Doctoral Thesis

Studies on Implicit Graph Enumeration Using Decision Diagrams

Yu Nakahata

Graduate School of Informatics
Kyoto University

September 2021

Abstract

Graphs are ubiquitous objects in the real world. Especially, enumerating subgraphs of a given graph is a fundamental task in computer science. Since the number of subgraphs can be exponentially larger than the input graph size, it is not practical to list all subgraphs one by one. To overcome the difficulty, we focus on *implicit enumeration* algorithms. Such an algorithm constructs a *decision diagram* (DD) representing the set of subgraphs instead of explicitly enumerating the subgraphs. This thesis is devoted to designing some implicit enumeration algorithms. We theoretically estimate their complexity and experimentally confirm their efficiency. In this thesis, we mainly use zero-suppressed binary decision diagrams (ZDDs) as DDs.

First, we focus on the evacuation planning problem. For this problem, the existing method was limited to grid graphs. We generalize the definition of convexity of regions and propose an algorithm to enumerate partitioning patterns into such regions for general graphs. The efficiency of the proposed algorithm is confirmed by the experiments using real-world map data.

Second, we move on to the balanced graph partitioning problem. We propose an algorithm to enumerate all the graph partitions such that all the weights of the connected components are at least a specified value. Our algorithm uses not only ZDDs but also ternary decision diagrams (TDDs) and realizes an operation, which seems difficult to be designed only by ZDDs. Experimental results show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.

Next, we try to extend the types of subgraphs that can be enumerated by ZDDs. We focus on the forbidden minor characterization of graphs and propose a method to enumerate subgraphs having such characterization. Such graphs include planar, outerplanar, series-parallel, and cactus graphs. Experimental results show that our algorithm can find all planar subgraphs in a given graph up to five orders of magnitude faster than a naive backtracking-

based method.

Finally, we deal with another decision diagram than ZDDs, Zero-suppressed Sentential Decision Diagrams (ZSDDs). ZSDDs are generalizations of ZDDs and can be substantially smaller than ZDDs when representing the same family set. However, efficient algorithms to construct ZSDDs were known only for specific types of subgraphs: matchings and paths. We propose a novel framework to construct ZSDDs, which enables us to deal with several types of subgraphs such as matchings, paths, cycles, and spanning trees. We show that the sizes of constructed ZSDDs are bounded by the branch-width of the input graph. Experiments show that proposed methods can construct ZSDDs faster than ZDDs and that the constructed ZSDDs are smaller than ZDDs representing the same sets of subgraphs.

Acknowledgements

First of all, I would like to express my greatest appreciation to my supervisor, Shin-ichi Minato. He always gives me insightful comments, which are essential to improve my research. I am also deeply grateful to Jun Kawahara for his help since my master's course. I sincerely thank them and Akihiro Yamamoto for reading this thesis and giving constructive comments.

I would like to thank all current and former members of Minato laboratory. Especially, I am grateful to David Avis, François Le Gall, Yuni Iwamasa, Ryosuke Matsuo, Shinsaku Sakaue, and Suguru Tamaki for their valuable comments and exciting discussion. I also thank the secretaries of our laboratory, Shinako Fukumura, Seiko Jinno, and Sachie Oomae.

I would like to sincerely thank my coauthors: Shuhei Denzumi, Takashi Horiyama, Masakazu Ishihata, Shoji Kasahara, Yasuaki Kobayashi, Kazuhiro Kurita, Masaaki Nishino, Kengo Ohsawa, Hirofumi Suzuki, Kunihiro Wasa, Katsuhisa Yamanaka, and Kazuaki Yamazaki.

I appreciate the financial support by JSPS Research Fellowship for Young Scientists. I am also grateful to JSPS KAKENHI(S) Discrete Structure Manipulation System Project and MEXT Algorithmic Foundations for Social Advancement Project for fostering communications of researchers.

Finally, I would like to thank my family and friends for their kind support.

List of Publications

Publications Included in This Thesis

This thesis includes contents of the following four publications.

Refereed Journal Articles

1. Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shoji Kasahara.
   Enumerating All Spanning Shortest Path Forests with Distance and Capacity Constraints.
   IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 101-A(9):1363–1374, 2018.

Refereed Conference Proceedings

1. Yu Nakahata, Jun Kawahara, and Shoji Kasahara.
   Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams.
   In *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA 2018)*, pp. 21:1–21:13, 2018.

2. Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shin-ichi Minato.
   Implicit Enumeration of Topological-minor-embeddings and Its Application to Planar Subgraph Enumeration.
   In *Proceedings of the 14th International Conference and Workshops on Algorithms and Computing (WALCOM 2020)*, pp. 211–222, 2020.

3. Yu Nakahata, Masaaki Nishino, Jun Kawahara, and Shin-ichi Minato.
   Enumerating All Subgraphs Under Given Constraints Using
   Zero-suppressed Sentential Decision Diagrams.

In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA 2020)*, pp. 9:1–9:14, 2020.

Unrefereed Preprints

The author also addressed the following works, which are currently unrefereed preprints and not included in this thesis.

1. Takashi Horiyama, Jun Kawahara, Shin-ichi Minato, and Yu Nakahata.
   Decomposing a Graph into Unigraphs.
   arXiv preprints, arXiv:1904.09438, 2019.

2. Yu Nakahata.
   On the Clique-width of Unigraphs.
   arXiv preprints, arXiv:1905.12461, 2019.

3. Yasuaki Kobayashi and Yu Nakahata.
   A Note on Exponential-time Algorithms for Linearwidth.
   arXiv preprints, arXiv:2010.02388, 2020.

4. Yu Nakahata, Takashi Horiyama, Shin-ichi Minato, and Katsuhisa Yamanaka.
   Compiling Crossing-free Geometric Graphs with Connectivity Constraint for Fast Enumeration, Random Sampling, and Optimization.
   arXiv preprints, arxiv:2001.08899, 2020.

Contents

Chapter 1

Introduction

1.1 Background

Graphs are widely used to model real-world objects such as communication networks, distribution networks, and road networks. When dealing with graphs, enumerating subgraphs of a given graph under some constraint is a fundamental task. There are enumeration algorithms for several types of subgraphs such as cliques [1], paths [2], and spanning trees [3]. These algorithms list all subgraphs one by one in a small amount of time per subgraph. However, such algorithms take at least linear time and space to the number of subgraphs. Since the number of subgraphs can be exponentially larger than the size of the input graph, it is trouble when applied to practical problems.

