{Y})$-decompose a Boolean function.

Each node of a vtree is called a *vnode*. In a vtree, each leaf corresponds to a distinct item of the propositional variables. Here, the left and right children of the internal vnode α are denoted by α^l and α^r, respectively. The set of variables corresponding to the leaves of a vtree rooted at vnode α is denoted by $\boldsymbol{E}_\alpha$, and the variable corresponding to a leaf vnode α is denoted by E_α. An internal vnode α represents a partition of the set of items $\boldsymbol{E}_\alpha$ into two sets $\boldsymbol{E}_{\alpha^l}$ and $\boldsymbol{E}_{\alpha^r}$. We use a vtree to recursively $(\boldsymbol{X},\boldsymbol{Y})$-decompose

a Boolean function, starting from the root of the vtree. We show an example of a vtree in Fig. 2.6. In the figure, the number of each node is the ID of the vnode. For example, the root vnode 3 corresponds to a $(\boldsymbol{X},\boldsymbol{Y})$-decomposition, where $\boldsymbol{X} = \{A,B\}$ and $\boldsymbol{Y} = \{C,D\}$. Similarly, vnode 1 corresponds to a $(\boldsymbol{X},\boldsymbol{Y})$-decomposition, where $X = \{A\}$ and $Y = \{B\}$.

Structured d-DNNF

Structured d-DNNF is a tractable indexing structure that represents a Boolean function [Pipatsrisawat and Darwiche, 2008]. Structured d-DNNF represents a Boolean function as a DAG that shows recursive $(\boldsymbol{X},\boldsymbol{Y})$-decompositions of the function.

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

Definition 2.2.3. *A structured d-DNNF that respects vtree Φ is one of the following.*

1. $d = \top$ *or* $d = \bot$.
   These represent Boolean functions true and false, respectively.

2. $d = E_\alpha$ *or* $d = \neg E_\alpha$.
   These represent Boolean functions E_α and $\neg E_\alpha$, 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. $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$.
   Here, β^l and β^r are the left and right children of the internal vnode β of a vtree Φ, respectively. d represents a Boolean function $\bigvee_{i=1}^n (\langle d^i_{\beta^l} \rangle \wedge \langle d^i_{\beta^r} \rangle)$, where $d^1_{\beta^l}, \ldots, d^n_{\beta^l}$ are dnodes that respect β^l; $d^1_{\beta^r}, \ldots, d^n_{\beta^r}$ are dnodes that respect β^r; and $\langle d \rangle$ is $(\boldsymbol{X},\boldsymbol{Y})$-decomposed with $X = \boldsymbol{E}_{\beta^l}$ and $Y = \boldsymbol{E}_{\beta^r}$.

The former two dnodes are called terminal dnodes and the latter is called a decision dnode. The ordered pair of dnodes $(d^i_{\beta^l}, d^i_{\beta^r})$ that appears in a decision dnode is called an element. The size of a structured d-DNNF is defined as the number of elements.
A decision dnode combined with its child elements is called a decomposition. A decision dnode d_β is said to satisfy determinism if $(\langle d^i_{\beta^l} \rangle \wedge \langle d^i_{\beta^r} \rangle) \wedge (\langle d^j_{\beta^l} \rangle \wedge \langle d^j_{\beta^r} \rangle) = false$ for any $i \neq j$. An $(\boldsymbol{X},\boldsymbol{Y})$-decomposition $\{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, \}$ in a decision dnode d_β is said to satisfy structured decomposability if $\boldsymbol{E}_{\beta^l} \cap \boldsymbol{E}_{\beta^r} = \emptyset$. Every $(\boldsymbol{X},\boldsymbol{Y})$-decomposition that corresponds to a decision dnode of a structured d-DNNF satisfies determinism and structured decomposability.

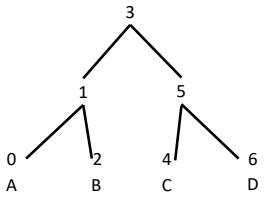


Figure 2.6. Example of vtree.

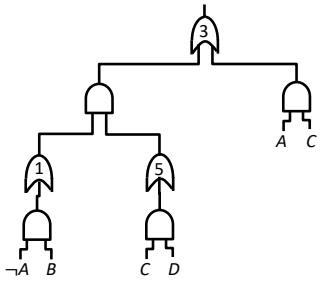


Figure 2.7. Structured d-DNNF that respects the vtree in Fig. 2.6 and represents the Boolean function, $(\neg A \wedge B \wedge C \wedge D) \vee (A \wedge C)$.

Note that unlike BDDs [Minato, 1993], a structured d-DNNF is not always canonical. That is, more than one semantically identical structured d-DNNFs that respect the same vtree are not always syntactically identical. Some queries can be efficiently answered with a structured d-DNNF, including the computation of the size of a Boolean function or enumeration of all valid assignments. However, unlike BDDs, the operations supported by structured d-DNNF is limited.

In Fig. 2.7, we show an example of a structured d-DNNF that respects the vtree in Fig. 2.6, and represents a Boolean function $(\neg A \wedge B \wedge C \wedge D) \vee (A \wedge C)$. An "or-gate"-shaped node in the figure represents a decision dnode, with its number represents the ID of the vnode that it respects. Furthermore, "and-gate"-shaped node combined with its left and right children represents an element. For example, the leftmost element $(\neg A, B)$ represents a Boolean function $\neg A \wedge B$. On the other hand, the element in the middle (C, D) represents a Boolean function $C \wedge D$, while the rightmost element (A, C) represent the families of sets, $A \wedge C$. In addition, the upper element other than (A, C) represents a Boolean function $\neg A \wedge B \wedge C \wedge D$. Consequently, the root dnode of Fig. 2.7 represents a Boolean function $(\neg A \wedge B \wedge C \wedge D) \vee (A \wedge C)$. The size of the structured d-DNNF is four.

From the definition of determinism, Proposition 2.2.1 holds. Fig. 2.8 illustrates an example of Proposition 2.2.1. It shows the structured d-DNNF of Fig. 2.7 with the assignment $(A, B, C, D) = (false, false, true, true)$. A red line represents true, whereas red circle represents the true element of an internal node. For three internal decision dnodes in the figure, at most one element is true.

Proposition 2.2.1. *For any internal dnodes* $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$ *of structured d-DNNF, at most one element* $(d^i_{\beta^l}, d^i_{\beta^r})$ *is true for any assignment.*

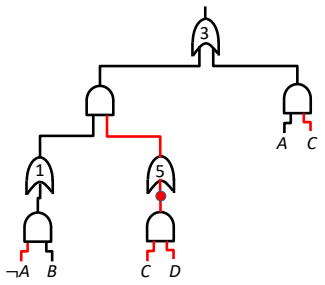


Figure 2.8. The structured d-DNNF of Fig. 2.7 with the assignment $(A,B,C,D) = (false, false, true, true)$. By determinism, at most one element is true for all internal dnodes.

2.2.4 SDDs

As same as structured d-DNNF, *SDD* is also a tractable indexing structure that represents a Boolean function [Darwiche, 2011a]. However, SDD is based on $(\boldsymbol{X},\boldsymbol{Y})$-partition, instead of $(\boldsymbol{X},\boldsymbol{Y})$-decomposition. That is, SDD represents a Boolean function as a DAG that shows recursive $(\boldsymbol{X},\boldsymbol{Y})$-partitions of the function. Since $(\boldsymbol{X},\boldsymbol{Y})$-partition is a more structured type of $(\boldsymbol{X},\boldsymbol{Y})$-decomposition, SDD has canonicity and has the same capability of query and transformation with BDD.

In the definition of $(\boldsymbol{X},\boldsymbol{Y})$-partition, the distinction between left and right side of each element in a decision dnode is crucial.

Definition 2.2.4. *In a decision dnode $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$ of a structured d-DNNF, the left side of each element $d^i_{\beta^l}$ is called prime and the right side is called sub. An decision dnode d_β is said to satisfy partitioned-determinism if its primes meet the following conditions:*

1. $\langle d^i_{\beta^l}\rangle \wedge \langle d^j_{\beta^l}\rangle = false$ *for any* $i \neq j$ *(mutually exclusiveness).*
2. $\bigvee_{i=1}^n \langle d^i_{\beta^l}\rangle = true$ *(consistency).*
3. $\forall i, \langle d^i_{\beta^l}\rangle \neq false$ *(validity).*

An decision dnode d_β that meets partitioned-determinism, is said to be compressed if its subs meet the following condition:

1. $d^i_{\beta^r} \neq d^j_{\beta^l}$, *for any* $i \neq j$.

Also, an decision dnode d_β that meets partitioned-determinism, is said to be trimmed if it is neither of the form $\{(\top, d)\}$ nor $\{(d,\top),(\neg d,\bot)\}$.

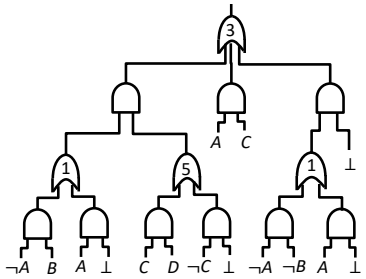


Figure 2.9. SDDs that respect the vtree in Fig. 2.6 and represent the Boolean function, $(\neg A \wedge B \wedge C \wedge D) \vee (A \wedge C)$

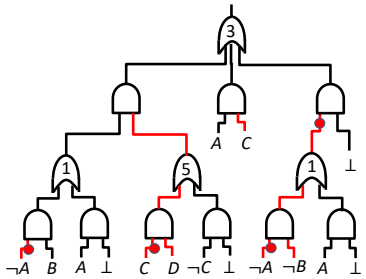


Figure 2.10. SDDs of Fig. 2.9 with the assignment $(A,B,C,D) = (false, false, true, true)$. By partitioned-determinism, exactly one prime of the elements is true for all internal dnodes.

Definition 2.2.5. *A structured d-DNNF is called SDDs if its all decision dnodes satisfies partitioned-determinism.*

Fig. 2.9 shows SDDs that represents a Boolean function, $(\neg A \wedge B \wedge C \wedge D) \vee (A \wedge C)$.

If all decision dnodes of SDDs are compressed and trimmed, they are canonical [Darwiche, 2011b]. Same as BDDS, SDDs also support apply operations.

From the definition of partitioned-determinism, Proposition 2.2.2 holds. Fig. 2.10 illustrates an example of Proposition 2.2.2. It shows the SDDS of Fig. 2.9 with the assignment $(A,B,C,D) = (false, false, true, true)$. A red line represents true, whereas a red circle represents a true prime of an element. For four internal decision dnodes in the figure, exactly one prime of the elements is true.

Proposition 2.2.2. *For any internal dnodes $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$ of SDDs, exactly one prime $d^i_{\beta^l}$ of the elements is true for any assignment.*

2.3 Tractable Indexing Structure with Zero-suppression

In the previous Section 2.2, structures are designed to represent a Boolean function. However, structures explained in this section are focused to represent a family of sets. The feature of structure in this section is zero-suppression. Firstly, we explain the correspondence between an indicator function and a family of sets. From this point of view, we see that a relationship between a Boolean function and a family of sets. Secondly, we explain a tractable indexing structure ZDDs. Finally, we explain a tractable indexing structure ZSDDs.

2.3.1 Indicator Function and a Family of Sets

Definition 2.3.1. *Indicator function of a set A is a function*

$$f_A : \boldsymbol{X} \to \{0,1\} \tag{2.8}$$

defined as:

$$f_A|_X := \begin{cases} 1 & if\ X \in A, \\ 0 & if\ X \notin A. \end{cases} \tag{2.9}$$

Note that Boolean function can be used to represent the family of sets as an indicator function. For example, an indicator function $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2)$ with variables $\{X_1, X_2\}$, represents a family of sets $\{\{x_x\}, \{x_1, x_2\}\}$ on the ground set $\{X_1, X_2\}$.

What is important in our argument is the "default" value, in other words, the omitted value in the expression. For example, with variables $\{X_1, X_2\}$, an indicator function $(x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$ is equal to x_1. It can be interpreted as "omitting a variable X_2 that does not affect the evaluation of the function". On the other hand, the set of families $\{\{x_1\}\}$ is represented with an indicator function $x_1 \wedge \neg x_2$. The expression $\{\{x_1\}\}$ can be interpreted as "omitting a variable X_2 that does not exist". It suggests that suppressing the variable that does not exist (zero-suppression) plays an important role to succinctly represent a family of sets.

2.3.2 ZDDs

Zero-suppressed binary decision diagram (ZDD) is a tractable indexing structure that represents a family of sets. ZDDs represent a family of sets as a DAG that shows recursive Shannon decomposition of a family of sets.

ZDDs consist of one or more nodes. Each node corresponds to a family of sets. To a node z, we let $\langle z \rangle$ be the family of sets corresponding to the node. $\sqcup$ represents the operation *join*, defined as $g \sqcup h = \{a \cup b \mid a \in g \text{ and } b \in h\}$. For example, when $g = \{\{1\}, \{2\}\}$ and $h = \{\{3\}, \{4\}\}$ are given, $g \sqcup h$ is equal to $\{\{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}\}$.

As same as BDDs, prior to the definition of ZDD, a linear order $P = (x_1, x_2, \ldots, x_n)$ is defined with a ground set $\{x_1, x_2, \ldots, x_n\}$. ZDDs respecting a linear order P is inductively defined as follows.

Definition 2.3.2. *ZDDs that respect a linear order P is one of the following.*

1. $z = \top$ *or* $z = \bot$.
   Semantics: $\top = \{\emptyset\}$ *and* $\bot = \emptyset$. *Here,* $\{\emptyset\}$ *is a family having only an empty set and* $\emptyset$ *is an empty family.*

2. $z = (x_i, z_j^{\mathrm{LO}}, z_j^{\mathrm{HI}})$, *where* $i \geq 1$. *And* $z_j^{\mathrm{LO}}(z_j^{\mathrm{HI}})$ *is a ZDD that respects a linear order* P_j, *where* $j > i$.
Semantics: $\langle z \rangle = (\{x_i\} \sqcup \langle z_j^{\mathrm{HI}} \rangle) \cup \langle z_j^{\mathrm{LO}} \rangle$.

The former is called a leaf node and the latter is called an internal node. When ZDDs meets the following, they are said to be reduced.

- *There is no internal node in the form* $(x_i, z_j^{\mathrm{LO}}, \bot)$.
- *In arbitrary two distinct internal nodes* $(x_i, z_j^{\mathrm{LO}}, z_j^{\mathrm{HI}})$ *and* $(x_k, z_l^{\mathrm{LO}}, z_l^{\mathrm{HI}})$, $(z_j^{\mathrm{LO}}, z_j^{\mathrm{HI}}) \neq (z_l^{\mathrm{LO}}, z_l^{\mathrm{HI}})$.

The former rule is called node sharing and the latter is called node deletion.

Same as BDDs, in the reduced ZDDs, $\langle z \rangle = \langle z' \rangle \Rightarrow z = z'$. That is, semantically equal ZDDs are also syntactically identical. However, note that the rules of node deletion of ZDDs and BDDs are different. Since ZDDS adopt the rule of node deletion in this section, ZDDs are said to be "zero-suppressed".

Fig. 2.11 shows an example of ZDDs that represents a family of sets $\{\{x_1, x_2\}, \{x_1, x_3\}, \{x_3\}\}$. The outline of node deletion of ZDDs is shown in Fig. 2.12.

Same as BDDs and SDDs, ZDDs support apply operations.

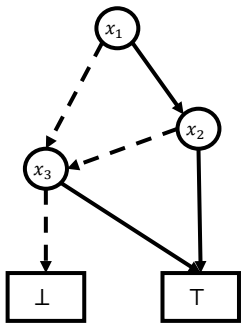


Figure 2.11. An example of ZDDs that represents a family of sets $\{\{x_1, x_2\}, \{x_1, x_3\}, \{x_3\}\}$.

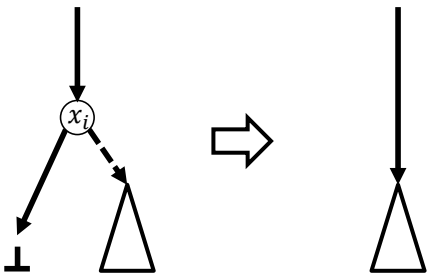


Figure 2.12. Node deletion of ZDDs.

2.3.3 ZSDDs

Zero-suppressed sentential decision diagram (ZSDD) is a tractable indexing structure that represents a family of sets. ZSDDs represent a family of sets as a DAG that shows

recursive $(\boldsymbol{X},\boldsymbol{Y})$-decomposition of the set, while ZDDs do recursive Shannon decomposition. ZSDD is a super set of ZDD and zero-suppressed version of SDD.

As described in Section 2.3.1, Boolean function can be used to represent the family of sets as an indicator function. Therefore, we redefine the notions used in Section 2.2 to represent a family of sets. Hereinafter, as long as a tractable indexing structure represents a family of sets, the notions defined in this section is applied to our argument.

$(\boldsymbol{X},\boldsymbol{Y})$-Decomposition

$(\boldsymbol{X},\boldsymbol{Y})$-decomposition is a method for decomposing given family of sets into subfamilies.