To overcome the difficulty, we focus on *implicit enumeration* algorithms [4, 5, 6]. Such an algorithm constructs a *decision diagram* (DD) [7, 8] representing the set of subgraphs instead of explicitly enumerating the subgraphs. In this thesis, we consider the edge-induced subgraphs, which means each subgraph can be identified by a subset of the graph edges. DDs are known as efficient data structures for representing set families. We use a DD to represent a set of subgraphs, each of which is a subset of the edges. The efficiency of an implicit enumeration algorithm does not directly depend on the number of subgraphs but rather on the size of the output DD [4]. The size of a DD can be much (exponentially in some cases) smaller than the number of subgraphs, and thus, in such cases, we can expect that the implicit algorithms will work much faster than explicit ones. Using DDs, we can perform several useful queries on the set of subgraphs. For example, we can count the number of subgraphs, randomly sample a subgraph, find an optimal subgraph with respect to a linear function [5].

1.2 Related Work

Enumeration. Enumeration algorithms have been studied for several types of subgraphs such as paths [2, 9, 10], cycles [10, 11, 12] spanning trees [3, 10, 13], matchings [14, 15, 16, 17], and cliques [1, 18, 19]. There are general methods to design enumeration algorithms such as binary partition [17], gray code [20], and reverse search [21]. These algorithms explicitly enumerate solutions one by one. As a result, they need at least proportional time and memory to the number of solutions, which can be exponentially larger than the input size. Therefore, in this thesis, we focus on implicit enumeration using decision diagrams (DDs).

Decision diagrams (DDs). Binary decision diagrams (BDDs) were introduced by Lee [22] and Akers [23]. Later, Bryant [7] found that reduced and ordered BDDs (ROBDDs) have a canonical representation. Using this property, he proposed Apply operations, which enables the synthesis of BDDs. His paper leads to wide applications of BDDs such as logic synthesis [24, 25, 26], model checking [27], and logic optimization [28]. There are several techniques to implement BDDs efficiently, for example, variable ordering [29, 30, 31, 32], hash table [33], attributed edges [34], and shared-BDD [34].

ZDDs were proposed by Minato [8] as a variant of BDDs. ZDDs tend to be smaller than BDDs when representing sparse set families. By this property, ZDDs have been applied to wide areas such as data mining [35, 36, 37], game theory [38], graph optimization [39], and combinatorial optimization [40, 41, 42]. BDDs and ZDDs are well surveyed in Knuth's book [5].

There are several variations of BDDs/ZDDs: Sequence BDDs (SeqBDDs) [43] for sets of sequences, πDDs [44], rot-πDDs [45], and Group Decision Diagrams (GDDs) [46] for sets of permutations, and multi-valued decision diagrams (MDDs) [47] for multi-valued logic functions. For logic functions or set families, there are variants of BDDs/ZDDs. Sentential Decision Diagrams (SDDs) [48] are generalizations of BDDs. Zero-suppressed SDDs (ZSDDs) [49] are generalizations of ZDDs and the zero-suppressed variant of SDDs. There is a trade-off between succinctness and types of queries supported by DDs. Darwiche and Marquis studied this trade-off as a knowledge compilation map [50].

DDs for graph problems. Sekine et al. [6] proposed an algorithm to compute the Tutte polynomial of a graph using BDDs. This algorithm essentially constructs a BDD representing all the spanning trees of the input graph. Knuth [5] proposed an algorithm to construct a ZDD representing all the paths in a given graph. These algorithms are generalized as frontier-based search (FBS) by Kawahara et al [4]. The framework has been applied for several problems. A prominent application is network reliability evaluation [51, 52, 53, 54, 55], which is known to be #P-hard [56]. Other application consists of NP-hard problems such as distribution loss minimization [57], influence maximization [58], evacuation planning [59], political redistricting [60], longest one-way ticket problem [61], and link puzzles [62]. There are libraries such as Graphillion [63] and TdZdd. The complexity of algorithms based on FBS is measured by the path-width of the input graph [64].

1.3 Our contribution

In this thesis, we propose implicit enumeration algorithms for the following problems:

1. Evacuation planning for general graphs
2. Balanced graph partition
3. Planar subgraph enumeration
4. FBS for zero-suppressed sentential decision diagrams (ZSDDs)

We summarize each contribution in the below:

1. Evacuation planning for general graphs: In this problem, we are given a graph representing the road network of the target area. Every vertex has a population near the vertex and some vertices are marked as shelters and they have capacities. Our task is to partition a graph into several regions so that each region contains exactly one shelter. There are several constraints to this problem. Each region must be convex to reduce intersections of evacuation routes, the distance between each point to a shelter must be bounded so that inhabitants can quickly

evacuate from a disaster, and the number of inhabitants assigned to each shelter must not exceed the capacity of the shelter. We formulate the convexity of connected components as a *spanning shortest path forest* for general graphs and propose a novel algorithm to tackle this multi-objective optimization problem. The algorithm not only obtains a single partition but also enumerates all partitions simultaneously satisfying the above complex constraints, which is difficult to be treated by existing algorithms, using ZDDs as a compressed representation. The efficiency of the proposed algorithm is confirmed by the experiments using real-world map data. The results of the experiments show that the proposed algorithm can obtain hundreds of millions of partitions satisfying all the constraints for input graphs with a hundred edges in a few minutes.

2. Balanced graph partition: Partitioning a graph into balanced components is important for several applications. For multi-objective problems, it is useful not only to find one solution but also to enumerate all the solutions with good values of objectives. We propose an algorithm to enumerate all the graph partitions such that all the weights of the connected components are at least a specified value. Our algorithm utilizes not only ZDDs but also ternary decision diagrams (TDDs) and realizes an operation, which seems difficult to be designed only by ZDDs. Experimental results show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.

3. Planar subgraph enumeration: Given graphs G and H, we propose a method to implicitly enumerate topological-minor-embeddings of H in G using decision diagrams. We show a useful application of our method to enumerating subgraphs characterized by forbidden topological minors, including planar, outerplanar, series-parallel, and cactus subgraphs. Computational experiments show that our method can find all planar subgraphs in a given graph up to five orders of magnitude faster than a naive backtracking-based method. We apply our method also for outerplanar, series-parallel, and cactus subgraphs.

4. FBS for ZSDDs: ZSDDs [49] are recently proposed DD as generalizations of ZDDs. ZSDDs can be smaller than ZDDs when representing the same set of subgraphs [65]. In addition, like ZDDs, ZSDDs support

several poly-time queries such as counting, random sampling, and Apply operations [49]. However, efficient algorithms to construct ZSDDs are known only for specific types of subgraphs: matchings and paths. In the chapter, we propose a novel framework of top-down construction algorithms for ZSDDs. To design a top-down construction algorithm using our framework, one only has to prove a recursive formula for the desired set of subgraphs. Using the recursive formula, we can theoretically show the correctness and the complexity of the algorithm, which was difficult with the existing method. We apply our framework to the three fundamental constraints used in ZDDs: the number of edges, degrees of vertices, and connectivity of vertices. We show that the sizes of constructed ZSDDs are bounded by the branch-width of the input graph. Experiments show that proposed methods can construct ZSDDs faster than ZDDs and that the constructed ZSDDs are smaller than ZDDs representing the same sets of subgraphs.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 presents preliminaries commonly used in this thesis. Chapter 3 develops a ZDD-based algorithm for evacuation planning problem. In Chapter 4, we propose an efficient algorithm for implicit enumeration of balanced graph partitions. We propose implicit enumeration algorithms for planar and related subgraphs in Chapter 5. In Chapter 6, we propose implicit enumeration algorithms using ZSDDs. Finally, we conclude this thesis in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we give preliminaries commonly used in the thesis. We introduce notations in Section 2.1. In Section 2.2 and Section 2.3, we explain a zero-suppressed binary decision diagram (ZDD) and frontier-based search, respectively.

2.1 Notations

Let $\mathbb{Z}$ be the sets of integers. $\mathbb{Z}^+$ and $\mathbb{N}$ denote the set of positive and non-negative integers, respectively. For $k \in \mathbb{Z}^+$, we define $[k] = \{1, \dots, k\}$. $\mathbb{R}$ denotes the set of real numbers and we use $\mathbb{R}^+$ to represent the set of positive real numbers.

Let $G = (V, E)$ be an undirected graph where V is the vertex set and E is the edge set. $|V|$ and $|E|$ denote the number of vertices and edges, respectively. For vertex subset $U \subseteq V$, the *vertex-induced subgraph* $G[U]$ is the subgraph $(U, E[U])$, where $E[U]$ is the set of edges whose endpoints are both in U. For edge subset $S \subseteq E$, the *edge-induced subgraph* $G[S]$ is the subgraph $(V[S], S)$, where $V[S] \subseteq V$ is the set of vertices to which an edge in S is incident. We often identify U with $G[U]$ and S with $G[S]$. For $S \subseteq E$ and $u \in V$, the *degree* $\deg_S(u)$ of u in S is the number of edges incident to u in S. Graphs G and H are *isomorphic* if there exists a bijection $\psi\colon V(G) \to V(H)$ such that, for all $u, v \in V(G)$, $\{u, v\} \in E(G) \Leftrightarrow \{\psi(u), \psi(v)\} \in E(H)$.

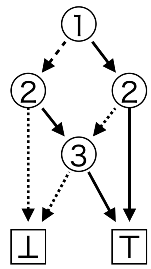


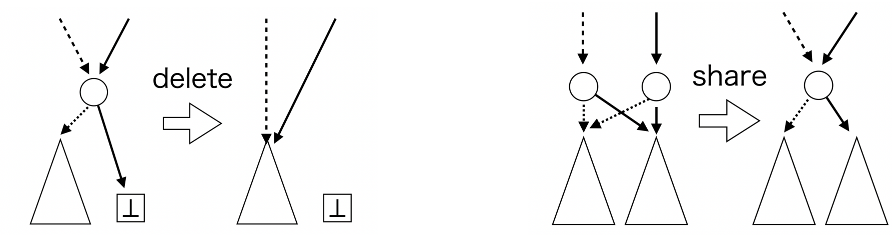
Figure 2.1: The ZDD representing the family $\{\{1,3\},\{2,3\},\{3\}\}$. A square represents a terminal node. A circle is a non-terminal node and the number in it is a label. A solid arc is a 1-arc and a dashed arc is a 0-arc.

2.2 Zero-suppressed binary decision diagram

A *zero-suppressed binary decision diagram* (ZDD) [8] is a directed acyclic graph $Z = (N_Z, A_Z)$ representing a family of sets. Here N_Z is the set of *nodes* and A_Z is the set of *arcs* (directed edges).[1] For an arc $(\alpha, \beta) \in A_Z$, we call α *head* and β *tail*. N_Z contains two *terminal nodes* $\top$ and $\bot$. The other nodes than the terminal nodes are called *non-terminal nodes*. Each non-terminal node α has the *0-arc*, the *1-arc*, and the *label* corresponding to an item in the universe set. For $x \in \{0,1\}$, we call the tail of the x-arc of a non-terminal node α the x*-child* of α, denoted by α_x. We denote the label of α by $l(\alpha)$ and assume that $l(\alpha) \in \mathbb{Z}^+ \cup \{\infty\}$ for any $\alpha \in N_Z$. For convenience, we let $l(\top) = l(\bot) = \infty$. For each arc $(\alpha, \beta) \in A_Z$, the inequality $l(\alpha) < l(\beta)$ holds, which ensures that Z is acyclic. There is exactly one node whose in-degree is zero, called the *root node* and denoted by r_Z. The number of the non-terminal nodes of Z is called the *size* of Z and denoted by $|Z|$.

A ZDD Z represents the family of sets in the following way. Let $\mathcal{P}_Z$ be the set of all the directed paths from r_Z to $\top$. For a directed path $p = (n_1, a_1, \ldots, n_k, a_k, \top) \in \mathcal{P}_Z$ with $n_i \in N_Z$, $a_i \in A_Z$, and $n_1 = r_Z$, we define $S_p = \{l(n_i) \mid a_i \in A_{Z,1}, i \in [k]\}$, where $A_{Z,1}$ is the set of the 1-arcs of Z. We interpret that Z represents the family $\{S_p \mid p \in \mathcal{P}_Z\}$. In other words, a directed path from r_Z to $\top$ corresponds to a set in the family represented by Z. For example, Fig. 2.1 shows a ZDD representing the set family $\{\{1,2\},\{1,3\},\{2,3\}\}$. In the figure, a dashed arc ($\dashrightarrow$) and a solid

[1] To avoid confusion, we use the words “vertex” and “edge” for input graphs and “nodes” and “arcs” for decision diagrams.