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

The $(\boldsymbol{X},\boldsymbol{Y})$-decomposition of a family of sets f is described as shown below; this decomposition is referred to as $(\boldsymbol{X},\boldsymbol{Y})$-decomposition.

$$f=[p_1^l(X)\sqcup p_1^r(Y)]\cup\ldots\cup[p_n^l(X)\sqcup p_n^r(Y)]. \tag{2.10}$$

Here, $\sqcup$ represents the operation *join*, defined as $g\sqcup h=\{a\cup b\mid a\in g \text{ and } b\in h\}$, where f and g 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 [Darwiche, 2011a]. Additionally, n in the equation (2.10) is $O(2^{|X|}2^{|Y|})$ with a naive method. In Chapter 4 and Chapter 5, we describe how to reduce the number of cases in our method.

Vtree

A *vtree* is a rooted, full, and ordered binary tree. In a vtree, each leaf corresponds to a distinct item of the ground set of the families of sets.

The set of items corresponding to the leaves of a vtree rooted at vnode α is $\boldsymbol{E}_\alpha$, and the item corresponding to a leaf vnode α is E_α. An internal vnode α represents a partition of the set of items $\boldsymbol{E}_\alpha$ into two sets $\boldsymbol{E}_{\alpha^l}$ and $\boldsymbol{E}_{\alpha^r}$. A vtree is used to recursively $(\boldsymbol{X},\boldsymbol{Y})$-decompose a family of sets, starting from the root of the vtree.

ZSDDs

As same as ZDD, *ZSDD* is also a tractable indexing structure that represents a family of sets [Nishino et al., 2017]. However, ZSDD represents a family of sets as a DAG that

shows recursive $(\boldsymbol{X},\boldsymbol{Y})$-partition of the function, in stead of Shannon decomposition. As same as SDD, ZSDD has canonicity. In addition, it has the same capability of query and transformation with ZDD.

The definition of ZSDDs is as follows:

Definition 2.3.3. *ZSDDs that respect a vtree Φ is one of the following.*

1. $d = \varepsilon$ *or* $d = \bot$.
   These represent families of sets $\{\emptyset\}$ and $\emptyset$, respectively.

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. $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$.
   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 $\bigcup_{i=1}^n \langle d^i_{\beta^l}\rangle \sqcup \langle d^i_{\beta^r}\rangle$, where $d^1_{\beta^l}, \ldots, d^n_{\beta^l}$ are dnodes that respect β^l; $d^1_{\beta^r}, \ldots, d^n_{\beta^r}$ are dnodes that respect β^r; and $\langle d\rangle$ is $(\boldsymbol{X},\boldsymbol{Y})$-decomposed with $X = \boldsymbol{E}_{\beta^l}$ and $Y = \boldsymbol{E}_{\beta^r}$.

The former two dnodes are called terminal dnodes and the latter is called a decision dnode. The ordered pair of dnodes $(d^i_{\beta^l}, d^i_{\beta^r})$ that appears in a decision dnode is called an element. The size of ZSDDs is defined as the number of elements.
A decision node combined with its child elements is called a decomposition. An $(\boldsymbol{X},\boldsymbol{Y})$-decomposition $\{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots,\}$ is said to satisfy structured decomposability if $\boldsymbol{E}_{\beta^l} \cap \boldsymbol{E}_{\beta^r} = \emptyset$. In a decision dnode, the left side of each element $d^i_{\beta^l}$ is called prime and the right side is called sub. An decision dnode d_β of ZSDDs satisfy partitioned-determinism that is defined as follows;

1. $\langle d^i_{\beta^l}\rangle \cap \langle d^j_{\beta^l}\rangle = \emptyset$ *for any $i \neq j$ (mutually exclusiveness).*

2. $\bigcup_{i=1}^n \langle d^i_{\beta^l}\rangle = 2^{E_{\beta^l}}$ *(consistency).*

3. $\forall i, \langle d^i_{\beta^l}\rangle \neq \emptyset$ *(validity).*

Every $(\boldsymbol{X},\boldsymbol{Y})$-decomposition that corresponds to a decision dnode of ZSDDs< satisfies partitioned-determinism and structured decomposability. An decision dnode d_β that meets partitioned-determinism, is said to be compressed if its subs meet the following condition: $d^i_{\beta^r} \neq d^j_{\beta^l}$, for $i \neq j$. Also, decision dnode d_β that meets partitioned-determinism, is said to be trimmed if it is neither of the form, $\{(d,\varepsilon),(d',\bot)\}$, $\{(\varepsilon,d),(d',\bot)\}$, nor $\{(d,\bot)\}$ [2]

[2] [Nishino et al., 2016] defines that it is neither of the form $\{(\varepsilon,\alpha),(\overline{\varepsilon},\bot)\}$,$\{(\alpha,\varepsilon),(\overline{\alpha},\bot)\}$ nor $\{(\mathrm{p},\bot)\}$. However, since ZDDs allow implicit partitioning, we adopt this definition.

Figure 2.13 shows ZSDDs that respect the vtree of Figure 2.6 and represents the family of sets $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$. If all decision dnodes of ZSDDs are compressed, they are canonical [Nishino et al., 2017]. Same as BDDs, SDDs, and ZDDs, ZSDDs support apply operations.

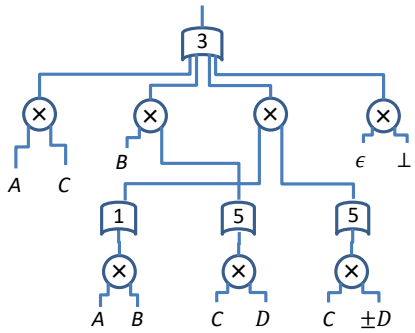


Figure 2.13. ZSDDs that respect the vtree in Fig. 2.6 and represent the family of set, $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$.

As described in Section 2.3.3 and Section 2.3.2, BDDs and SDDs have their zero-suppressed versions, ZDDs and ZSDDs. However, if we see the whole system of a knowledge compilation map, some tractable indexing structures lack their zero-suppressed versions. In Chapter 3, we define the missing part of the map and present a new tractable indexing structure that is suitable to design an efficient compilation method.

2.4 Compilation

2.4.1 Frontier-based Search

Knuth proposed *Simpath* [Knuth, 2011], which is a method to compile graph-substructures contained in a graph into ZDD [Minato, 1993]. Among the graph-substructures, for example, *s-t* paths, cycles, and Hamiltonian paths are studied. This method is also called a *frontier-based search* [Kawahara et al., 2017]. In this method, an exhaustive search is conducted on the edges or the gnodes of the given graph successively in a predefined order. ZDDs are simultaneously constructed, and consequently each ZDD node represents the candidate subsets of graph-substructures. 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 each gnode. The label is mapped to each gnodes based on the state of the incident edges. That is, the labels 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 [Knuth, 2011], can be efficiently 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 [Kawahara et al., 2017].

2.4.2 Compilation of *s-t* paths

In this section, we take *s-t* simple paths as a target of compilation. In the compilation process, we either select or do not select each edge of an input graph with a predetermined order. An edge determined to be selected or not is said to be *processed*, whereas an edge where the selection is not determined yet is said to be *unprocessed*. In addition, we simultaneously construct ZDDs, where each ZDD node corresponds to a candidate of *s-t* paths. In each candidate, a set of selected edges represents the *s-t* path fragments and each vertex is in one of three states: not contained in any path fragment; an endpoint of a path fragment; or an intermediate point of a path fragment.

To efficiently manage the states above, we classify the gnodes of a graph as follows: Let us denote the set of gnodes connected with the processed edges as U_1, and the set of gnodes connected with the unprocessed edges as U_2. Then, we call the gnode set $U = U_1 \cap U_2$ as the *frontier gnodes of edges*. Here, let $U_1' = U_1 \setminus U$ and $U_2' = U_2 \setminus U$. Then, it is clear that no edge connects a gnode in U_1' and U_2'. Thus, in order to classify these three states above, we need to focus only on the frontier gnode set U. Notice that a frontier gnode set has the same property with separation lemma of tree decomposition in Section 2.1.3. The size of the set of frontier gnodes of edges is at most the path width plus one [Inoue and Minato, 2016].

The state of the frontier gnode is represented by the array called *mate*. The size the array equals the size of the frontier gnodes. The state of each frontier gnode $u \in V$ is stored in $mate[u]$. That is, we assign the values of *mate* corresponding to the state of gnode of the input graph as follows:

$$mate[u] = \begin{cases} u & if \text{ gnode } u \text{ does not connect to a selected edge (isolated point).} \\ v & if \text{ gnode } u \text{ and } v \text{ are the terminals of a path fragment.} \\ 0 & if \text{ gnode } u \text{ is an intermediate point of a path fragment.} \end{cases} \tag{2.11}$$

The assignment of *mates* to the frontier gnodes is called *configuration*.

If the values of the mates assigned to the frontier gnodes are equivalent in two candidates, they share the same remaining search space. Then, the corresponding ZDD nodes can be merged. On the other hand, pruning is performed when the path fragment of a candidate meets the following conditions. That is because, these conditions are clearly invalid to construct *s*-*t* paths.

(1) The degree of s or t is 2.

(2) The degree of a gnode other than s,t is 3.

(3) A cycle occurs.

(4) When s or t goes out of the frontier, their degrees are determined to be zero.

(5) When gnodes other than s,t go out of the frontier, their degrees are determined to be one.

The above judgement is conducted as follows. If $mate[u_1] = u_1$, the degree of u_1 is zero. If $mate[u_1] = u_2 (\neq u_1)$, the degree of u_1 is one. If $mate[u_1] = 0$, the degree of u_1 is two. By using these facts, we judge that a fragment under construction has reached the degree defined in (1) or (2) by the new addition of an edge. When we newly connect the edge $e = (u_1, u_2)$ with $mate[u_1] = mate[u_2]$, (3) holds. (4) holds if a gnode s or t goes out of the frontier and $mate[s] = s$ or $mate[t] = t$. (5) holds if a gnode u_i other than s or t goes out of the frontier and $mate[u_i] = u_j (\neq u_i)$.

In addition, if $mate[s] = t, mate[t] = s$ in a candidate, an *s*-*t* simple path is accomplished, as long as no end point exists in any gnode other than s or t. However contrary, if more than one connected component other than the *s*-*t* path exists, this candidate is invalid. This is the pruning condition (6). Thus, we have six types of pruning conditions in total.

Examples for mates corresponding to the pruning conditions from (1) to (6) are shown in the figures, from Fig. 2.14 to Fig. 2.19 respectively.

2.4.3 Compilation of independent sets

In this section, we take independent sets as a target of compilation. As a trivial extension of Knuth or Kawahara's approach, the independent sets contained in a given 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. Also, we simultaneously construct ZDDs, where each ZDD node corresponds to a candidate of independent sets. In the procedure, $mate[v] = 1$ is assigned to gnodes v contained in the independent set candidate, whereas $mate[u] = 0$ is assigned to gnodes u not contained in the candidate.

Here, a gnode where no *mate* is assigned is said to be *unprocessed*. On the other hand, a gnode where *mates* are assigned including all of its adjacent gnodes is said to be *processed*. Then, the remaining gnodes are defined as *frontier gnodes*. By definition, an unprocessed gnode is not adjacent to any processed gnode. Therefore, by the same reason with the compilation of the *s*-*t* paths, only the adjacency of the frontier gnodes are needed to be checked. Thus, if the values of the mates assigned to the frontier gnodes are equivalent in two 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. Consequently, the corresponding ZDD node is pruned.

Note that, frontier gnode set is equal to *vertex separator*. 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 $\max_{1 \leq i \leq |V|} |F_i|$. 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 [Kinnersley, 1992].

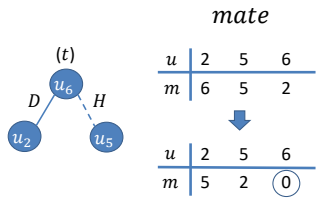


Figure 2.14. (1) The degree of s or t is determined as two.

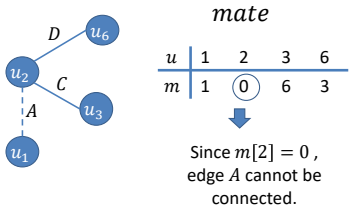


Figure 2.15. (2) The degree of gnodes other than s and t is determined as three.

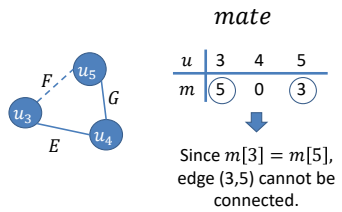


Figure 2.16. (3) A cycle occurs.

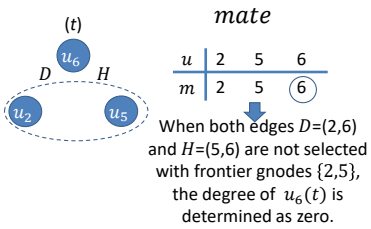


Figure 2.17. (4) When leaving the frontier, the degree of s or t is determined as zero.

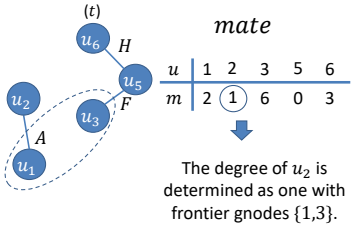


Figure 2.18. (5) When leaving the frontier, the degree of gnodes other than s and t is determined as one.

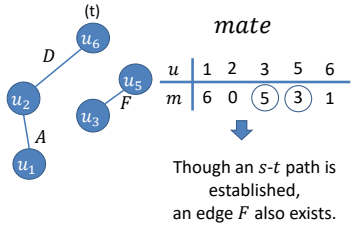


Figure 2.19. (6) Though an s-t path is accomplished, more than one connected component other than the s-t path exists.

Chapter 3

Structured Z-d-DNNF and Zero-suppressed Knowledge Compilation Map

In this chapter, firstly, we present our tractable indexing structure structured Z-d-DNNF. Unlike ZSDDs, it adopts determinism instead of partitioned-determinism. As it is less structured than ZSDDs, its capability is also limited. However it also supports a certain capability for the application after the compilation. In addition, the simplicity of this structure helps the compilation method to be simpler and faster than ZSDDs. The compilation method of our structure will be detailed in Chapter 4 and Chapter 5.

Secondly, we present zero-suppressed knowledge compilation map, which is given by introducing zero-suppression towards the whole knowledge compilation map. While a conventional knowledge compilation map is a system of tractable indexing structures to represent a Boolean function in general, our zero-suppressed one is focused on representing a family of sets, especially graph substructure.

3.1 Structured Z-d-DNNF

In this section, we present structured Z-d-DNNF.

Structured Z-d-DNNF is a tractable indexing structure that represents a family of sets [Sugaya et al., 2017, Sugaya et al., 2018]. Structured Z-d-DNNF represents a family of sets as a DAG that shows recursive $(\boldsymbol{X},\boldsymbol{Y})$-decompositions of the function. As described below, it does not represent a family of sets with Shannon decomposition as ZDDs do. That is, it does not respect a linear order. Additionally, it does not represent a family of sets with $(\boldsymbol{X},\boldsymbol{Y})$-partition as ZSDDs do. Indeed, the only difference between structured Z-d-DNNF and ZSDD is that the former is dependent on $(\boldsymbol{X},\boldsymbol{Y})$-decomposition, while the latter is dependent on $(\boldsymbol{X},\boldsymbol{Y})$-partition.

Vtree

In this Section, we apply the notion vtree explained in Section 2.3.3. However, we expand the notion of vtree to adopt a "dummy vnode", namely a vnode that has no corresponding item.

In a vtree, each leaf corresponds to a distinct item of the ground set of a family of sets, except the dummy vnode. 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, which will be explained in Section 4.4. A vtree that includes dummy nodes is called a *dummy-added vtree*. The example of a vtree with dummy nodes is shown in Fig. 4.1 in Section 4.4.

Structured Z-d-DNNF

Structured Z-d-DNNF is a tractable indexing structure that represents families of sets [Sugaya et al., 2018]. Structured Z-d-DNNF represents a family of sets as a DAG that shows recursive $(\boldsymbol{X},\boldsymbol{Y})$-decompositions of the family of sets.

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 family of sets corresponding to d. Structured Z-d-DNNF respecting a vtree Φ is inductively defined as follows.

Definition 3.1.1. *A structured Z-d-DNNF that respects vtree Φ is one of the following.*

1. $d = \varepsilon$ *or* $d = \bot$.
   These represent families of sets $\{\emptyset\}$ and $\emptyset$, respectively.

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. $d = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$.
   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 $\bigcup_{i=1}^n \langle d^i_{\beta^l} \rangle \sqcup \langle d^i_{\beta^r} \rangle$, where $d^1_{\beta^l}, \ldots, d^n_{\beta^l}$ are dnodes that respect β^l, while $d^1_{\beta^r}, \ldots, d^n_{\beta^r}$ are dnodes that respect β^r. Thus, $\langle d \rangle$ is $(\boldsymbol{X},\boldsymbol{Y})$-decomposed with $\boldsymbol{X} = E_{\beta^l}$ and $\boldsymbol{Y} = E_{\beta^r}$.