(a) Node deletion.

(b) Node sharing.

Figure 2.2: Reduction rules for the ZDD.

arc ($\rightarrow$) are a 0-arc and a 1-arc, respectively. On the ZDD in Fig. 2.1, there are three directed paths from the root node to $\top$: $1 \rightarrow 2 \rightarrow \top, 1 \rightarrow 2 \dashrightarrow 3 \rightarrow \top$, and $1 \dashrightarrow 2 \rightarrow 3 \rightarrow \top$, which correspond to $\{1,2\}, \{1,3\}$, and $\{2,3\}$, respectively.

In general, there are multiple ZDDs representing the same set family. To reduce the size of ZDDs, we apply the following two reduction rules:

- Node deletion: Delete a node α if $\alpha_1 = \bot$ and, for all arcs whose tail is α, replace it by α_0. (Fig. 2.2(a))

- Node sharing: Merge two nodes α and β if $\ell(\alpha) = \ell(\beta)$, $\alpha_0 = \beta_0$, and $\alpha_1 = \beta_1$. (Fig. 2.2(b))

A ZDD is called *reduced* if we can no longer apply reduction rules to the ZDD. The reduced ZDD has a canonical and minimum form [8]. The ZDD in Fig. 2.1 is reduced. We denote the reduced ZDD representing a family $\mathcal{F}$ by $Z_{\mathcal{F}}$. The size of the reduced ZDD depends on the variable ordering, i.e., the order of labels. Finding an optimal variable ordering is NP-complete [66].

ZDDs support several useful queries about set families. For example, we can count the number of sets in the family, randomly sample a set, optimize a linear function, in $\mathcal{O}(|Z|)$ time [5]. In addition, there are binary operations between ZDDs. Given two ZDDs $Z_{\mathcal{F}}$ and $Z_{\mathcal{G}}$ respectively representing set families $\mathcal{F}$ and $\mathcal{G}$, we can construct $Z_{\mathcal{F}\cup\mathcal{G}}$, $Z_{\mathcal{F}\cap\mathcal{G}}$, and $Z_{\mathcal{F}\setminus\mathcal{G}}$, in $\mathcal{O}(|Z_{\mathcal{F}}|\,|Z_{\mathcal{G}}|)$ time [67]. There are more involved operations. For set families $\mathcal{F}$ and $\mathcal{G}$, the *restriction* of $\mathcal{F}$ by $\mathcal{G}$ is defined as $\mathcal{F} \rhd \mathcal{G} = \{X \mid X \in \mathcal{F}, \exists Y \in \mathcal{G}, X \supseteq Y\}$. Similarly, the *permission* of $\mathcal{F}$ by $\mathcal{G}$ is defined as $\mathcal{F} \lhd \mathcal{G} = \{X \mid X \in \mathcal{F}, \exists Y \in \mathcal{G}, X \subseteq Y\}$. Given two ZDDs $Z_{\mathcal{F}}$ and $Z_{\mathcal{G}}$, there are algorithms to construct $\mathcal{Z}_{\mathcal{F}\rhd\mathcal{G}}$ and $\mathcal{Z}_{\mathcal{F}\lhd\mathcal{G}}$ [5]. However, these algorithms do not have

Table 2.1: Binary operations between set families

Name	Operator	Formula	Result
union	$\cup$	$\mathcal{F} \cup \mathcal{G}$	$\{X \mid X \in \mathcal{F} \text{ or } X \in \mathcal{G}\}$
intersection	$\cap$	$\mathcal{F} \cap \mathcal{G}$	$\{X \mid X \in \mathcal{F} \text{ and } X \in \mathcal{G}\}$
difference	$\setminus$	$\mathcal{F} \setminus \mathcal{G}$	$\{X \mid X \in \mathcal{F}, X \notin \mathcal{G}\}$
restriction	$\triangleright$	$\mathcal{F} \triangleright \mathcal{G}$	$\{X \mid X \in \mathcal{F}, \exists Y \in \mathcal{G}, X \supseteq Y\}$
permission	$\triangleleft$	$\mathcal{F} \triangleleft \mathcal{G}$	$\{X \mid X \in \mathcal{F}, \exists Y \in \mathcal{G}, X \subseteq Y\}$

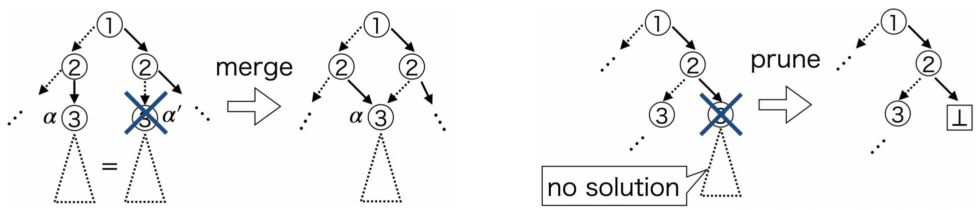
polynomial-time guarantee. For $\cup, \cap$ and $\setminus$, we can use efficient recursive algorithms called Apply operation [7]. In contrast, algorithms for $\triangleright$ and $\triangleleft$ are doubly recursive (for example, the recursion of $\triangleright$ calls the recursion of $\cup$ inside), which makes theoretical analysis difficult. Table 2.1 shows the list of binary operations between set families that are supported by ZDDs and we use in this thesis. We refer [5] for other binary operations between ZDDs and the details of algorithms of binary operations.

2.3 Frontier-based search

Frontier-based search [4, 5, 6] (FBS) is a framework of algorithms that efficiently construct a decision diagram representing the set of subgraphs satisfying given constraints of an input graph. We explain the general framework of FBS. Given a graph $G = (V, E)$, let $\mathcal{M}$ be a class of subgraphs we would like to enumerate (for example, $\mathcal{M}$ is the set of all the s-t paths on G). Frontier-based search constructs the ZDD representing the family $\mathcal{M}$ of subgraphs. By fixing G, a subgraph is identified with the edge set the subgraph has, and thus the ZDD represents the family of edge sets actually. Non-terminal nodes of ZDDs constructed by frontier-based search have labels $e_1, \ldots, e_m$. We identify e_i with the integer i. We assume that it is determined in advance which edge in G has which index i of e_i.

We directly construct the ZDD in a breadth-first manner. We first create the root node of the ZDD, make it have label e_1, and then we carry out the following procedure for $i = 1, \ldots, m$. For each node n_i with label e_i, we create two nodes, each of which is either a terminal node or a non-terminal node whose label is e_{i+1} (if $i = m$, the candidate is only a terminal node), as the 0-child and the 1-child of n_i.

Which node the x-arc of a node n_i with label e_i points at is determined by



(a) Node merging.

(b) Pruning.

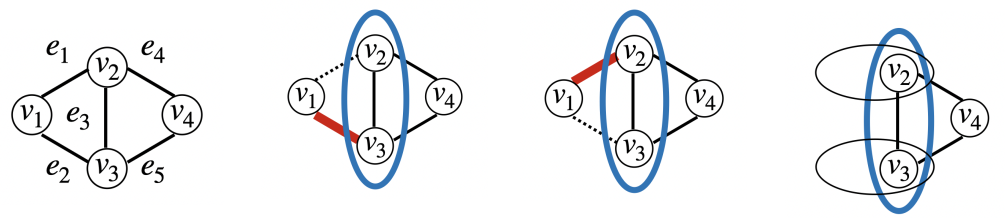
Figure 2.3: Procedures of FBS.

a function, called MakeNewNode, of which we design the detail according to $\mathcal{M}$, i.e., what subgraphs we want to enumerate. Here we describe the generalized nature that MakeNewNode must possess. The node n_i represents the set of the subgraphs, denoted by $\mathcal{G}(n_i)$, corresponding to the set of the directed paths from the root node to n_i. Each subgraph in $\mathcal{G}(n_i)$ contains only edges in $\{e_1, \ldots, e_{i-1}\}$. Note that $\mathcal{G}(\top)$ is the desired set of subgraphs represented by the ZDD after the construction finishes. To decide which node the x-arc of n_i points at without traversing the ZDD (under construction), we make each node n_i have the information $n_i.\texttt{conf}$ (called *configuration*), which is shared by all the subgraphs in $\mathcal{G}(n_i)$. The content of $n_i.\texttt{conf}$ also depends on $\mathcal{M}$ (for example, in the case of s-t paths, we store degrees and components of the subgraphs in $\mathcal{G}(n_i)$ into $n_i.\texttt{conf}$). MakeNewNode creates a new node, say n_{new}, with label e_{i+1} and must behave in the following manner.

1. For all edge sets $S \in \mathcal{G}(n_{\text{new}})$, if there is no edge set $S' \subseteq \{e_{i+1}, \ldots, e_m\}$ such that $S \cup S' \in \mathcal{M}$, the function discards n_{new} and returns $\perp$ to avoid redundant expansion of nodes. (*pruning*) In other words, if any subgraph represented by n_{new} cannot be extended to a solution, we no longer expand n_{new}. (Fig. 2.3(a))

2. Otherwise, if $i = m$, the function returns $\top$, which indicates the subgraphs represented by n_m are in solutions.

3. Otherwise, the function calculates $n_{\text{new}}.\texttt{conf}$ from $n_i.\texttt{conf}$. If there is a node n_{i+1} such that whose label is e_{i+1} and $n_{\text{new}}.\texttt{conf} = n_{i+1}.\texttt{conf}$, the function abandons n_{new} and returns n_{i+1}. (*node merging*) This is needed to merge nodes corresponding to the same state and avoid



(a) Input graph. (b) Intermediate solution 1. (c) Intermediate solution 2. (d) Configuration.

Figure 2.4: Intermediate solutions with the same configuration for spanning forests. Bold, dashed, and solid edges indicate adopted, unadopted, and unprocessed edges, respectively. The vertices inside the ellipses are the frontier.

constructing redundant nodes. If there is no node with the same state, the function returns n_{new}. (Fig. 2.3(b))

We make the x-arc of n_i point at the node returned by MakeNewNode.

As for n_i.`conf`, in the case of several kinds of subgraphs such as paths and cycles, it is known that we only have to store states relating to the vertices to which both an edge in $\{e_1, \ldots, e_{i-1}\}$ and an edge in $\{e_i, \ldots, e_m\}$ are incident into each node [5] (in the case of s-t paths, we store degrees and components of such vertices into each node). The set of the vertices is called the *frontier*. More precisely, the *i-th frontier* is defined as $F_i = (\bigcup_{j=1}^{i-1}\{\{u, v\} \mid e_j = \{u, v\}\}) \cap (\bigcup_{k=i}^{m}\{\{u, v\} \mid e_k = \{u, v\}\})$. Since we have assumed that the edge ordering is determined in advance, the i-th frontier is uniquely determined for every i. For convenience, we define $F_0 = F_m = \emptyset$. States of vertices in F_{i-1} are stored into n_i.`conf`. By limiting the domain of the information to the frontier, we can reduce memory consumption and share more nodes, which leads to a more efficient algorithm.

For example, when we want to enumerate spanning forests, i.e., subgraphs with no cycles, we have to maintain the connectivity of the vertices in the frontier as configuration. We consider the input graph in Fig. 2.4(a). Now we processed e_1 and e_2 and there are two intermediate solutions. The two intermediate solutions are different as edge subsets. However, they are equivalent in the sense that they will be spanning forests unless we adopt all edges from e_3, e_4, and e_5. Thus, the ZDD nodes corresponding to these intermediate solutions can be merged. This can be detected by the configuration in Fig. 2.4(d). The current frontier is $\{v_2, v_3\}$ and there are two connected components containing v_2 and v_3.

The efficiency of an algorithm based on FBS is often evaluated by the *width of a ZDD* constructed by the algorithm. The width W_Z of a ZDD Z is defined as $W_Z = \max\{|\mathcal{N}_i| \mid i \in [m]\}$, where $\mathcal{N}_i$ denotes the set of nodes whose labels are e_i. Using W_Z, the number of nodes in Z can be written as $|Z| = \mathcal{O}(mW_Z)$ and the time complexity of the algorithm is $\mathcal{O}(\tau|Z|)$, where τ denotes the time complexity of MakeNewNode for one node.