The former two dnodes are called terminal dnodes and the latter is called a decision dnode. The ordered pair of dnodes $(d^i_{\beta^l}, d^i_{\beta^r})$ 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 decision dnode d_β of a structured Z-d-DNNF is said to satisfy determinism if $(\langle d^i_{\beta^l} \rangle \sqcup$

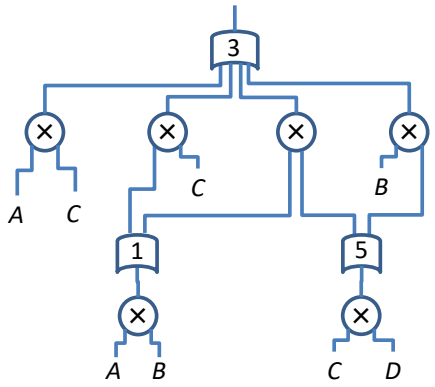


Figure 3.1. Structured Z-d-DNNF, which respects the vtree in Fig. 2.6 and represents the family of sets, $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$.

$\langle d^i_{\beta^r}\rangle)\cap(\langle d^j_{\beta^l}\rangle \sqcup \langle d^j_{\beta^r}\rangle) = \emptyset$ *for any* $i \neq j$. *An* $(\boldsymbol{X},\boldsymbol{Y})$*-decomposition* $\{(d^1_{\beta^l}, d^1_{\beta^r}),\ldots\}$ *in a decision dnode* d_β *is said to satisfy structured decomposability if* $E_{\beta^l} \cap E_{\beta^r} = \emptyset$. *Every* $(\boldsymbol{X},\boldsymbol{Y})$*-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 $(\boldsymbol{X},\boldsymbol{Y})$-decomposition of the form $\{(\varepsilon,\alpha)\}$ or $\{(\alpha,\varepsilon)\}$, it is said to be *empty-trimmed*. If a structured Z-d-DNNF has no $(\boldsymbol{X},\boldsymbol{Y})$-decomposition of the form $(\bot,\alpha)$ or $(\alpha,\bot)$, 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 [Minato, 1993], a reduced structured Z-d-DNNF is not always canonical, i.e., semantically equal structured Z-d-DNNFs are not always syntactically identical.

In Fig. 3.1, we show an example of a structured Z-d-DNNF that respects the vtree shown in Fig. 2.6 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. 3.1 represents a family of sets $\{\{A,C\},\{A,B,C\},\{A,B,C,D\},\{B,C,D\}\}$.

3.2 Query and Transformation supported by Structured Z-d-DNNF

In this subsection, we explain some of queries and transformations supported by Structured Z-d-DNNF. Here, the family of sets represented by a structure is called model.

3.2.1 Query

First, we take queries. Let n be the cardinality of the set of the dnodes contained in a structured Z-d-DNNF and m be the size of the family of sets that a structured Z-d-DNNF represents. By the same calculation as that of d-DNNF [Darwiche, 2001b, Darwiche and Marquis, 2002], the following theorem holds.

Theorem 3.2.1. *The size of the family of sets represented by a structured Z-d-DNNF can be calculated in time $O(n)$. In addition, the family of sets represented by a structured Z-d-DNNF can be enumerated in time $O(p(n,m))$. Here, $p(n,m)$ is a polynomial for n,m.*

Proof. Structured Z-d-DNNF satisfies decomposability and determinism. Therefore, as same as d-DNNF, it can yield the number of models and enumerate its representing family of sets by the following calculation [Darwiche, 2001b]. Assuming that the cardinality of the family of sets represented by dnode d is $\mathrm{CT}(d)$, the following holds.

- $\langle d\rangle = \emptyset, \mathrm{CT}(d) = 0$: When d is the terminal dnode labeled with $\bot$.
- $\langle d\rangle = \{\emptyset\}, \mathrm{CT}(d) = 1$: When d is the terminal dnode labeled with ε.
- $\langle d\rangle = \{\{X\}\}, \mathrm{CT}(d) = 1$: When d is the terminal dnode labeled with X.
- $\langle d\rangle = \{\{X\}, \emptyset\}, \mathrm{CT}(d) = 2$: When d is the terminal dnode labeled with $\pm X$.
- $\langle h^i\rangle = \langle d_l^i\rangle \sqcup \langle d_r^i\rangle, \mathrm{CT}(h^i) = \mathrm{CT}(d_l^i) * \mathrm{CT}(d_r^i)$: When h^i is an i-th element of some dnode. Here, d_l^i and d_r^i are the left and right elements of h^i, respectively.
- $\langle d\rangle = \bigcup_i \langle h^i\rangle, \mathrm{CT}(d) = \sum_i \mathrm{CT}(h^i)$: When d is a decision dnode. Here, h_i is a child element of d.

$\mathrm{CT}(root)$ of the root dnode $root$ is the total number of models. $\mathrm{CT}(root)$ is obtained in time $O(n)$. Likewise, $\langle root\rangle$ of the root dnode $root$ is the family of sets in the whole representation. $\langle root\rangle$ is obtained in time $O(p(n,m))$. □

3.2.2 Transformation

Next, we take *Random Sampling* as an example of transformation. After the compilation of a certain family of sets of a given graph, we sometimes need to pick some samples from them. However, the total size of the a family of sets tends to be very large. In those cases, in order to select a few of them, a naive enumeration method is impractical. For this purpose, if the compilation result is available, it is useful to randomly sample a family of sets.

We assume that the a family of sets of a graph is already compiled and reduced, additionally the size of a family of sets are already calculated. In addition, we also assume the cumulative sum of the child elements at each dnodes is already calculated, since it is necessary for the 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 $(d_l^i, d_r^i) \in d$ with probability $\frac{\mathrm{CT}(d_l^i)\mathrm{CT}(d_r^i)}{\mathrm{CT}(d)}$, where $\mathrm{CT}(d)$ represents the cardinality of the a family of sets represented by d. We repeat this procedure until we reach a terminal, which yields a random sampling of the a family of sets represented by d. The time complexity is proportional to $\sum_{i\in\{1,\ldots,H\}} \log(\mathrm{Ave}[|d(i)|])$. Here, 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 3.2.1.

Algorithm 3.2.1 RandomSample(d)

Let $\mathrm{CT}(d)$ be the size of a family of sets represented by d. We assume that d is already reduced and $\mathrm{CT}(d)$ is calculated.

if d is terminal **then**
  if $d = \varepsilon$ or $d = e_\alpha$ for an item e_α **then**
    return d
  else if $d = \pm e_\alpha$ for an item e_α **then**
    Assign e_α or ε to d' with probability $\frac{1}{2}$ respectively
    return d'
  end if
else
  Assign (d_l^i, d_r^i) to (d_l', d_r') with probability $\frac{\mathrm{CT}(d_l^i)\mathrm{CT}(d_r^i)}{\mathrm{CT}(d)}$ respectively.
  $d_l' \leftarrow$ RandomSample(d_l')
  $d_r' \leftarrow$ RandomSample(d_r')
  return $d' = \{(d_l', d_r')\}$
end if

3.3 Zero-suppressed Knowledge Compilation Map

In this section, we explain zero-suppressed knowledge compilation map and the difference between ZSDDs over our proposed structure structured Z-d-DNNFFig. 3.2.

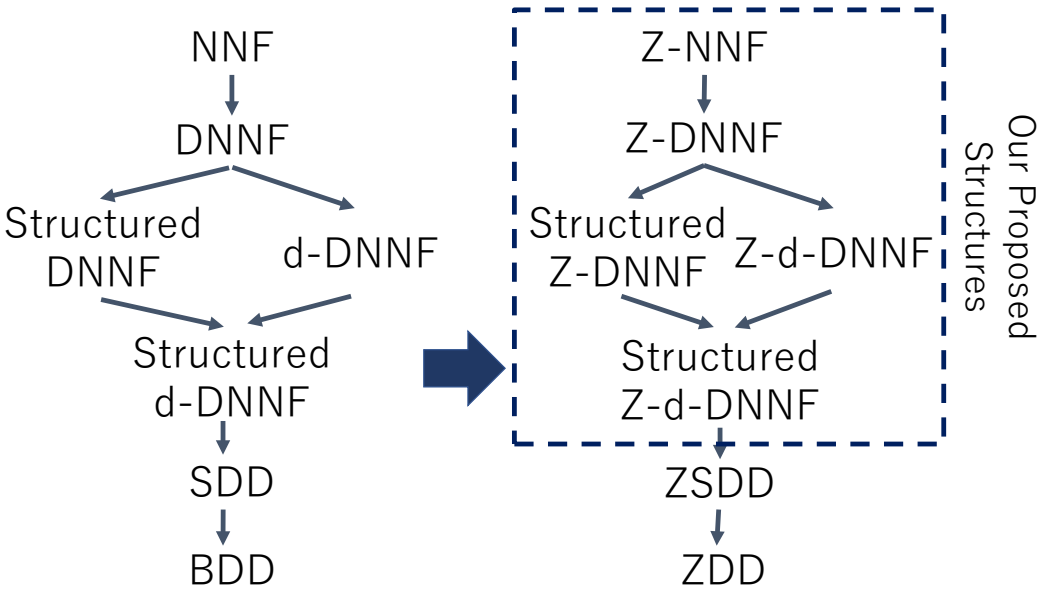


Figure 3.2. System of knowledge compilation map,
where the left side is that of Darwiche et al. and the right side is zero-suppressed ones.

As shown in Definition 2.2.2 in Section 2.2.2, NNF is DAG, which leaves are labeled with true, false, X and $\neg X$, where X is included in a denumerable set of propositional variables. And its inner nodes are labeled with $\wedge$ or $\vee$. Besides, we define Z-NNF as follows:

Definition 3.3.1. *Z-NNF is DAG, which leaves are labeled with $\bot$, ε, X and $\pm X$, where X is included in a denumerable set of propositional variables. And its inner nodes are labeled with union $\cup$ or join $\sqcup$.*

In a word, Z-NNF is a zero-suppressed version of NNF. From this point of view, we see that our proposed structured Z-d-DNNF is a zero-suppressed version of structured d-DNNF, while ZSDDs are zero-suppressed version of SDDs. That is, Z-NNF that satisfies determinism and structured decomposability is structured Z-d-DNNF, while NNF that satisfies determinism and structured decomposability is structured d-DNNF. On the other hand, Z-NNF that satisfies partitioned-determinism and structured decomposability is ZSDD, while NNF that satisfies partitioned-determinism and structured decomposability is SDD.

The difference between ZSDDs and our proposed structure structured Z-d-DNNF is follows. Since structured Z-d-DNNF does not satisfy partitioned-determinism, it does not support canonicity in general, unlike ZSDDs do. As a result, the reduced instance of structured Z-d-DNNF empirically tends to be larger than those of ZSDDs. Moreover,

structured Z-d-DNNF does not support some of the transformation like bounded union and also does not support apply operations, which are supported by ZSDDs.

However, since structured Z-d-DNNF is less structured than ZSDDs, the design of a compilation method tend to be simpler than that of ZSDDs. As a result, the running time of compilation tends to be faster than that of ZSDDs. We show the experimental results regarding the advantages and disadvantages of our proposed structure structured Z-d-DNNF over ZSDDs in Section 4.6.

3.4 Concluding Remarks

In this chapter, we have presented our tractable indexing structure structured Z-d-DNNF, and a new system of structures, zero-suppressed knowledge compilation map. Unlike ZSDDs, structured Z-d-DNNF adopts determinism instead of partitioned-determinism. As a result, the capability of structured Z-d-DNNF is limited and supports less query and transformation than SDDs. However, it supports certain amount of applications after the compilation, such as model counting, model enumeration, and random sampling. On the other hand, the simplicity of this structure helps the compilation method to be simpler and faster than ZSDDs, which will be detailed in the following Chapter 4 and Chapter 5.

In addition, we have presented zero-suppressed knowledge compilation map. Though, BDDs have had more succinct structures as its extension, ZDDs lacked the extension except ZSDDs. We defined Z-NNF as a top level structure instead of NNF. Subsequently, we applied (structured) decomposability and (partitioned) determinism to the descendants of Z-NNF, and realized a tractable indexing structure system with zero-suppression,

Our future work is to study the detailed capability of each structure in zero-suppressed knowledge compilation map. Darwiche clarified polytime queries and transformations supported by each structure in knowledge compilation map [Darwiche, 2001b, Darwiche and Marquis, 2002]. Further study of capability of the structures in zero-suppressed knowledge compilation map is inevitable to clarify the detail of zero-suppressed knowledge compilation map and find the applications of the structures.

Chapter 4

Merging Frontier-Based Search to Compile Independent Sets

In this and next chapter, we explain our method merging frontier-based search. Merging frontier-based search is a method of compilation with using structured Z-d-DNNF as a tractable indexing structure. In this chapter, we focus on the compilation of the independent sets in an input graph [Sugaya et al., 2020].

Same as conventional frontier-based search, in order to compile the independent sets, we conduct an exhaustive search on gnodes of an input graph. However, there are three features that we need to take care when we design the compilation method with using structured Z-d-DNNF.

1. Frontier: To compile the independent sets, unlike conventional frontier-based search, our method uses a bag of tree decomposition as frontier. We explain a bag indeed plays the same role as a frontier gnode set of conventional frontier-based search, with using a notion separator,

2. The construction of a vtree: In order to design an efficient compilation method with exploiting the tree-width of an input graph, it is necessary to device the construction of a vtree. We explain a method to construct a vtree based on the tree decomposition of an input graph.

3. The management of the states of searching branches: As we explained in Chapter 3, structured Z-d-DNNF respects a vtree instead of the linear order of gnodes. Therefore, unlike conventional frontier-based search, we need to merge the states of the searching branches. We explain how to merge the states in the process of the compilation of the independent sets.

4.1 Overview

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. 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 [Sugaya et al., 2018]. 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 explain our method for compiling the independent sets in an input graph. In our method, we apply a preceding dynamic programming of finding the maximum number of independent sets, which is described in Section 2.1.3. Firstly, to clarify the comparison of preceding method and ours, we describe our compilation method as a dynamic programming scheme for computing the independent sets (Section 4.2). Secondly, we explain the method for constructing the vtree from the nice tree decomposition of an input graph (Section 4.3). Thirdly, we explain the method for compiling the independent sets into structured Z-d-DNNFs using the vtree (Section 4.4). 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. Fourthly, we explain the correctness and complexity of the algorithm (Section 4.5), and finally, conclude this chapter (Section 4.7).

4.2 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 2.1.3, 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 as in Section 2.1.3, we need to distinguish which gnodes are and are not contained in an independent set. Thus, 1 is mapped to the former and 0 to the latter. The former gnodes must not be adjacent and it can be judged by locally checking the adjacency of gnodes because of the separation Lemma 2.1.1 in Section 2.1.3.

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:

$$\mathcal{S}(i; P_i, Q_i) = \left\{ S_i \;\middle|\; \begin{array}{l} S_i \subseteq V_i \\ S_i \text{ is an independent set} \\ P_i \subseteq S_i, Q_i \cap S_i = \emptyset, P_i \cup Q_i = B_i \end{array} \right\}. \tag{4.1}$$

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 2.4.3, 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 2.1.3, 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 4.2.1, 4.2.2, 4.2.3, and 4.2.4. Note that, as a consequence of Lemma 2.1.1, 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 2.4.3.

Theorem 4.2.1 (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 4.2.2 (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 2.1.3: (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*

$$\mathcal{S}(i; P_i, Q_i) = \left\{ \begin{array}{ll} \{\emptyset\} & \text{(I)} \\ \mathcal{S}(j; P_i \setminus \{u\}, Q_i) \sqcup \{\{u\}\} & \text{(II)} \\ \mathcal{S}(j; P_i, Q_i \setminus \{u\}). & \text{(III)} \end{array} \right. . \tag{4.2}$$

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 2.1.1, $I \cup \{u\}$ 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 4.2.3 (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

$$\mathcal{S}(i;P_i,Q_i) = \mathcal{S}(j;P_i \cup \{w\},Q_i) \cup \mathcal{S}(j;P_i,Q_i \cup \{w\}). \tag{4.3}$$

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

$$\mathcal{S}(i;P_i,Q_i) = \mathcal{S}(j;P_i,Q_i) \sqcup \mathcal{S}(k;P_i,Q_i). \tag{4.4}$$

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 2.1.1, 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 $\forall I_k \in \mathcal{S}(k;P_i,Q_i)$. □

4.3 Vtree Construction from a Nice Tree Decomposition

Next, we explain the construction of a dummy-added vtree from a nice tree decomposition. Hereinafter, for simplicity, a dummy-added vtree is called a vtree. 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. In contrast, among two children j and k of a join node of a nice tree decomposition, V_j and V_k are not always mutually exclusive.

Therefore, we first transform a given nice tree decomposition $\mathcal{T}$. That is, we conduct transformation to make the gnodes disjoint, which are contained in the left and right children of each internal node. The transformation method is as follows: Without loss of generality, we can specify j as the left child and k as the right child of a join tnode i in $\mathcal{T}$. For j and k, we let V_m be $V_m = V_j \cap V_k$. Furthermore, for each tnode k included in 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 $\mathcal{T}'$.[1]

Secondly, in order to generate a vnode, we conduct a depth-first search on the tree $\mathcal{T}'$ with priority to the left side. Here, we inductively define the process of vnode generation. A node of tree $\mathcal{T}'$ is denoted as $i, j, k, \ldots$.

[1] $\mathcal{T}'$ cannot always be a tree decomposition, because it does not always satisfy (2) in Definition 2.1.1.

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

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

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

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

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

In Fig. 4.1, we show an example of a vtree generated from a nice tree decomposition of Fig. 2.1.

4.4 Compilation with Structured Z-d-DNNF

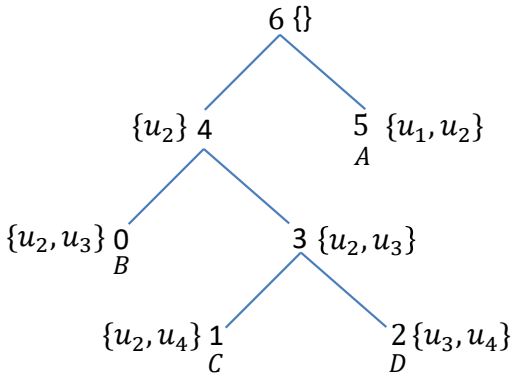


Figure 4.1. Example of a dummy-added vtree generated from a nice tree decomposition in Fig. 2.1.

In this section, we present a method of compilation into structured Z-d-DNNF based on the dynamic programming scheme described in Section 4.2. 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, in a tree decomposition, 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.

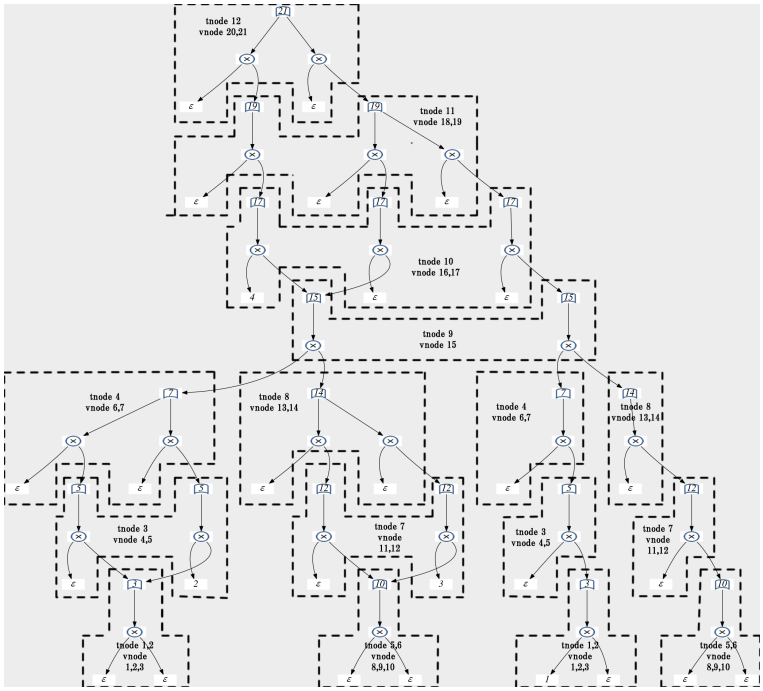


Figure 4.2. Compilation result of the graph in Fig. 2.1.

Therefore, we limit $\mathcal{S}(i)$ to the tree $\mathcal{T}'$ in Section 4.3. Suppose that we have a join tnode i and its left and right children j and k. Let $V_m = V_j \cap V_k$. Then, for the right child and its descendant r, we exchange $\mathcal{S}(r)$ with $\mathcal{S}'(r) = \{S \setminus V_m | S \in \mathcal{S}(r)\}$. That is, in each $S(i;P_i,Q_i)$, we exchange with $S'(i;P'_i,Q'_i)$ with $P'_i = P_i \setminus V_m$, $Q'_i = Q_i \setminus V_m$. 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 roots $\mathcal{S}'(root)$ and $\mathcal{S}(root)$ are equal. Consequently, all the independent sets of the input graph are represented by the root $\mathcal{S}'(root)$. We introduce the limitation above towards the dynamic programming in Section 4.2. As shown in Section 4.5, the hierarchy of these nodes meets the definition of a

Table 4.1. Process with inputs Fig. 2.1 and Fig. 4.1.

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	$\mathcal{S}(12;\emptyset,\emptyset)$	$\{\emptyset,\{2\},\{3\},\{4\},\{2,4\},\{3,4\},\{1\}\}$

structured Z-d-DNNF.

In the following, we inductively define the method for composing a dnode with regarding to a node type i. Each family of sets $\mathcal{S}'(i;P'_i,Q'_i)$ below has one-to-one correspondence with $\mathcal{S}(i;P_i,Q_i)$ of the dynamic programming in Section 4.2.

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

(2) When a node i has one child j with $B'_i = B'_j \cup \{u\}$: There are two cases of which sets contains u, P'_i and Q'_i. In the first case, 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 $\mathcal{S}'(j;P'_i \setminus \{u\},Q'_i)$. In the second case, we construct an element with the ordered pair of dnodes, one of which is a terminal dnode representing $\{\varepsilon\}$, and the other 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 with$B'_i \cup \{w\} = B'_j$: 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 $\{\varepsilon\}$ 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 with $B'_i = B'_j$: We construct an element with the ordered pair of dnodes, one of which is a terminal dnode that represents $\{\varepsilon\}$, and the other 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 $\mathcal{S}'(i;P'_i,Q'_i)$ with $P'_i = P'_j$ and $B'_i = B'_j$.

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

Example 4.4.1. *We describe an example of the method for constructing a structured Z-d-DNNF. We take the graph of Fig. 2.1 and its vtree Fig. 4.1 as inputs. Then, we have Fig. 4.2 as the result of compiling all the independent sets contained in the input graph. In Fig. 4.2, 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_3^1, d_3^2 from the left, respectively. Furthermore, Table 5.1 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*. The other is* $\mathcal{S}(2;\{1\},\emptyset) = \{\{1\}\}$*, which contain gnode* 1 *in* B_2*. The corresponding elements are* $(\varepsilon,\varepsilon)$ *and* (v_1,ε)*, which are contained in the two decision dnodes* d_3^1 *and* d_3^2*, 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_3^1)*,* (v_2,d_3^1)*, and* (ε,d_3^2)*, which are contained in the decision dnodes* d_5^1*,* d_5^2*, and* d_5^3*. A vnode* 5 *is respected by these decision dnodes. In tnode* 4*, because gnode* 2 *is forgotten, we have the families of independent sets* $\mathcal{S}(4;\emptyset,\{1\}) = \{\emptyset,\{2\}\}$ *and* $\mathcal{S}(4;\{1\},\emptyset) = \{\{1\}\}$*. The corresponding elements are* (ε,d_5^1)*,* (ε,d_5^2)*, and* (ε,d_5^3)*. 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_7^1*, and the last is contained in decision dnode* d_7^2*; 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 is constructed that represents* $\{\emptyset\}$ *and 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* $(\varepsilon,\varepsilon)$ *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_{10}^1 *and* d_{10}^2*, 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. Then, we let them be contained in decision dnodes* d_{15}^1 *and* d_{15}^2*, 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}*. It 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 4.4.1. The function 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 the tnodes of a tree decomposition from the bottom, where the process is based on the types of tnodes. In Algorithm 4.4.1, 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.

Algorithm 4.4.1 Construct(D, i)

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

4.5 Correctness and Complexity

Theorem 4.5.1. *The proposed method correctly compiles the independent sets contained in the input graph.*

Proof. In the hierarchy of nodes presented in Section 4.4, 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 3.1.1. Consequently, it is sufficient to prove that the hierarchy meets the requirements of structured decomposability and determinism. Firstly, 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 $(\boldsymbol{X}, \boldsymbol{Y})$-decomposition. Thus, the hierarchy of Section 4.4 satisfies structured decomposability. Next, we prove that the hierarchy of Section 4.4 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 4.5.2. *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 2.1.3. Because the time required to construct each dnode corresponding to each state is constant, the time complexity for constructing all the dnodes is $O(2^W|T(\mathcal{T})|)$. This is also the same as the time required by the method described in Section 2.1.3. 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 $O(2^W|T(\mathcal{T})|)$. □

4.6 Experiments

In this section, we describe the results of our experimental evaluation. To compare the performance of the proposed method, we conducted conventional 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 2.4.3. We implemented the frontier-based search with using the implementation of TdZdd[2].

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-PACE2017-public-2016-12-02[3], 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[4], 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.[5] Therefore, we uniformly terminated the calculation for each graph in two seconds.

[2]https://github.com/kunisura/TdZdd

[3]https://people.mmci.uni-saarland.de/~hdell/pace17/ex-instances-PACE2017-public-2016-12-02.tar.bz2

[4]https://github.com/kit-algo/flow-cutter-pace17

[5]https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/

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 [Inoue and Minato, 2016], which equals to the path-width [Kinnersley, 1992]. We used our own implementation for the beam search with following the method detailed in Inoue et. al [Inoue and Minato, 2016]. 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.

Table 4.2. 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

In Table 4.2, we show the instances for the ten smallest and ten largest of the path-width / tree-width ratio. Further, in Table 4.3 and Table 4.4, we show the experimental results of construction, reduction. Also, in Table 4.5, we show the results of query and transformation. The rest of the instances is omitted due to lack of space.

[10] In these instances, path-width is smaller than tree-width because we used heuristics to calculate both values. In the following three tables, Table 4-6, we took the path decomposition as the tree decomposition i.e., $\mathrm{PW}/\mathrm{TW} = 1.00$.

Table 4.3. 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	>1h		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	>1h		3.04
ex155	10,002	12,799,222	>1h		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	>1h		3.53
ex149	75	123,749	>1h		3.92

In Table 4.2, 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 vertex separation number, which is gained with the gnode order in the last paragraph, is equal to the path-width [Kinnersley, 1992].

On the other hand, in the columns headed Our Method and Frontier-Based Search of Table 4.3 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 ">1h". Further, the columns headed Reduction of Table 4.4 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, the time required for model counting is shown. The instances that did not finish running within one hour are marked as ">1h". The columns headed Random show the time required for the random sampling. We tried random sampling both 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 prepossess of binary searches in a

Table 4.4. 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	>1h		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	>1h		3.04
ex155	15,000	998,317	>1h		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	>1h		3.53
ex149	71	42,867	>1h		3.92

random walk. (Cf. Section 3.2.) 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 bosh

Table 4.5. 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	>1h		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	>1h		3.04
ex155	919	1,116	35	>1h		3.05
ex081	0	1	1	9	0	3.17
ex011	10	12	6	5,917	91	3.30
ex129	80	96	8	>1h		3.53
ex149	31	36	8	>1h		3.92

for the time needed for the compilation and the size of the output. As we pointed in Section 4.3, 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.

Figure 4.3. Compilation time of proposed method vs. Tree-width.

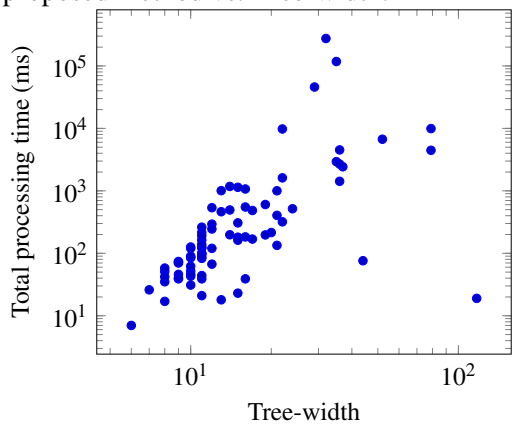
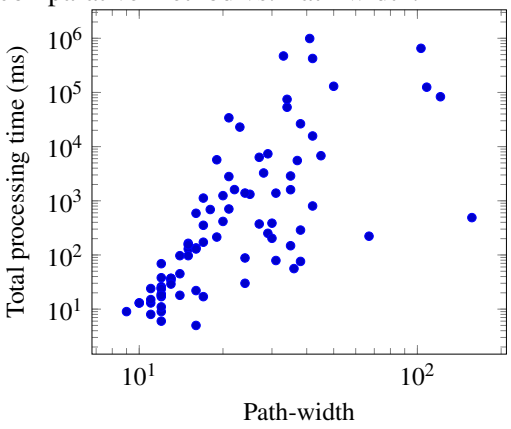


Figure 4.4. Compilation time of comparative method vs. Path-width.



we compared the time required for compilation by proposed method with the tree-width of the input graphs in Fig. 4.3, and the time required for compilation by comparative method with the path-width of the input graphs in Fig. 4.4. In addition, we compared the size of the output of proposed method with tree-width of the input graphs in Fig. 4.5, and the size of the output of compared method with path-width of the input graphs in Fig. 4.6. Former two graphs, Fig. 4.3 and Fig. 4.4, 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, Fig. 4.5 and Fig. 4.6, show that the size of the output of those two methods are in proportion to the tree-width and path-width, respectively. Fig. 4.3 and Fig. 4.5 support the theoretical time complexity value discussed in Section 4.5.

In Fig. 4.7, 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-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. 4.8. 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 has a tendency to be orders of magnitude smaller than that of comparative method.

4.7 Concluding Remarks

In this chapter, we have proposed a method for compiling the independent sets of a given graph. In the proposed method, we have used structured Z-d-DNNF. Additionally,

Figure 4.5. Size of the output of proposed method vs. Tree-width.

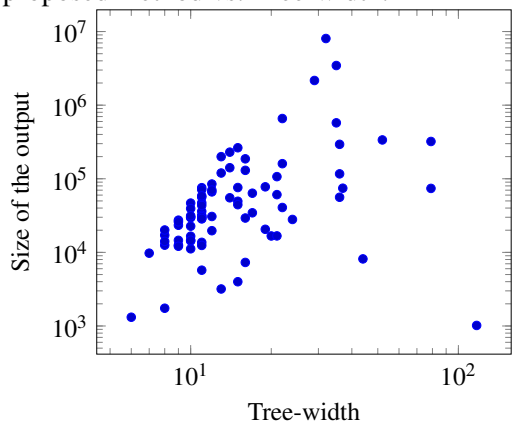
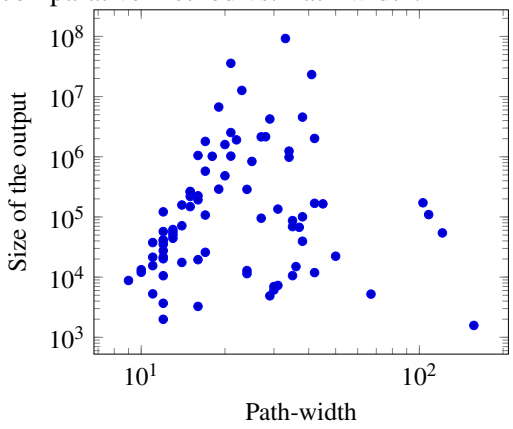


Figure 4.6. Size of the output of comparative method vs. Path-width.



we have applied the previous method to compute the size of the maximum independent set with using tree decomposition.

As a result, the independent sets were computed with the time complexity equivalent to the previous method. In addition, after the compilation, we could efficiently do the random sampling or counting of the independent sets with dynamic programming on a structured Z-d-DNNF. The experimental results showed 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.

Our future work will be as follows.

Firstly, it is to apply our method to compile graph-substructure other than independent sets. It is an interesting work, because the methods to solve the optimization problems with using tree decomposition are applied to varieties of problems on graph-substructure. For example, it is known that an independent set is maximum if and only if it is dominant [Hayase et al., 1995]. Therefore, we could expand our method to compile maximum independent set by applying a previous method for computing the size of the minimum dominant set [Hedetniemi and Laskar, 1985].

Secondly, it is to compile with other recent tractable indexing structure such as Zero-suppressed SDD (ZSDD) [Nishino et al., 2016]. Because ZSDD is a subset tractable indexing structure of structured Z-d-DNNF and supports more operations than the latter. In addition, ZSDD supports operations called apply, that cover a variety of transformations or queries. Therefore, the ZSSD's output could be applied broader than structured Z-d-DNNF.

Figure 4.7. Compilation time ratio of comparison method to proposed method vs. Ratio of tree-width to path-width.