A ZDD constructed by FBS may not be reduced. To obtain the reduced ZDD, we have to apply reduction rules [5].

Chapter 3

Evacuation Planning for General Graphs

3.1 Introduction

In this chapter, we consider the following variant of the graph partitioning problem, called the evacuation planning problem: We are given a graph $G = (V, E)$ representing an area and a set $S \subseteq V$ of shelters (or evacuation centers). Each vertex has an integer value representing the population and each shelter has an integer value, called *shelter-capacity*, that means the number of evacuees that the shelter can accommodate. The goal is to find a partition of G such that each connected component contains exactly one shelter in S. There are several constraints we must consider in the problem: the structural, distance and shelter-capacity constraints. The *structural constraint* requires that each component is *convex* to reduce intersections of evacuation routes. The *distance constraint* is that the distances from vertices to the assigned shelters should be short. In addition, for fairness, it is not preferable that evacuees are assigned to a far shelter even though another shelter exists near them. The *shelter-capacity constraint* is about the capacities of shelters: the number of evacuees assigned to each shelter should not exceed its shelter-capacity. In practice, it is often that the total shelter-capacity of shelters is insufficient to accommodate all inhabitants in an area. Thus, although we allow a shelter to accommodate evacuees more than its shelter-capacity, we want to reduce the ratio of the number of evacuees assigned to a shelter to its shelter-capacity. This multi-objective property makes it difficult to define what is the best partition. Therefore, it is useful not only to find one partition but also to enumerate partitions which satisfy the constraints. Once we enumerate partitions, administrators can evaluate enumerated partitions from various perspectives and select one of them.

Takizawa et al. [59] proposed an algorithm for a special case of the problem in the following way. They first split a target area into square cells and enumerated all partitions such that each connected component contains exactly one shelter. They consider the convexity constraint first introduced by Chen et al. [68]. In their definition, a component containing a shelter s is called convex if the component can be written as the union of rectangles each of which contains s. However, their definition of convexity is limited to square cells.

In this chapter, we reformulate the convexity for *general graphs* from the definition for grid graphs (the case in Takizawa et al. [59]). We formulate the convexity of connected components as a *spanning shortest path forest*, in short, *SSPF*. An SSPF has good properties to avoid intersections of evacuation routes.

Our approach is as follows: First, we construct ZDDs representing a set of partitions satisfying the structural and distance constraints. As we discuss in Section 3.4, it seems computationally difficult to directly construct a ZDD representing a set of partitions simultaneously satisfying all the constraints. Hence we divide the process of construction of the ZDD into some steps. To construct a ZDD efficiently, we propose algorithms based on *frontier-based search*[6, 5, 4], which is a framework to construct a ZDD representing a set of constrained subgraphs in a given graph. In particular, we propose a novel algorithm to enumerate all SSPFs in a given graph with the distance constraint. The efficiency of frontier-based search is usually evaluated in terms of *the width of a ZDD constructed by the algorithm*, which is a rough indication of the computation time and memory usage. As for the general graph partitioning problem, the algorithm with the width of a ZDD $\mathcal{O}(B_f 2^{f^2})$ is known [60], where B_f is the f-th Bell number and f is the maximum frontier size, which is a parameter of a frontier-based search-like algorithm. Our algorithm exploits the property of SSPFs and achieves the width of a ZDD $\mathcal{O}(B_f 2^{rf})$, where r is the number of shelters. This bound is tighter than $\mathcal{O}(B_f 2^{f^2})$ when r is smaller than f.

Second, we obtain a ZDD representing a set of partitions satisfying all the constraints by operations between ZDDs. Here we propose an algorithm to deal with the population constraint. Our algorithm first constructs a ZDD representing a set containing all the minimal patterns violating the population constraint, and then extract solutions using operations between ZDDs. To construct the ZDD, we also devise a new algorithm based on frontier-based search. The width of a ZDD constructed by our algorithm is $\mathcal{O}(B_f P)$ where P is the total population over vertices, while that of the previous method [60] is $\mathcal{O}(B_f P^f)$.

To evaluate our proposed algorithm, we conduct numerical experiments using real-world map data. Our algorithm constructs a ZDD representing a set of solutions of input graphs with a hundred of edges in a few minutes.

This chapter is organized as follows. In Section 3.2, we give some preliminaries and formulate our problem. We propose our algorithm in Sections 3.3 and 3.4. Section 3.5 gives experimental results.

3.2 Preliminaries

3.2.1 Notation

In this subsection, the input graph is a vertex and edge weighted graph $G = (V, E, popu, w)$. Assume that G is simple, connected and undirected. Here, $V = \{1, 2, \dots, n\}$ is a vertex set and $E \subseteq \{\{u, v\} \mid u, v \in V\}$ is an edge set. The function $popu : V \to \mathbb{Z}^+$ is a vertex weight function. For a vertex v, $popu(v)$ indicates the population of v. The function $w : E \to \mathbb{R}^+$ is an edge weight function. For an edge e, $w(e)$ means the length of e. Hereinafter, we sometimes drop $popu$ and w from $(V, E, popu, w)$ and write $G = (V, E)$ for simplicity. Let $S = \{1, 2, \dots, r\} \subseteq V$ be a set of *shelters*. Note that $r = |S|$ and $\forall s \in S, \forall v \in V \setminus S,\ s < v$. We are also given $cap : S \to \mathbb{Z}^+$. For a shelter $s \in S$, $cap(s)$ denotes the shelter-capacity of s.

We give some additional notation for this chapter. For a vertex v and a subgraph $E' \subseteq E$, let $C_{E'}(v)$ be the set of the vertices that are connected to v in E', containing v. Intuitively, $C_{E'}(v)$ means the connected component including v in E'. When there is no ambiguity, we omit E' and write $C(v)$. We denote the shortest distance between vertices u and v in G as $d_G(u, v)$. Let $d^*(v)$ be the shortest distance from v to the nearest shelter in G, that is, $d^*(v) = \min\{d_G(s, v) \mid s \in S\}$. B_f denotes the f-th Bell number, which is the number of partition of f items.

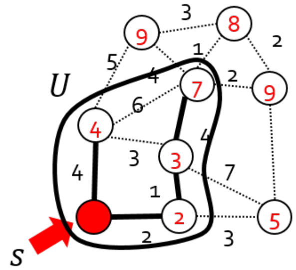


Figure 3.1: Example of a shortest path tree.

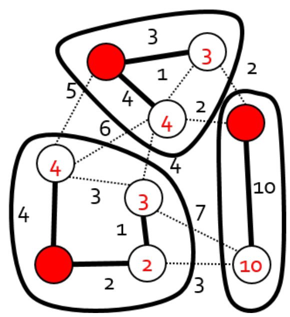


Figure 3.2: Example of a spanning shortest path forest.

3.2.2 Formulation

We introduce the constraints on the structure of components in a partition, distances from each vertex to a shelter, and the shelter-capacity of shelters.

It is required that each component should be connected and that intersections of evacuation routes are avoided. We assume that each evacuee on a vertex evacuates to a shelter along the shortest path from the vertex to the shelter. To impose the constraint, we represent a partition as a *spanning shortest path forest*, in short, *SSPF*. To define an SSPF, we give the definition of a shortest path tree, in short, SPT.

Definition 3.1 (Shortest path tree (SPT)). *We say that* $T = (U, E'), U \subseteq V, E' \subseteq E$, *is a* shortest path tree *(an* SPT*) of* $G = (V, E)$ *rooted at* $s \in S$ *if* T *is a spanning tree of* $G[U]$, $s \in T$ *and* $d_T(s, u) = d_G(s, u)$ *for all* $u \in U$.

Fig. 3.1 shows an example of an SPT. In the figure, the colored vertex is a shelter and thick edges compose the tree. The numbers near the edges are edge weights. Each number in a vertex is the shortest distance from the shelter to itself. Next, an SSPF is defined as follows.

Definition 3.2 (Spanning shortest path forest (SSPF))**.** *We say that* $F = (V, E'), E' \subseteq E$*, is a* spanning shortest path forest *(an* SSPF*) of* $G = (V, E)$ *if every connected component in* F *has exactly one shelter* $s \in S$*, and is an SPT rooted at* s*.*

Fig. 3.2 shows an example of an SSPF. In the figure, colored vertices are shelters. Suppose that an SSPF F is given. We say that $s \in S$ is the *assigned shelter* of v if s is the root of the SPT containing v in F. In F, for all vertices $v \in V$, evacuees on v can go to the assigned shelter in the shortest distance without passing through edges in other trees. This property leads to less intersections of evacuation routes. We call the condition that a partition is represented as an SSPF the *structural constraint*. In what follows, we identify a partition with an SSPF.

Next, we discuss the rest of the constraints. We introduce two parameters $D, R \in \mathbb{R}^+$. D is an upperbound of the distance from any vertex to the assigned shelter. That is, for all $v \in V, d_G(v, s_v) \leq D$ must hold, where s_v is the assigned shelter to v in an SSPF F. In addition to restricting the maximum distance of evacuation routes, we would like to avoid assigning a vertex to a far shelter even though there is another shelter close to the vertex. We impose the restriction that any vertex must not be assigned to a shelter R times farther than the nearest shelter. That is, for all $v \in V$, $d_G(v, s_v) \leq R \cdot d^*(v)$ must hold. We call the above constraint the *distance constraint*. In addition, we introduce a parameter $K \in \mathbb{R}^+$, which is the maximum acceptable ratio of the number of evacuees assigned to a shelter to its shelter-capacity, that is,

$$\forall s \in S, \sum_{v \in C_F(s)} popu(v) \leq K \cdot cap(s), \tag{3.1}$$

which we call the *shelter-capacity constraint*. Note that we cannot assign a vertex to the nearest shelter s' when the total population on the vertices near s' is too much.

As a summary, our problem is defined as follows.

Input

- A vertex and edge weighted graph $G = (V, E, popu, w)$, where

  - vertex set $V = \{1, 2, \ldots, n\}$,
  - edge set $E = \{e_1, e_2, \ldots, e_m\}$,
  - vertex weight function (population) $popu : V \to \mathbb{Z}^+$,
  - edge weight function (distance) $w : E \to \mathbb{R}^+$.

- A set of shelters $S = \{1, 2, \ldots, r\} \subseteq V$,
- Capacities of shelters $cap : S \to \mathbb{Z}^+$,
- Parameters $D, R, K \in \mathbb{R}^+$.

Solution

- An SSPF F of G (the *structural constraint*) satisfying the following constraints:

  1. The *distance constraint*:
  $$\forall v \in V, d(v, s_v) \leq \min\{D, R \cdot d^*(v)\}, \tag{3.2}$$
  where s_v is the nearest shelter to v in F.
  2. The *shelter-capacity constraint*:
  $$\forall s \in S, \sum_{v \in C_F(s)} popu(v) \leq K \cdot cap(s). \tag{3.3}$$

3.3 Structural and distance constraints

Let us describe an overview of our proposed method. Because dealing with all the constraints at the same time seems computationally difficult as we show in Section 3.4, we divide the procedure into three steps:

1. Construct ZDD Z_1 representing the set of all the SSPFs satisfying the distance constraint.
2. Construct ZDD Z_2 representing a set containing all the minimal trees violating the shelter-capacity constraint.
3. Obtain ZDD Z_3 representing the set of all the SSPFs satisfying all the constraints by operations between Z_1 and Z_2.

In the rest of this section, we explain Step 1. First, we explain a basic algorithm for explanation, and then we show a more memory-efficient algorithm.

3.3.1 Basic algorithm

Before explaining the algorithm, we examine the properties of SPTs. Consider an SSPF F. Let $T \subseteq F$ be an SPT rooted at $s \in S$. If an edge $e = \{u, v\}$ is an element of T, one of Eqs. (3.4) and (3.5) is satisfied:

$$d_G(s,u) + w(e) = d_G(s,v), \tag{3.4}$$

$$d_G(s,v) + w(e) = d_G(s,u). \tag{3.5}$$

Conversely, if either Eqs. (3.4) or (3.5) holds for $s \in S$, e can be an element of an SPT rooted at s. Since $w(e) > 0$ for all $e \in E$, Eqs. (3.4) and (3.5) are never satisfied simultaneously. In T, we orient e in the direction $u \to v$ if Eq. (3.4) is satisfied, which implies u is a parent in T, and $v \to u$ if Eq. (3.5) is satisfied. Then T can be seen as a directed tree; the in-degree of s in T is zero and those of others in T are one.