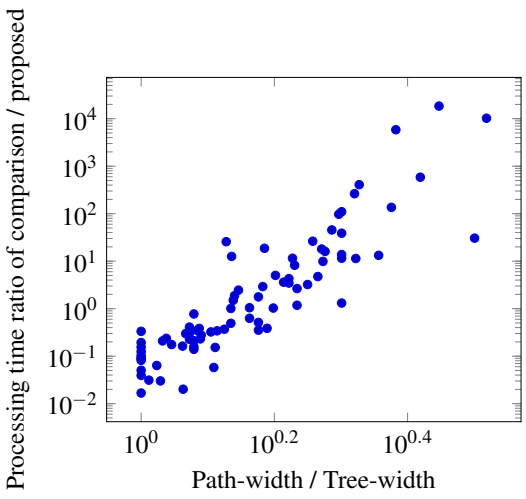
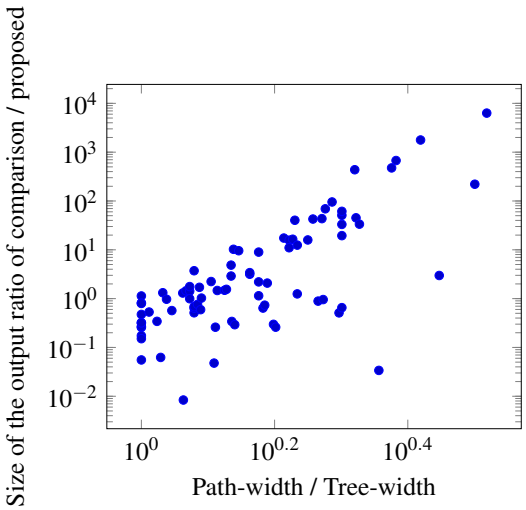


Figure 4.8. Size of the output ratio of comparison method to proposed method vs. Ratio of tree-width to path-width.



Chapter 5

Merging Frontier-Based Search to Compile *S*-*T* Paths

In this chapter, we focus on the compilation of the *s*-*t* paths in an input graph [Sugaya et al., 2017, Sugaya et al., 2018]. In order to compile the *s*-*t* paths, we conduct an exhaustive search on edges of an input graph, instead of gnodes. It makes a difference between this and previous chapter's compilation methods. Here, we explain the overview of the compilation along the three features: frontier, the construction of a vtree, and the management of the states of the searching branches.

1. Frontier: To compile the *s*-*t* paths, in stead of bag of tree decomposition, we use an e-separator of branch decomposition as frontier. An e-separator plays the same role as a frontier gnode set of conventional frontier-based search.

2. The construction of a vtree: To exploit the branch width of an input graph, it is also necessary to device the construction of a vtree. However, for the compilation of *s*-*t* paths, it is enough to use the preceding construction method of a vtree which is presented by Nishino et al. [Nishino et al., 2017].

3. The management of the states of searching branches: As same as the compilation of independent sets in Chapter 4, we need to merge the states of the searching branches for the compilation of *s*-*t* paths. We explain how to merge the states in the process of the compilation.

5.1 Overview

In this section, we describe the algorithm for compiling the *s*-*t* simple paths contained in the graph $G = (V, E)$. Firstly, we present the overview of our algorithm. Let s be the

start point of the simple paths and t be the end point. We consider a graph-substructure

$$P(s,t) = \{A \subseteq E \mid G[A] \textit{ is a simple path connecting } s \textit{ and } t.\}. \tag{5.1}$$

Our aim is to efficiently compile $P(s,t)$.

In our method, the inputs are a graph $G = (V,E)$, and its corresponding vtree Φ. The element corresponding to each leaf of the vtree Φ is a certain edge of the input graph. On the other hand, the output is a structured Z-d-DNNF that represents $P(s,t)$ and respects the vtree Φ.

We conduct depth first search (DFS) from the root of the input vtree Φ. In the process of backtrack of DFS, we construct a structured Z-d-DNNF in bottom up manner, which represents the *s-t* path fragments being processed. At the leaf α of the vtree Φ, we construct the dnodes that represent $\{e_\alpha\}$ or $\{\emptyset\}$. At the internal node β of the vtree Φ, we construct the ordered set with one level lower dnodes $d^i_{\beta^l}$ and $d^i_{\beta^r}$ that represents the path fragments $\langle d^i_{\beta^l}\rangle \sqcup \langle d^i_{\beta^r}\rangle$. Then we include them in the dnode d_β. This dnode d_β represents the path fragments that are equal to the union of the path fragments of the elements $\{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$ with one level lower. At the end of our algorithm, we obtain the structured Z-d-DNNF whose root dnode respects the root of the vtree Φ, where the root dnode represents all the *s-t* paths of the input graph G.

When we have more than one dnodes that share the same remaining search space, we merge these dnodes. To effectively conduct merge, we apply *mate* technique of frontier-based search that we explained in Section 2.4.2. In addition, we also use *mates* to judge some pass fragments accomplished *s-t* paths, or to discard pass fragments that are not expected to be *s-t* paths no more. In this way, we construct the structured Z-d-DNNF representing $P(s,t)$ with the method referred to as merging frontier-based search. Then we reduce it and produce the output.

In the following section, we first define the concept of frontier and the configuration of merging frontier-based search. Secondly, we describe the method of compiling an *s-t* simple path by merging frontier-based search and its method of merging mates. Finally, we show the algorithm's correctness and complexities.

5.2 Frontier and Configuration

Firstly we define the frontier in a vtree.

Definition 5.2.1. *Let gnode sets of $G[E_\alpha]$ and $G[E \setminus E_\alpha]$ be U_1 and U_2 respectively. We call the gnode set $U = U_1 \cap U_2$ as the frontier gnode.*

That is, the frontier gnodes is a set of gnodes that are common to the edges corresponding to the leaves of a subtree of a vnode and the remaining edges. If α is a leaf,

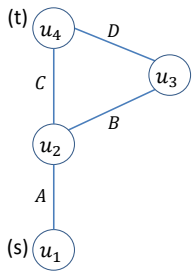


Figure 5.1. Example of an input graph. U_1 is the start point s and u_4 is the end point t.

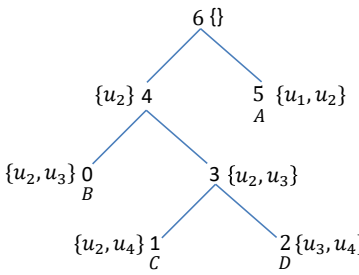


Figure 5.2. Vtree corresponding to the graph of Fig. 5.1. The edge corresponding to a vnode is shown below it. The frontier gnodes corresponding to a vnode is shown next to it.

$U = \{u, w\}$ for $e_\alpha = \{u, w\}$. In addition, if α is the root of a vtree, $U = \emptyset$. For example, in a vtree of Fig. 5.2, the frontier gnodes of the vnode 1 is a set $\{u_2, u_4\}$ and the frontier gnodes of the vnode 2 is a set $\{u_3, u_4\}$. In the vnode 3, the edges corresponding to the leaves of the subtree of vnode 3 are C and D. The rest of the edges are A and B. Therefore the frontier gnodes is a set $\{u_2, u_3\}$.

In the following, the frontier gnode set of a vnode α is denoted by F_α. The input vtree of our method is generated by the branch decomposition of the input graph. Each frontier gnode set of a vtree corresponds to an e-separator of the branch decomposition. Consequently, the maximum size W of the frontier gnodes matches the branch width of the graph [Nishino et al., 2017]. An example of an input graph is shown in Fig. 5.1 and an example of a corresponding vtree and frontier gnode set is shown in Fig. 5.2.

In our method, we conduct a bottom-up search from the leaves to the root of a vtree. At each vnode α, the proposed method constructs a set of dnodes D_α. They represent the candidates of $P(s,t)$ among all subsets in E_α. Each $d \in D_\alpha$ represents a family of sets $\langle d \rangle \subseteq 2^{E_\alpha}$.

A subgraph $G[A]$ for $A \subseteq E_\alpha$ has the property of $G[A] \in P(s,t)$ or $G[A] \notin P(s,t)$. For efficiency of calculation, the following concept of *equivalence* is introduced.

Definition 5.2.2. *For $A, A' \subseteq E_\alpha$, $G[A]$ and $G[A']$ are equivalent if for all $B \subseteq (E \setminus E_\alpha)$, $G[A \cup B]$ and $G[A' \cup B]$ have the same set of extensions to fully completed simple paths. Dnodes d, d' are equivalent if $G[E]$ and $G[E']$ are equivalent for all $E \in \langle d \rangle$ and $E' \in \langle d' \rangle$. We denote this condition by $d \equiv d'$.*

In order to efficiently judge the equivalency, we introduce *configuration* ϕ [Maehara et al., 2017]. The configuration of the family of sets $\langle d_\alpha \rangle$ is defined with the mates of the frontier gnode sets F_α and the start point s and the end point t. We added s and t to the frontier

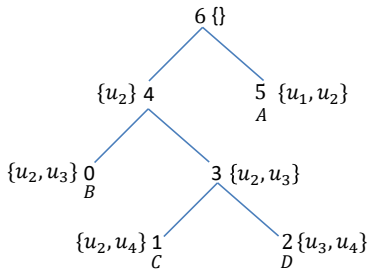


Figure 5.3. A Structured Z-d-DNNF that represents all *s-t* paths in the graph of Fig. 5.1, $\{\{A,C\},\{A,B,D\}\}$.

gnodes, in order to make it easy to judge the accomplishment of a *s-t* pass. The mate of a family of sets of d_α is denoted by $mate_{\langle d_\alpha \rangle}$. The notation is determined to make sure that configuration is calculated with the frontier gnodes F_α of a vtree α that is respected by d. The configuration of $\langle d_\alpha \rangle$ is defined as follows:

$$\phi(\langle d_\alpha \rangle) := \{(v, mate_{\langle d_\alpha \rangle}[v]) | v \in F_\alpha \cup \{s,t\}\}. \tag{5.2}$$

The family of sets with equal configurations are equivalent.

Theorem 5.2.1. $\phi(\langle d_\alpha \rangle) = \phi(\langle d'_\alpha \rangle) \Rightarrow d_\alpha \equiv d'_\alpha$

Proof. The (u_1,u_2)-path of G that connects gnode $u_1 \in (U_1 \setminus U)$ and gnode $u_2 \in (U_2 \setminus U)$ always contains the gnode of frontier U. Let $A, A' \subseteq E_\alpha$. Then, if the degree and the connected component of each gnode $u \in U$ are equal, $G[A \cup B]$ and $G[A' \cup B]$ are equivalent for all $B \subseteq (E \setminus E_\alpha)$. Since the degree and connected component of each gnode u can be judged with $mate_{\langle d_\alpha \rangle}[u]$, if $mate_{\langle d_\alpha \rangle}[u]$ is equal in A and A' for all $u \in U$, $G[A \cup B]$ and $G[A' \cup B]$ are equivalent. □

Since configuration ϕ depends only on the mate and is independent of the choice of A or A', the well-definedness of configuration is also proven.

5.3 Merging Frontier-based Search for *S-T* Paths

In this section, we present an overview of the *s-t* simple path compilation by merging frontier-based search. Firstly, we show the dnodes, their semantics, configuration, and set D_α created by the frontier-based search.

1. If vnode α is a leaf:
   - Dnode: Terminal dnode $d = \varepsilon$ or $d = e_\alpha$.
   - Semantics: $\langle \varepsilon \rangle = \{\emptyset\}, \langle e_\alpha \rangle = \{e_\alpha\}$.
   - Configuration: $\phi(\langle \varepsilon \rangle) \neq \phi(\langle e_\alpha \rangle)$.
   - $D_\alpha = \{\varepsilon, e_\alpha\}$.

2. If vnode β is an internal node:

   - Dnode: Decision dnode d_β, where $d_\beta = \{(d^1_{\beta^l}, d^1_{\beta^r}), \ldots, (d^n_{\beta^l}, d^n_{\beta^r})\}$. Here, $d^1_{\beta^l}, \ldots, d^n_{\beta^l}$ denote decision dnodes or terminal dnodes respecting β^l and $d^1_{\beta^r}, \ldots, d^n_{\beta^r}$ denote those respecting β^r.
   - Semantics: $\langle d \rangle = \bigcup_{i=1}^{n} \langle d^i_{\beta^l} \rangle \sqcup \langle d^i_{\beta^r} \rangle$.
   - Configuration: For all $1 \leq i < j \leq n$, $\phi(\langle (d^i_{\beta^l}, d^i_{\beta^r}) \rangle) = \phi(\langle (d^j_{\beta^l}, d^j_{\beta^r}) \rangle)$.
     In addition,
   - $D_\alpha = \{d^1, \ldots, d^m\}$. Here, $d^1, \ldots, d^m$ are decision dnodes or terminal dnodes.
   - Configuration: For all $1 \leq i < j \leq m$, $\phi(\langle d^i \rangle) \neq \phi(\langle d^j \rangle)$.

If vnode α is a leaf of the vtree, we create two terminal dnodes: one corresponding to e_α and the other $\emptyset$. Subsequently, we calculate each configuration and set $D_\alpha = \{e_\alpha, \varepsilon\}$. If vnode β is an internal node of the vtree, by inductive assumption, we obtain the set of decision dnodes or terminal dnodes $D_{\beta^l} = \{d^1_{\beta^l}, \ldots, d^n_{\beta^l}\}, D_{\beta^r} = \{d^1_{\beta^r}, \ldots, d^n_{\beta^r}\}$. Subsequently, we take each pair $(d^i_{\beta^l}, d^j_{\beta^r})$ from D_{β^l} and D_{β^r}, then examine the left and right mates. Judging from the mate of each gnode, when we find that a cycle is formed or if the degree is considerably large, we discard the pair. Then, if $\langle d \rangle$ accomplishes simple *s-t* paths, it is added to D_{root}; otherwise, it is added to D_β. Note that the configurations of all the elements contained in a decision dnode d are equal, whereas those of the decision dnodes contained in a D_α are different. We repeat the above process until the vtree root. At the root of the vtree, since $F_{root} = \emptyset$, s and t always go out of the frontier. Thus, if the *s-t* simple path is not accomplished in a decision dnode, pruning conditions (1), (4), and (6) are always satisfied.

In merging frontier-based search, a candidate that is judged not to be a solution is discarded from the structured Z-d-DNNF under construction. In addition, if a candidate is determined to accomplish an *s-t* path and contained in D_{root}, we exclude it from the search space. On the other hand, in the preceding method top-down construction, even if a candidate that is judged not to be a solution, the corresponding dnodes remain in ZSDDs under construction. These dnodes are to be removed in the reduction process after the compilation. Therefore, the search space for compilation can be greatly reduced compared to the top-down construction method.

The method described above is shown in Algorithm 5.3.1 and Algorithm 5.3.2. We first execute RecursiveConstruct(G, r, α, D) and construct a structured Z-d-DNNF in D that represents $P(s,t)$ of the input graph G. Then, we execute ReduceEx(D), where, in addition to the reduction, we change a decomposition $\{(X,d),(\varepsilon,d)\}$ to $\{\pm X, d\}$ and a decomposition $\{(d,X),(d,\varepsilon)\}$ to $\{d, \pm X\}$. (Algorithm 5.3.1).

We explain RecursiveConstruct(G,r,α,D) in Algorithm 5.3.2. If α^l is a leaf, we store the terminal dnode that corresponds to $\{e_{\alpha^l}\}$ or $\{\emptyset\}$ into D_{α^l} (line 2). That is, LeafChild outputs e_{α^l} if the second parameter is true; otherwise, it outputs ε. If α^l is not a leaf, we call RecursiveConstruct(G,r,α,D) with α^l as a third parameter (line 4). The same occurs when α^r is not a leaf (from line 6 to line 10).

Next, for all pairs of decision dnodes included in D_{α^l} and D_{α^r}, we execute DecompChild$(\alpha^l,\alpha^r,d_{\alpha^l},d_{\alpha^r})$ (line 14). In DecompChild$(\alpha^l,\alpha^r,d_{\alpha^l},d_{\alpha^r})$, we create a mate corresponding to each combination of decision dnodes of d_{α^l} and d_{α^r}. Then, we generate the decision dnode d and consequently include element $(d_{\alpha^l},d_{\alpha^r})$ in it. Here, we output $(d,\top)$ if $\langle d\rangle$ accomplishes *s-t* simple paths, $(d,\bot)$ if there is no chance of $\langle d\rangle$ becoming *s-t* simple paths; otherwise, we output $(d,null)$.

If *term* is not $\bot$, we execute UniqueNode(d,D) (line 16, 19). UniqueNode(d,D) outputs the corresponding dnode if d is a terminal dnode. Otherwise, it checks whether some d' in D_α is equivalent to the dnode d. If such d' exists, we add all elements in d to d' and then output d'. If it does not exist, we add d to D_α and output d.

In the proposed method, if a $\langle d\rangle$ accomplishes *s-t* simple paths, we can exclude it from the search space by including the corresponding dnode in D_{root}. Furthermore, unexpected dnodes can be discarded from the search space. Consequently, the search space is smaller than in the top-down method with ZSDD. We experimentally demonstrate this in Section 5.6.

Algorithm 5.3.1 *Merging frontier based search algorithm*

Require: Graph $G=(V,E)$, *root* of a vtree.
Ensure: Z-st-d-DNNF representing $P(s,t)$ of G.
1: $\forall\alpha\in$ vtree$, D_\alpha\leftarrow\emptyset$
2: RecursiveConstruct$(G,root,root,D)$ // Constructs a structured Z-d-DNNF representing $P(s,t)$ in D
3: ReduceEx(D) // Changing $X,\varepsilon\Rightarrow\pm X$ and reduction
4: **return** D

5.4 Merging mates

In this section, we explain a method to merge mates. Let $h^i_\beta=(d^i_{\beta^l},d^i_{\beta^r})$. Merging mates is a method to obtain $mate_{\langle h^i_\beta\rangle}$ on the assumption that $mate_{\langle d^i_{\beta^l}\rangle}$ and $mate_{\langle d^i_\beta\rangle}$ are correctly calculated. This process is necessary in DecompChild of Algorithm 5.3.2. Firstly, we construct mates $mate_{\langle h^i_\beta\rangle}[]$ from the pair of mates, which are $mate_{\langle d^i_{\beta^l}\rangle}[]$ of $\langle d^i_{\beta^l}\rangle$ and $mate_{\langle d^i_{\beta^r}\rangle}[]$ of $\langle d^i_{\beta^r}\rangle$. Then, we call the function *Merge*,

$$mate_{\langle h^i_\beta\rangle}[] := \text{Merge}(mate_{\langle d^i_{\beta^l}\rangle}[], mate_{\langle d^i_{\beta^r}\rangle}[]). \tag{5.3}$$

Algorithm 5.3.2 RecursiveConstruct(G, r, α, D)

Require: Graph G, *root* of a vtree r, vnode α, D
Ensure: Z-st-d-DNNF representing (fragments of) $P(s,t)$ of G.

if α^l is a leaf **then**
  $D_{\alpha^l} \leftarrow \{\text{LeafChild}(\alpha^l, d_{\alpha^l}, \textbf{false})\} \cup \{\text{LeafChild}(\alpha^l, d_{\alpha^l}, \textbf{true})\}$
else
  RecursiveConstruct(G, r, α^l, D) // α^l is an internal node.
end if
if α^r is a leaf **then**
  $D_{\alpha^r} \leftarrow \{\text{LeafChild}(\alpha^r, d_{\alpha^r}, \textbf{false})\} \cup \{\text{LeafChild}(\alpha^r, d_{\alpha^r}, \textbf{true})\}$
else
  RecursiveConstruct(G, r, α^r, D) // α^r is an internal node.
end if
// Make pairs with $\forall d_{\alpha^l} \in D_{\alpha^l}$ and $\forall d_{\alpha^r} \in D_{\alpha^r}$.
for d_{α^l} *in* D_{α^l} **do**
  for d_{α^r} *in* D_{α^r} **do**
    $(d, term) \leftarrow \text{DecompChild}(\alpha^l, \alpha^r, d_{\alpha^l}, d_{\alpha^r})$
    if $term = \top$ **then**
      $d' \leftarrow \text{UniqueNode}(d, D)$ // An s-t simple path is accomplished.
      $D_r \leftarrow \{d'\} \cup D_r$
    else if $term \neq \bot$ **then**
      $d' \leftarrow \text{UniqueNode}(d, D)$ // Continue.
      $D_\alpha \leftarrow \{d'\} \cup D_\alpha$
    end if
  end for
end for

In conventional frontier-based search, we add an edge one by one to the solution candidates. For example, consider that edges $e = \{u_1, u_2\}$ of the input graph are newly included in the s-t simple path. Assuming that $mate[u_1] = \hat{u_1}, mate[u_2] = \hat{u_2}$, we update as $mate[u_1] \leftarrow 0$, $mate[u_2] \leftarrow 0$, $mate[\hat{u_1}] \leftarrow \hat{u_2}$, and $mate[\hat{u_2}] \leftarrow \hat{u_1}$ [Knuth, 2011]. However, in our merging frontier-based search, we need to merge mates of two families of sets instead of adding an edge. Therefore, we need to calculate $\{mate_{\langle h^i_\beta \rangle}[u] \mid u \in (F_{\beta^l} \cup F_{\beta^r} \cup \{s,t\})\}$ from $\{mate_{\langle d^i_{\beta^l} \rangle}[u^l] \mid u^l \in (F_{\beta^l} \cup \{s,t\})\}$ and $\{mate_{\langle d_{\beta^r} \rangle}[u^r] \mid u^r \in (F_{\beta^r} \cup \{s,t\})\}$. When an s-t simple path cannot be accomplished, we prune the calculation of mates.

Firstly, we consider pairs of arbitrary edge sets $\exists E^l \in \langle d_{\beta^l} \rangle$ and $\exists E^r \in \langle d_{\beta^r} \rangle$. Let the degree of gnode u of graph $G[E^l]$ be $deg^l(u)$ and that of $G[E^r]$ be $deg^r(u)$. Since $E^l \cap E^r = \emptyset$ from the property of a vtree, the order $deg(u)$ of gnode u in $G[E^l \cup E^r]$ is equal to $deg^l(u) + deg^r(u)$. Therefore, by examining $mate_{\langle d_{\beta^l} \rangle}, mate_{\langle d_{\beta^r} \rangle}$ for each gnode u of $G[E^l], G[E^r]$, the degree of each gnode u of $G[E^l \cup E^r]$ can be calculated. As a result, it can be determined whether the pruning condition is satisfied. If $mate_{\langle d_{\beta^l} \rangle}[u] = mate_{\langle d_{\beta^r} \rangle}[u] (\neq u)$, a cycle occurs and pruning condition (3) holds. If $mate_{\langle d_{\beta^l} \rangle}$ and

$mate_{\langle d_{\beta^r}\rangle}$ do not meet the pruning condition, we can update them. Here, to identify whether the opposite side is from the left or the right, gnodes are denoted by v^l or v^r. Assuming that $mate_{\langle d_{\beta^l}\rangle}[u] = v^l$ and $mate_{\langle d_{\beta^r}\rangle}[u] = v^r$, we let $mate_{\langle d_{\beta^l}\rangle}[u] \leftarrow 0, mate_{\langle d_{\beta^r}\rangle}[u] \leftarrow 0, mate_{\langle d_{\beta^l}\rangle}[u^l] \leftarrow v^r$, and $mate_{\langle d_{\beta^r}\rangle}[u^r] \leftarrow v^l$. Since $\forall E^l \in \langle d_{\beta^l}\rangle$ and $\forall E^r \in \langle d_{\beta^r}\rangle$ have the same configuration, the above discussion holds regardless of the selection of E^l, E^r (well-definedness).

We show details of the merging method of mates in Algorithm 5.3.3. We judge $\forall u \in F_{\beta^l} \cap F_{\beta^r}$ sequentially (from line 1). If the degree of one of the left and right mates is zero, we use the value of the other mate (from line 2 to 7). Otherwise, if the sum of the degrees of the left and right mates is greater than two, it corresponds to pruning condition (2) (from line 8 to 10). In the rest of the cases, the degrees of u and u' are one. If $u = s$ or $u = t$, it corresponds to the pruning condition (1) (from line 13 to 15). If $mate_{\langle d_{\beta^l}\rangle}[u] = mate_{\langle d_{\beta^r}\rangle}[u]$, it corresponds to pruning condition (3) (from line 16 to 18). If it does not meet any of the conditions above, we update $mate_{\langle d_{\beta^l}\rangle}$ and $mate_{\langle d_{\beta^r}\rangle}$, then memorize whether the gnode belongs to E^l or E^r. (from line 19 to line 24).

After processing for $\forall u \in F_{\beta^l} \cap F_{\beta^r}$, we update $mate$ for $\forall u \in (F_{\beta^l} \cup F_{\beta^r} \cup \{s,t\})$ (from line 27 to line 35). When $mate_{\langle d_{\beta^l}\rangle}[u] \neq u$ and $mate_{\langle d_{\beta^r}\rangle}[u] \neq u$, it means that u becomes a midpoint after merging; thus, $mate_{\langle d_{\beta^l}\rangle}[u] = mate_{\langle d_{\beta^r}\rangle}[u] = 0$. When $mate_{\langle d_{\beta^l}\rangle}[u] = u$ and $mate_{\langle d_{\beta^r}\rangle}[u] = u$, the degree of u is zero. In the rest of the cases, only one of $mate_{\langle d_{\beta^l}\rangle}[u]$ or $mate_{\langle d_{\beta^r}\rangle}[u]$ is not equal to u. Therefore, in each gnode, it is enough that we substitute $mate[u]$ with the values of $mate^l[u]$ and $mate^r[u]$ that are not equal to u.

After the processing of Algorithm 5.3.3, we judge pruning condition (4) of gnode s,t and pruning condition (5) of $\forall u \in ((F_{\beta^l} \cup F_{\beta^r}) \setminus F)$. Further, if $mate_{\langle d_\beta\rangle}[s] = t$ and $\exists u \in (V \setminus \{s,t\}), mate_{\langle d_\beta\rangle}[u] = u' (u' \in V \text{ and } u' \neq u)$, pruning condition (6) holds. If $mate_{\langle d_\beta\rangle}[s] = t$ and $mate_{d\langle\beta\rangle}[u] = 0$ or $mate_{\langle d_\beta\rangle}[u] = u$ for $\forall u \in (V \setminus \{s,t\})$, it is judged that the *s-t* simple path is accomplished. Through the above processing, $\phi(\langle (d_{\beta^r}, d_{\beta^r})\rangle)$ is obtained from $\phi(\langle d_{\beta^l}\rangle)$ and $\phi(\langle d_{\beta^r}\rangle)$.

At the end of the compilation algorithm, we have:

Theorem 5.4.1. *$D_{root} = \{d_{root}\}$ and $\langle d_{root}\rangle = P(s,t)$.*

Proof. Firstly, we show that D_{root} contains only the decision nodes that represent the *s-t* simple paths in an input graph. Since the frontier gnode is empty at the root, all the dnodes are merged into one dnode d_{root}. In the mates of d_{root}, we have $mate_{\langle d_{root}\rangle}[s] = t$ and $mate_{\langle d_{root}\rangle}[t] = s$ for gnodes $u = s$ or $u = t$, on the other hand, we have $mate_{\langle d_{root}\rangle}[u] = u$ *or* 0 for gnodes $u \neq s$ neither $u \neq t$. Thus, only *s-t* simple paths are contained.

Secondly, we show that D_{root} contains all of the decision nodes that represent the *s-t* simple paths in an input graph. Suppose an *s-t* simple path $G[A]$ is not contained in the family of sets represented by a decision node d_{root}. Then, the decision dnode represent-

ing $G[A]$ appears in the dnode that corresponds to the vnode α which is the lowest common ancestor (LCA) of edge set A. Subsequently, in the process of RecursiveConstruct at this dnode, this dnode shall be judged that it accomplished an *s-t* path then be included in the D_{root}. However, it is not included in D_{root}, which is a contradiction. □

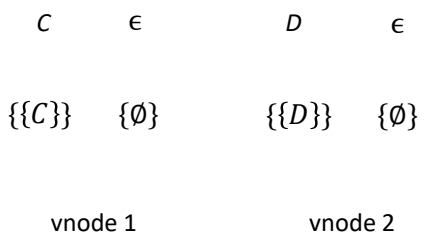


Figure 5.4. Vnode 1 represents $\{\{C\}\}$ and $\{\emptyset\}$, while vnode 2 represents $\{\{D\}\}$ and $\{\emptyset\}$.

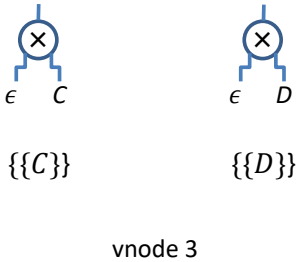


Figure 5.5. Solution candidates at vnode 3, which are $\{\{C\}\}$, $\{\{D\}\}$.

Example 5.4.1. *An example of our method, we take a graph of Fig. 5.1 and a vtree of Fig. 5.2 as an input, and we show the output of the compilation with the structured Z-d-DNNF of all the s-t paths of the input graph in Fig. 5.3. The root dnode of Fig. 5.3 represents the family of sets* $\{\{A,C\},\{A,B,D\}\}$. *We show the process in each vnode in Table 5.1, which part of them will be described in detail below. In the table, mate is shown as m for short, the mates of s and t are omitted in case of obvious , and gnodes* $u_1,u_2,\ldots$ *are shown as* $1,2,\ldots$.

The results of the process at the leaf 0 *of the vtree* Φ *are* $\{\emptyset\}$ *and* $\{\{B\}\}$. *The results of the process at the leaf* 1 *of the vtree* Φ *are* $\{\emptyset\}$ *and* $\{\{C\}\}$ *and those at the leaf* 2 *are* $\{\emptyset\}$ *and* $\{\{D\}\}$ *(Fig. 5.4). Next, respecting the vnode* 3, $\{\{C\}\}\sqcup\{\emptyset\}=\{\{C\}\}$ *and* $\{\{D\}\}\sqcup\{\emptyset\}=\{\{D\}\}$ *are obtained (Fig. 5.5). Since* $mate[4]=4$ *in* $\{\{\emptyset\}\}\sqcup\{\{\emptyset\}\}=\{\{\emptyset\}\}$, *it satisfies the pruning condition (4). In addition, since* $mate[4]=0$ *in* $\{\{C\}\}\sqcup\{\{D\}\}=\{\{C,D\}\}$, *it satisfies the pruning condition (1) (Fig. 5.6). As the frontier*

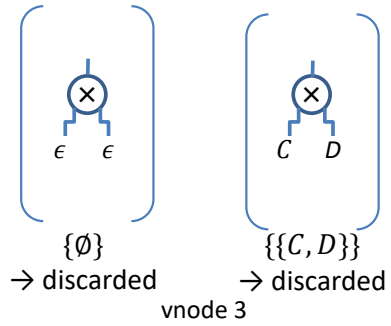


Figure 5.6. $\{\emptyset\}$ and $\{\{C,D\}\}$ are discarded at vnode 4 because each satisfies pruning condition (4) and (1) respectively.

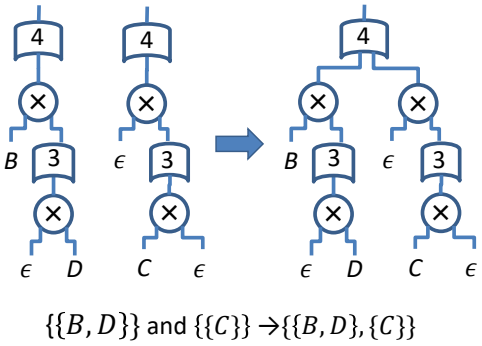


Figure 5.7. Solution candidates at vnode 4, which are $\{\{B,D\}\}$ and $\{\{C\}\}$ (left). Two dnodes are merged and represents $\{\{B,D\},\{C\}\}$ (right).

gnodes is $\{2\}$ *at vnode* 4*, the configuration is* $\{mate[2] = 4\}$ *in both* $\{\{B\}\} \sqcup \{\{D\}\} = \{\{B,D\}\}$ *and* $\{\{C\}\} \sqcup \{\emptyset\} = \{\{C\}\}$*. As a result, both are merged and represented with one dnode(Fig. 5.7). On the other hand, since* $mate[3] = 4$ *in both* $\{\{B\}\} \sqcup \{\{C\}\} = \{\{B,C\}\}$ *and* $\{\{D\}\} \sqcup \{\emptyset\} = \{\{D\}\}$*, they satisfies the pruning condition (5). The results of the process at the leaf* 5 *of the vtree* Φ *are* $\{\{A\}\}$ *and* $\{\emptyset\}$*. At the last vnode* 6*,* $\{\{B,D\},\{C\}\} \sqcup \{\{A\}\} = \{\{A,B,D\},\{A,C\}\}$ *are obtained as all the s-t paths of the input graph (Fig. 5.3). Since* $mate[1] = 1$ *in* $\{\{B,D\},\{C\}\} \sqcup \{\emptyset\} = \{\{B,D\},\{C\}\}$*, it satisfies the pruning condition (4).*

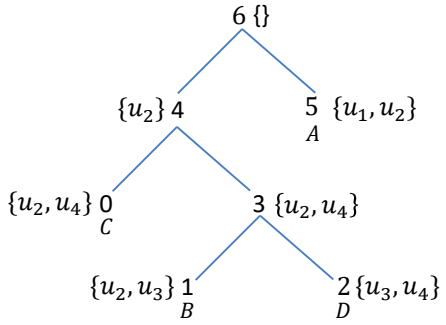


Figure 5.8. Another example of a vtree corresponding to the graph of Figure 5.1, which is used in Table 5.3.

Example 5.4.2. *As the 1st example of the merging of mates, we take a graph of Fig. 5.1*

and a vtree of Fig. 5.2 as an input (Table 5.2). We notice vnodes 4, 5, and 6. When we obtain $\{\{A,B,D\},\{A,C\}\}$ *from* $\{\{B,D\},\{C\}\} \sqcup \{\{A\}\}$*, we need to merge mates of both, i.e.,* $\{mate_{d_4}[1]=1, mate_{d_4}[2]=4, mate_{d_4}[4]=2\}$ *and* $\{mate_{d_5}[1]=2, mate_{d_5}[2]=1, mate_{d_5}[4]=4\}$*. In the process of merging, we obtain* $mate_{d_4}[2] \leftarrow 0$*,* $mate_{d_5}[2] \leftarrow 0$*,* $mate_{d_4}[4^l] \leftarrow 1^r$*, and* $mate_{d_5}[1^r] \leftarrow 4^l$ *(line 23 and 23 of Algorithm 5.3.3). As a result, we obtain* $mate_{d_6}[1] \leftarrow 4$*,* $mate_{d_6}[2] \leftarrow 0$*, and* $mate_{d_6}[4] \leftarrow 1$*, which accomplishes s-t paths (line 27 to 35 of Algorithm 5.3.3).*

Example 5.4.3. *As the 2nd example of the merging mates, we also take a graph of Fig. 5.1 as an input. However, in this case, we take a vtree of Fig. 5.8 (Table 5.3). We notice vnodes 0, 3, and 4. When we merge mates,* $\{mate_{d_0}[1]=1, mate_{d_0}[2]=4, mate_{d_0}[4]=2\}$ *and* $\{mate_{d_3}[1]=1, mate_{d_3}[2]=4, mate_{d_3}[4]=2\}$*, we obtain* $\{\{B,C,D\}\}$ *from* $\{\{B,D\}\} \sqcup \{\{C\}\}$*. However, in this case, a circle takes place. In the process of merging, we find that* $mate_{d_0}[2] = mate_{d_3}[2] = 4$*, which satisfies the pruning condition (3) (line 16 to 18 of Algorithm 5.3.3). Thus, it will be discarded.*

5.5 Correctness and Complexities

The following theorem holds.

Theorem 5.5.1. *A tractable representation constructed with merging frontier-based search satisfies the definition of a structured Z-d-DNNF.*

Proof. Clearly, each node in a tractable representation constructed with merging frontier-based search satisfies one of lists in Definition 3.1.1.

We prove that it satisfies determinism and structured decomposability.

Consider the decision dnodes contained in D_β. Their children D_{β^l} and D_{β^r} respect the children β^l and β^r of a vnode β. Then, from the construction of the algorithm, we have $\forall d_{\beta^l i} \in D_{\beta^l}$, $\langle d_{\beta^l i}\rangle \subseteq 2^{E_{\beta^l}}$, and $\forall d_{\beta^r j} \in D_{\beta^r}$, $\langle d_{\beta^r j}\rangle \subseteq 2^{E_{\beta^r}}$. Thus, the representation satisfies structured decomposability.

On the other hand, for each vnode β, the decision dnodes or terminal dnodes contained in D_β have different configurations. Thus, they represent mutually exclusive families of sets . Hence, the representation satisfies determinism. □

From Theorem 3.2.1, 5.2.1, 5.4.1 and 5.5.1, the following theorem holds.

Theorem 5.5.2. *The proposed method correctly compiles* $P(s,t)$ *of the input graph.*

Next, in Theorem 5.5.3, we estimate the size of the structured Z-d-DNNF that is constructed by the proposed method. Then, in Theorem 5.5.4, we estimate the running time of our algorithm.

Theorem 5.5.3. *The size of the structured Z-d-DNNF constructed with our algorithm is* $O(|E|W^{2W})$.

Theorem 5.5.4. *The running time of our algorithm is* $O(W|E|W^{2W})$.

Proof of Theorem 5.5.3. If a vnode β is an internal node and dnode d_β respects β, since a mate $mate_{\langle d_\beta \rangle}[u]$ of each gnode u can take W values, the size of each D_β is $O(W^W)$. On the other hand, if vnode α is a leaf, the size of each D_α is at most two. Since an element is given by these joins, the size is $O(W^{2W})$. Since there are $|E|-1$ internal vnodes, the total size is $O(|E|W^{2W})$. □

Proof of Theorem 5.5.4. The proposed method consists of construction and reduction.

Regarding the construction, we need to estimate the running time of RecursiveConstruct in Algorithm 5.3.2 of each edge. Firstly, The running time of LeafChild is $O(1)$. Secondly, the running time of one merging process of DecompChild and the running time to determine the pruning condition are $O(W)$. Besides, since the cardinality of mates contained in $Z[\beta^l]$ and $Z[\beta^r]$ is $O(W^W)$, the size the double loop from line 12 to line 23 in Algorithm 5.3.2 is $O(W^{2W})$. Therefore, the running time per RecursiveConstruct is $O(WW^{2W})$. Thus, the running time to construct $P(s,t)$ is $O(W|E|W^{2W})$ in total.

Next, regarding the reduction, since it takes time proportional to the cardinality of nodes of the constructed structured Z-d-DNNF, the order of time is $O(|E|W^{2W})$.

Based on the above, the total running time is $O(W|E|W^{2W})$. □

5.6 Experiments

In this section, we describe the result of the experimental evaluation. We compared the proposed method with the top-down method of Nishino et al [Nishino et al., 2017]. For graphs, We used the data set of TSPLIB [Cook and Seymour, 2003], which consists of undirected graphs obtained by Delaunay division, and the RomeGraph data set[1]. For RomeGraph, we used only the instances in which the gnode number is 100. In both types of data, we set the first gnode in lexicographic order as the start point of the *s-t* simple path and the last gnode as the end point. On the other hand, for vtrees, we created them by a branch decomposition obtained using the clustering heuristic method proposed by Cook et al. [Cook and Seymour, 2003].

The experiment environment comprised a system with Intel Xeon CPU E7-8837(2.67 GHz) and 1.5 TB main memory. Both methods were implemented in C++ and built with g++ 4.8.5. In the experiment, we measured the time taken to construct the *s-t* simple paths of a graph, and the time taken for the reduction. Also, we recorded the number of nodes of a structured Z-d-DNNF and a ZSDD output.

[1] http://www.graphdrawing.org/download/rome-graphml.tgz

From the result of the experiment, the proposed method ran faster at 108 out of 136 instances among the instances, where at least one of the methods was processed to the end. Moreover, at all the instances where both the methods were processed to the end, the number of the structured Z-d-DNNF nodes after the enumeration was less than the number of ZSDD nodes. On the other hand, the structured Z-d-DNNF size after the reduction was smaller at 13 out of 136 instances.

The running time for performing only the construction is shown in Fig. 5.9, and the total running time including that for reduction is shown in Fig. 5.10. In these figures, the horizontal axis shows the running time of the top-down method, while the vertical axis shows the running time of our method. The running time for performing only the construction with the proposed method is shorter for most of the data. The trend of the total running time remains the same.

In addition, the node size for construction only is shown in Fig. 5.11, and the node size including the reduction is shown in Fig. 5.12. The node size for construction only is smaller in the proposed method than in the top-down method. On the other hand, when the reduction is included, the node size in the top-down method becomes smaller.

In Table 5.4, the results of the proposed method and the top-down method are listed for instances with the ten largest *s*-*t* simple paths. In the table, the columns named Construction indicate the running time or the number of nodes associated with only construction. On the other hand, the columns named Reduction indicate the running time or the number of nodes associated with only reduction. The column named Vtree lists the running time required to construct a vtree from an input graph. The instances that did not finish running within 10 min are marked as TO. The total running time of the proposed method is up to approximately 20 times less than that of the top-down method, which is achieved with grafo11227.100.

In addition, for the data of Table 5.4, we measured the running time of a conventional frontier-based search, Simpath [2]. However, it timed out for all data. On the other hand, even if the running time to generate the vtree from the input graph is included, the total running time of our method increases only slightly.

5.7 Discussion

Firstly, we explain about the case of only enumeration. The proposed method had shorter execution time and smaller node size than the top-down method. That would because, as shown in Section 5.3, our method is possible to remove the candidate from a search space as soon as they are judged unlikely to become a solution.

[2]The implementation was obtained from https://github.com/kunisura/TdZdd.

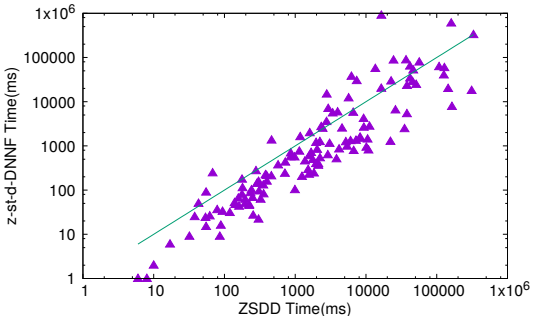


Figure 5.9. Only construction.

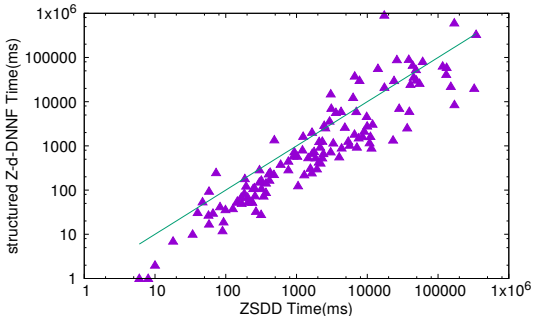


Figure 5.10. Reduction Included.

Running time.

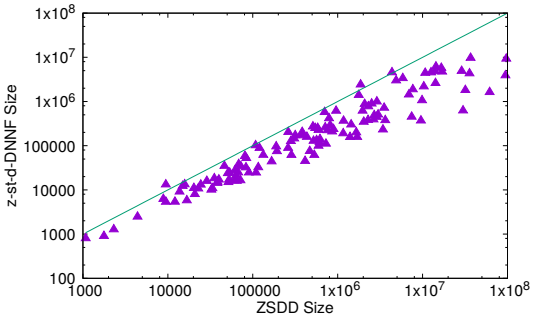


Figure 5.11. Only construction.

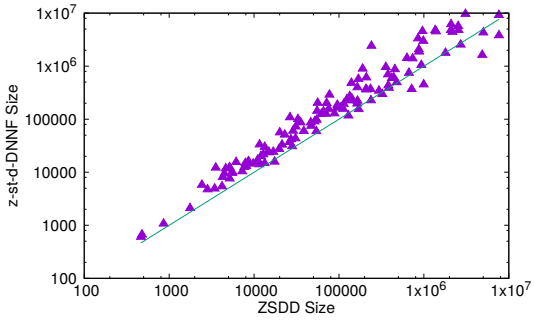


Figure 5.12. Reduction Included.

Node size.

Next, we explain about the case of reduction. Also in the case of reduction, the execution time of the proposed method is similarly shorter than the top-down method. However, the node size in the top-down method after reduction is smaller than the proposed method. There would be two reasons. Firstly, before reduction, the ZSDDs composed by the top-down method contain many candidates that are unlikely to become solutions (Section 5.3). Thus, the node size can be greatly reduced by removing all of unnecessary branches with reduction. However, in the proposed method, as described above, such candidates have already been removed at the time of enumeration. Secondly, our proposed structured Z-d-DNNF does not require (X, Y)-partition by definition. Thus, among the elements included in a certain decision dnode, more than two elements on the left side can be included in common. It causes the nodes of a structured Z-d-DNNF to take a redundant form compared with those of ZSDDs. Consequently, the node size of a structured Z-d-DNNF tends to be larger than those of ZSDDs.

Since structured Z-d-DNNF is a super set of ZSDD, the former can be succinct compared to the latter in general. However, owing to the implementation restrictions above, the node size is empirically larger in the former case.

5.8 Concluding Remarks

In this chapter, we have proposed a method to compile the *s*-*t* simple paths contained in a graph using structured Z-d-DNNF. The proposed method is an extension of frontier-based search designed to be based on branch decomposition. Although top-down construction is also based on branch decomposition and an extension of frontier-based search, our method considerably reduced the running time compared with top-down construction.

As future work, we will plan to expand the target of merging frontier-based search and attempt wider practical research. The previous frontier-based search including Simpath has already been applied to various graph-substructures and is widely applied to practical problems. Though top-down construction extremely reduced the execution time and index size than conventional frontier-based search, it requires complicated compilation method. On the other hand, our method is designed much simpler than top-down construction. Therefore, our method could be applied easier to other graph-substructures.

Although proposed structured Z-d-DNNF can be compiled faster than ZSDD, its structure capabilities are poorer than those of ZSDD. For example, our method does not support apply operations unlike ZSDD. Furthermore, the node size is actually larger than that of ZSDD. That is because, as shown in Section 5.7, a decision dnode in a structured Z-d-DNNF can contain more than two elements on the left side in common. Indeed, we attempted to construct the same structure as ZSDD by compressing these common elements. However, we could only find a way to spend a lot of time. According to our experimental result, the time to compress is larger than the time to construct *s*-*t* paths. Thus, efficient conversion method from structured Z-d-DNNF to ZSDD will be another future task.

Algorithm 5.3.3 Merge$(G, mate_{\langle d_{\beta^l}\rangle}, mate_{\langle d_{\beta^r}\rangle}, F_{\beta^l}, F_{\beta^r})$

Require: Graph G; the left and right mates $mate_{\langle d_{\beta^l}\rangle}$ and $mate_{\langle d_{\beta^r}\rangle}$ respectively; and the left and right frontier gnodes F_{β^l} and F_{β^r} respectively.

Ensure: If the pruning conditions are met, it returns $\perp$. Otherwise, it returns the mate m after the merging process.

for $u \in F_{\beta^l} \cap F_{\beta^r}$ **do**
  if $mate_{\langle d_{\beta^l}\rangle}[u] = u$ **then**
    // The degree of the left mate is zero.
    $mate_{\langle d_{\beta^l}\rangle}[u] \leftarrow mate_{\langle d_{\beta^r}\rangle}[u]$
  else if $mate_{\langle d_{\beta^r}\rangle}[u] = u$ **then**
    // The degree of the right mate is zero.
    $mate_{\langle d_{\beta^r}\rangle}[u] \leftarrow mate_{\langle d_{\beta^l}\rangle}[u]$
  else if $mate_{\langle d_{\beta^l}\rangle}[u] = 0$ or $mate_{\langle d_{\beta^r}\rangle}[u] = 0$ **then**
    // The sum of degrees of the left and right mates is greater than two. The pruning condition (2) holds.
    return $\perp$
  else
    // The degrees of both the left and right mates are one.
    if $u = s$ or $u = t$ **then**
      // Pruning condition (1) holds.
      return $\perp$
    else if $mate_{\langle d_{\beta^l}\rangle}[u] = mate_{\langle d_{\beta^r}\rangle}[u]$ **then**
      // Pruning condition (3) holds.
      return $\perp$
    else
      // Update the mate and memorize whether the opposite side is left or right.
      // Let $mate_{\langle d_{\beta^l}\rangle}[u] = u^l, mate_{\langle d_{\beta^r}\rangle}[u] = u^r$.
      $mate_{\langle d_{\beta^l}\rangle}[u] \leftarrow 0$; $mate_{\langle d_{\beta^r}\rangle}[u] \leftarrow 0$
      $mate_{\langle d_{\beta^l}\rangle}[u^l] \leftarrow u^r$; $mate_{\langle d_{\beta^r}\rangle}[u^r] \leftarrow u^l$
    end if
  end if
end for
for $u \in (F_{\beta^l} \cup F_{\beta^r} \cup \{s,t\})$ **do**
  // Update the mate.
  // $u = mate_{\langle d_{\beta^l}\rangle}[u]$ and/or $u = mate_{\langle d_{\beta^r}\rangle}[u]$ is satisfied in each gnode.
  if $mate_{\langle d_{\beta^l}\rangle}[u] \neq u$ **then**
    $mate_{\langle d_{\beta}\rangle}[u] \leftarrow mate_{\langle d_{\beta^l}\rangle}[u]$
  else
    $mate_{\langle d_{\beta}\rangle}[u] \leftarrow mate_{\langle d_{\beta^r}\rangle}[u]$
  end if
end for
return $mate_{\langle d_{\beta}\rangle}$

Table 5.1. Processing in each vnode.

Vnode id	Vnode type	Frontier Gnodes	Accepted $\langle d \rangle$s	Configurations	Discarded $\langle d \rangle$s	Pruning Conditions
0	leaf	$\{2,3\}$	$\{\emptyset\},\{\{B\}\}$	$\{m[2]=2,m[3]=3\},\{m[2]=3,m[3]=2\}$	none	none
1	leaf	$\{2,4\}$	$\{\emptyset\},\{\{C\}\}$	$\{m[2]=2,m[4]=4\},\{m[2]=4,m[4]=2\}$	none	none
2	leaf	$\{3,4\}$	$\{\emptyset\},\{\{D\}\}$	$\{m[3]=3,m[4]=4\},\{m[3]=4,m[4]=3\}$	none	none
3	internal	$\{2,3\}$	$\{\{C\}\},\{\{D\}\}$	$\{m[2]=4,m[3]=3\},\{m[2]=2,m[3]=4\}$	$\{\emptyset\},\{\{C,D\}\}$	$\{m[4]=4\},\{m[4]=0\}$
4	internal	$\{2\}$	$\{\{B,D\},\{C\}\}$	$\{m[2]=4\}$	$\{\{B,C\}\},\{\{D\}\}$	$\{m[3]=4\},\{m[3]=4\}$
5	leaf	$\{1,2\}$	$\{\emptyset\},\{\{A\}\}$	$\{m[1]=1,m[2]=2\},\{m[1]=2,m[2]=1\}$	none	none
6	internal	$\emptyset$	$\{\{A,B,D\},\{A,C\}\}$	none	$\{\{B,D\},\{C\}\}$	$\{m[1]=1\}$

Table 5.2. 1st Example of the merging of mates.
The input graph is Fig. 5.1 and the input vtree is Fig. 5.2.

	Vnode ID	Frontier Gnodes	A Set of Families	Configurations
Left	4	$\{2\}$	$\{\{B,D\},\{C\}\}$	$\{m_{d_4}[1]=1, m_{d_4}[2]=4, m_{d_4}[4]=2\}$
Right	5	$\{1,2\}$	$\{\{A\}\}$	$\{m_{d_5}[1]=2, m_{d_5}[2]=1, m_{d_5}[4]=4\}$
Merged	6	$\emptyset$	$\{\{A,B,D\},\{A,C\}\}$	$\{m_{d_6}[1]=4, m_{d_6}[2]=0, m_{d_6}[4]=1\}$

Table 5.3. 2nd Example of the merging of mates.
The input graph is Fig. 5.1 and the input vtree is Fig. 5.8.

	Vnode ID	Frontier Gnodes	A Set of Families	Configurations or Pruning Condition
Left	0	$\{2,4\}$	$\{\{C\}\}$	$\{m_{d_0}[1]=1, m_{d_0}[2]=4, m_{d_0}[4]=2\}$
Right	3	$\{2,4\}$	$\{\{B,D\}\}$	$\{m_{d_3}[1]=1, m_{d_3}[2]=4, m_{d_3}[4]=2\}$
Merged	4	$\{2\}$	none	$m_{d_0}[2]=m_{d_3}[2]=4$

Table 5.4. Experimental results of *s-t* simple paths compilation.

Instance	|V|	|E|	BW	num of paths	ZSDD				structured Z-d-DNNF				Vtree	Simpath
					Construction		Reduction		Construction		Reduction			
					time(ms)	nodes	time(ms)	nodes	time(ms)	nodes	time(ms)	nodes	time(ms)	time(ms)
grafo10638.100	100	141	10	9857246709	36801	12573212	1835	1387814	88010	4682754	2382	4620043	646	TO
grafo11227.100	100	142	10	10396775293	164193	61574987	6399	4863914	7801	1677616	758	1666372	648	TO
grafo10962.100	100	145	12	50186929684		TO			146441	18694077	12328	18229064	590	TO
grafo10469.100	100	148	13	51425390662		TO			454244	28882145	14597	28568578	730	TO
grafo10116.100	100	149	12	68950945569		TO			800248	11318708	3047	11205994	753	TO
grafo11335.100	100	153	12	7.08414E+11		TO			725132	27013341	5590	26978289	741	TO
att48	48	130	8	8.0841E+15	946	380008	62	118718	368	145741	30	142834	737	TO
berlin52	52	145	9	2.2578E+17	3166	1361494	229	450849	2604	695001	156	684363	767	TO
eil51	51	142	9	2.43471E+17	2420	914023	161	254678	5510	642953	107	637685	811	TO
eil76	76	215	11	1.48908E+19	224237	61749252	17983	23571302	125921	16850109	3924	16837970	1562	TO

Chapter 6

Conclusions and Future Work

In this chapter, we conclude this thesis and explain future work of our study.

6.1 Conclusions

In Chapter 3, we have presented our tractable indexing structure structured Z-d-DNNF. Unkike ZSDDs, structured Z-d-DNNF adopts determinism instead of partitioned-determinism. As a result, the capability of structured Z-d-DNNF is limited and supports less query and transformation than SDDs. However, it supports certain amount of applications after the compilation, such as model counting, model enumeration, and random sampling.

In addition, in Chapter 3, we have presented zero-suppressed knowledge compilation map. Though, BDDs have had more succinct structures as its extension, ZDDs lacked the extension except ZSDDs. We defined Z-NNF as a top level structure instead of NNF. Subsequently, we applied (structured) decomposability and (partitioned) determinism to the descendants of Z-NNF, and realized a tractable indexing structure system with zero-suppression,

In Chapter 4, we have proposed a method for compiling the independent sets of a given graph. In the proposed method, we have used structured Z-d-DNNF. Additionally, we have applied the previous method to compute the size of the maximum independent set with using tree decomposition.

As a result, the independent sets were computed with the time complexity equivalent to the previous method. In addition, after the compilation, we could efficiently do the random sampling or counting of the independent sets with dynamic programming on a structured Z-d-DNNF. The experimental results showed 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.

In Chapter 5, we have proposed a method to compile the *s-t* simple paths contained in a graph using structured Z-d-DNNF.The proposed method is an extension of frontier-based search designed to be based on branch decomposition. On the other hand, the preceding work, top-down construction of ZSDDs is also based on branch decomposition and an extension of frontier-based search. However, our method considerably reduced the running time compared with top-down construction.

6.2 Future Work

Our future work is as follows.

- To study the detailed capability of each structure in zero-suppressed knowledge compilation map: Darwiche clarified polytime queries and transformations supported by each structure in knowledge compilation map
  [Darwiche, 2001b, Darwiche and Marquis, 2002]. Further study of capability of the structures in zero-suppressed knowledge compilation map is inevitable to clarify the detail of the system and find the applications of the structures.

- To apply the preceding methods of counting or optimization to the compilation: To propose a new method of frontier-based search, the key issue is how to design *mates* that enables efficient compilation. Until today, we believe that it has been fully dependent on the "craftsmanship" of the designer. However, as shown in Chapter 4, the methods to solve the optimization problems with using tree decomposition have been already applied to varieties of counting or optimization problems of graph-substructure. Moreover, theoretical analysis of these methods are also provided by the preceding work. Among these applications, the method to calculate the size of maximum independent set is, we believe, one of the most simple one. Therefore, if we apply the other type of method than maximum independent set, we could extend frontier-based search more straightforwardly.

- To attempt wider practical research on the merging frontier-based search: In contrast to the compilation of independent sets, the compilation of *s-t* paths of our method is not dependent on the previous method of counting of optimization, but it is an extension of Simpath. Simpath has already been applied to various graph-substructures and is widely applied to practical problems. In order to realize more faster compilation, the use of ZSDDs or structured Z-d-DNNF become our option. As detailed in Chapter 2 and Chapter 5, ZSDDs support the same type of queries and transformation with ZDDs, whereas ZSDDs provides more efficient compilation method. However, as a trade off, the design of *mate* tends to be more

complicated than the conventional frontier-based search. On the other hand, structured Z-d-DNNF less queries and transformations than ZSDDs or ZDD, whereas the design of the compilation is more simple and consequently more faster than the compilation of ZSDDs. To study the further practical research, it would be important to identify the merits and demerits of these two options.

Acknowledgements

First of all, I would like to thank Professor Shin-ichi Minato. He was my supervisor Professor until he was transferred to Kyoto University. Since I entered Hokkaido University, he gave me a lot of suggestions and teachings on my research activity and on the articles I have written.

I also would like to thank Masaharu Yoshioka, the Professor at Knowledge-base Laboratory. He has supervised me since the previous supervisor Professor Minato transferred. He centrally advised me on important matters to take a doctoral degree. I also would like to thank Hiroki Arimura, the Professor at Information Knowledge Network Laboratory, and Takashi Horiyama, the Professor at Laboratory of Large-Scale Knowledge Processing, for their valuable comments and supports to write this thesis.

I also would like to thank Masaaki Nishino and Norihito Yasuda, the researchers of NTT Communication Science Laboratories. They gave me a lot of suggestions to write the articles which I have written while studying at a doctoral course. Especially, Dr. Nishino's articles on ZSDDs and their compilation methods gave me the essential intuition to create this thesis.

This research was supported by JSPS KAKENHI KIBAN (S) Discrete Structure Manipulation System Project under Grant Number 15H05711.

Finally, I would like to offer my special thanks to my mother, who supported my student days and always encouraged me.

Bibliography

[Amarilli et al., 2017] Amarilli, A., Bourhis, P., Jachiet, L., and Mengel, S. (2017). A circuit-based approach to efficient enumeration. In *ICALP*, volume 80 of *LIPIcs*, pages 111:1–111:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Arnborg et al., 1991] Arnborg, S., Lagergren, J., and Seese, D. (1991). Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340.

[Bagan, 2006a] Bagan, G. (2006a). MSO queries on tree decomposable structures are computable with linear delay. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, pages 167–181.

[Bagan, 2006b] Bagan, G. (2006b). Mso queries on tree decomposable structures are computable with linear delay. In *International Workshop on Computer Science Logic*, pages 167–181. Springer.

[Bodlaender et al., 2013] Bodlaender, H. L., Bonsma, P. S., and Lokshtanov, D. (2013). The fine details of fast dynamic programming over tree decompositions. In *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, pages 41–53.

[Bodlaender and Koster, 2008] Bodlaender, H. L. and Koster, A. M. C. A. (2008). Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269.

[Bordeaux et al., 2014] Bordeaux, L., Hamadi, Y., and Kohli, P., editors (2014). *Tractability Practical Approaches to Hard Problems*. Cambridge University Press.

[Bron and Kerbosch, 1973] Bron, C. and Kerbosch, J. (1973). Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577.

[Bryant, 1986a] Bryant, R. E. (1986a). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691.

[Bryant, 1986b] Bryant, R. E. (1986b). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

[Chiba and Nishizeki, 1985] Chiba, N. and Nishizeki, T. (1985). Arboricity and subgraph listing algorithms. *SICOMP*, 14(1):210–223.

[Chimani et al., 2012] Chimani, M., Mutzel, P., and Zey, B. (2012). Improved steiner tree algorithms for bounded treewidth. *J. Discrete Algorithms*, 16(Supplement C):67 – 78. Selected papers from the 22nd International Workshop on Combinatorial Algorithms (IWOCA 2011).

[Cook and Seymour, 2003] Cook, W. and Seymour, P. D. (2003). Tour merging via branch-decomposition. *INFORMS J. Comput.*, 15(3):233–248.

[Courcelle, 1990] Courcelle, B. (1990). The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75.

[Cygan et al., 2015] Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. (2015). *Parameterized Algorithms*. Springer.

[Darwiche, 1999] Darwiche, A. (1999). Compiling knowledge into decomposable negation normal form. In *IJCAI*, pages 284–289.

[Darwiche, 2001a] Darwiche, A. (2001a). Decomposable negation normal form. *J. ACM*, 48(4):608–647.

[Darwiche, 2001b] Darwiche, A. (2001b). On the tractable counting of theory models and its application to truth maintenance and belief revision. *JANCL*, 11(1-2):11–34.

[Darwiche, 2004] Darwiche, A. (2004). New advances in compiling CNF into decomposable negation normal form. In *ECAI*, pages 328–332.

[Darwiche, 2011a] Darwiche, A. (2011a). SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826.

[Darwiche, 2011b] Darwiche, A. (2011b). SDD: A new canonical representation of propositional knowledge bases. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826.

[Darwiche, 2014] Darwiche, A. (2014). Tractable knowledge representation formalisms. In *Tractability: Practical Approaches to Hard Problems*, pages 141–172. Cambridge University Press.

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

[Halin, 1976] Halin, R. (1976). S-functions for graphs. *Journal of geometry*, 8(1-2):171–186.

[Hamzaoglu and Patel, 2000] Hamzaoglu, I. and Patel, J. H. (2000). Test set compaction algorithms for combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):957–963.

[Hayase et al., 1995] Hayase, K., Sadakane, K., and Tani, S. (1995). Output-size sensitiveness of obdd construction through maximal independent set problem. In *International Computing and Combinatorics Conference*, pages 229–234. Springer.

[Hedetniemi and Laskar, 1985] Hedetniemi, S. and Laskar, R. (1985). Domination in trees: Models and algorithms. In *Graph Theory with Applications to Algorithms and Computer Science*, pages 423–442. John Wiley & Sons, Inc.

[Inoue et al., 2015] Inoue, T., Yasuda, N., Kawano, S., Takenobu, Y., Minato, S., and Hayashitakeru, Y. (2015). Distribution network verification for secure restoration by enumerating all critical failures. *IEEE Trans. Smart Grid*, 6(2):843–852.

[Inoue and Minato, 2016] Inoue, Y. and Minato, S. (2016). Acceleration of zdd construction for subgraph enumeration via path-width optimization. *Technical Report TCS-TR-A-16-80*.

[Ishihata et al., 2018] Ishihata, M., Maehara, T., and Rigaux, T. (2018). Algorithmic meta-theorems for monotone submodular maximization. *CoRR*, abs/1807.04575.

[Kawahara et al., 2017] Kawahara, J., Inoue, T., Iwashita, H., and Minato, S. (2017). Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Trans.*, 100-A(9):1773–1784.

[Kazana and Segoufin, 2011] Kazana, W. and Segoufin, L. (2011). First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2).

[Kazana and Segoufin, 2013] Kazana, W. and Segoufin, L. (2013). Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic*, 14(4):25:1–25:12.

[Kinnersley, 1992] Kinnersley, N. G. (1992). The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42(6):345–350.

[Kloks, 1994] Kloks, T. (1994). *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media.

[Knuth, 2011] Knuth, D. E. (2011). *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional.

[Luce and Perry, 1949] Luce, R. D. and Perry, A. D. (1949). A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116.

[Maehara et al., 2017] Maehara, T., Suzuki, H., and Ishihata, M. (2017). Exact computation of influence spread by binary decision diagrams. In *WWW*, pages 947–956.

[Makino and Uno, 2004] Makino, K. and Uno, T. (2004). New algorithms for enumerating all maximal cliques. In *SWAT*, volume 3111, pages 260–272. Springer.

[Minato, 1993] Minato, S. (1993). Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC*, pages 272–277.

[Mitchell and Hedetniemi, 1979] Mitchell, S. and Hedetniemi, S. (1979). Linear algorithms for edge-coloring trees and unicyclic graphs. *Information Processing Letters*, 9(3):110 – 112.

[Nishino et al., 2016] Nishino, M., Yasuda, N., Minato, S., and Nagata, M. (2016). Zero-suppressed sentential decision diagrams. In *AAAI*, pages 1058–1066.

[Nishino et al., 2017] Nishino, M., Yasuda, N., Minato, S., and Nagata, M. (2017). Compiling graph substructures into sentential decision diagrams. In *AAAI*, pages 1213–1221.

[Oztok and Darwiche, 2015] Oztok, U. and Darwiche, A. (2015). A top-down compiler for sentential decision diagrams. In *IJCAI*, pages 3141–3148.

[Pipatsrisawat and Darwiche, 2008] Pipatsrisawat, K. and Darwiche, A. (2008). New compilation languages based on structured decomposability. In *AAAI*, pages 517–522.

[Pipatsrisawat and Darwiche, 2010] Pipatsrisawat, K. and Darwiche, A. (2010). Top-down algorithms for constructing structured DNNF: theoretical and practical implications. In *ECAI*, pages 3–8.

[Rhodes et al., 2003] Rhodes, N., Willett, P., Calvet, A., Dunbar, J. B., and Humblet, C. (2003). Clip: similarity searching of 3d databases using clique detection. *J. Chem. Inf. Comput. Sci.*, 43(2):443–448.

[Robertson and Seymour, 1984] Robertson, N. and Seymour, P. D. (1984). Graph minors. III. planar tree-width. *Journal of Combinatorial Theory, Ser. B*, 36(1):49–64.

[Robertson and Seymour, 1991] Robertson, N. and Seymour, P. D. (1991). Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory. Ser. B*, 52(2):153–190.

[Samudrala and Moult, 1998] Samudrala, R. and Moult, J. (1998). A graph-theoretic algorithm for comparative modeling of protein structure. *J. Mol. Biol.*, 279(1):287–302.

[Seese, 1996] Seese, D. (1996). Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6(6):505–526.

[Sekine et al., 1995] Sekine, K., Imai, H., and Tani, S. (1995). Computing the tutte polynomial of a graph of moderate size. In *ISAAC*, pages 224–233.

[Sugaya et al., 2017] Sugaya, T., Nishino, M., Yasuda, N., and Minato, S. (2017). Fast compilation of s-t paths on a graph for counting and enumeration. In *Proceedings of The 3rd International Workshop on Advanced Methodologies for Bayesian Networks*, volume 73, pages 129–140. PMLR.

[Sugaya et al., 2018] Sugaya, T., Nishino, M., Yasuda, N., and Minato, S. (2018). Fast compilation of graph substructures for counting and enumeration. *Behaviormetrika*, pages 1–28.

[Sugaya et al., 2020] Sugaya, T., Nishino, M., Yasuda, N., and Minato, S. (2020). Tree decomposition-based approach for compiling independent sets. *Journal of Information Processing*, 28:354–368.

[Tomita et al., 2006] Tomita, E., Tanaka, A., and Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42.

[Tsukiyama et al., 1977] Tsukiyama, S., Ide, M., Ariyoshi, H., and Shirakawa, I. (1977). A new algorithm for generating all the maximal independent sets. *SICOMP*, 6(3):505–517.

[Uno, 2003] Uno, T. (2003). An output linear time algorithm for enumerating chordless cycles (in japanese). *92th SIGAL of Information Processing Society Japan*, pages 47–53.

[Valiant, 1979] Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *SIAM J. Comput*, 8(3):410–421.