Based on the above discussion, we explain the configuration we use in frontier-based search for our problem. In the following, we show the algorithm and explain the correctness at the same time. In what follows, we describe the configuration stored into a ZDD node, say N, having a label $e_i = \{u, v\}$. Recall that the node N corresponds to a set of subgraphs, which we denote $\mathcal{G}$. The values of the configuration stored into N represent the characteristic of *any* subgraph in $\mathcal{G}$, and conversely, by the merge process described in Section 2.3, two nodes are merged only when the values of the configuration of the two nodes are completely the same. Thus, we pick up a subgraph, say G', in $\mathcal{G}$ as a representative and associate G' with the values of the configuration stored into N. We define the configuration as a tuple $(\texttt{cmp}, \texttt{indeg}, \texttt{valid})$ of three arrays. We explain each arrays in the following.

First, to deal with connected components, for each $x \in F_i$, we introduce and store a function (or an array) $\texttt{cmp}[x]$ into N in the same way as in Section 2.3. Recall that the value $\texttt{cmp}[x]$ is maintained so that for $y, z \in F_i$, $\texttt{cmp}[y] = \texttt{cmp}[z]$ if and only if y and z belong to the same connected component in G'. Here, we maintain the value $\texttt{cmp}[x]$ as $\texttt{cmp}[x] = \min\{y \in F_i \mid y \in C(x)\}$, noting that $C(x)$ means the connected component of G' containing x, including x. Since $\forall s \in S, \forall x \in V \setminus S, s < x$ by definition in Section 3.2, we can detect whether $C(x)$ contains a shelter or not using $\texttt{cmp}$, that is, if $C(x)$ contains some shelter s, $\texttt{cmp}[x] = s \leq r = |S|$. Otherwise $r < \texttt{cmp}[x]$. Hereinafter, we regard the value of $\texttt{cmp}[x]$ in the same light as $C(x)$ in G'.

Second, we introduce $\texttt{indeg}[s][x]$ for $x \in F_i$ and $s \in S$. Consider the connected component $C(x)$ of G' such that $C(x) \cap S = \emptyset$. If some of $e_i, \ldots, e_m$ are added to G' and $C(x)$ is connected to s, $C(x)$ becomes a part of the SPT rooted at s. Recall that since N has the label e_i, G' has edges only in $\{e_1, \ldots, e_{i-1}\}$. Then, each edge in $C(x)$ is oriented in the SPT (rooted at s). We maintain the value of $\texttt{indeg}[s][x]$ so that $\texttt{indeg}[s][x]$ represents the in-degree of x assuming that $C(x)$ is a part of the SPT rooted at s. That is,

$$\texttt{indeg}[s][x] = \left| \left\{ e \in C(x) \;\middle|\; \begin{array}{l} e = \{y, x\}, \\ d_G(s, y) + w(e) = d_G(s, x) \end{array} \right\} \right|. \tag{3.6}$$

Third, when there is an edge e in a connected component C in G' containing no shelter such that neither Eqs. (3.4) nor (3.5) holds for e and $s \in S$, s cannot join C. Therefore, to detect the situation, for each connected component containing no shelter, we store a Boolean value which indicates whether or not each shelter can join the connected component into N as $\texttt{valid}[s][C]$. For all $s \in S$ and a connected component $C > r$, $\texttt{valid}[s][C] = \texttt{true}$ if s can join C, and $\texttt{valid}[s][C] = \texttt{false}$ if not.

We explain how to deal with the structural constraint. Consider the destination of the 1-arc of N (described above). This means that we add the edge $e_i = \{u, v\}$ to G'. Without loss of generality, we can assume the cases are of the following:

(a) $C(u) = C(v)$.

(b) $C(u) \neq C(v)$ and $C(u)$ contains a shelter s_u and $C(v)$ contains a shelter s_v.

(c) $C(u) \neq C(v)$ and $C(u)$ contains a shelter s_u and $C(v)$ contains no shelter.

(d) $C(u) \neq C(v)$ and neither $C(u)$ nor $C(v)$ contains any shelter.

In case (a), if we add e_i to G', we can no longer obtain the solution because a cycle is generated in $G' \cup \{e_i\}$. Therefore case (a) should be pruned. We also have to prune case (b) because we will connect different shelters s_u and s_v. In case (c), if $\texttt{valid}[s_u][C(v)] = \texttt{false}$, we should prune the case. In case pruning does not occur in all the cases above, the rest of the cases are (d) and the following (c'):

(c') $C(u) \neq C(v)$, $C(u)$ contains a shelter s_u, $C(v)$ contains no shelter, and $\texttt{valid}[s_u][C(v)] = \texttt{true}$.

Since we add e_i to G', the connected components $C(u)$ and $C(v)$ are merged in $G' \cup \{e_i\}$. Let $C(uv)$ be the generated connected component, that is, $C(uv) = C(u) \cup C(v)$.

Consider how to update the configuration of a ZDD node in cases (c') and (d) (we call making a node N' as the destination of an arc of N and setting the configuration of N' "updating the configuration"). Suppose that we are making a node N' as the destination of the 1-arc of N.

We describe updating `valid`. In case (c'), $\texttt{valid}[s_u][C(v)] = \texttt{true}$ is ensured because pruning by the condition $\texttt{valid}[s_u][C(v)] = \texttt{false}$ does not occur in case (c'), so we do not have to do anything. In case (d), for all $s \in S$, we set $\texttt{valid}[s][C(uv)]$ in N' to be `true` if and only if $\texttt{valid}[s][C(u)] = \texttt{true}$ and $\texttt{valid}[s][C(v)] = \texttt{true}$ in N. If $\texttt{valid}[s][C(uv)]$ is `false` for all $s \in S$ after updating, any shelter can no longer join $C(uv)$. Therefore we prune this case.

Next, we describe updating not only `valid` but also `indeg`. In case (c'), we have the following two situations.

(c'1) Equation (3.4) is satisfied for e_i and s_u.

(c'2) Otherwise.

Case (c'1) means that if e_i will be included in the SPT rooted at s_u in the future, the orientation of e_i in tree must be $u \to v$. Hence, if case (c'1) holds, adding e_i to G' increases the in-degree of v in the SPT (under construction) rooted at s_u. Therefore, in case (c'1), if $\texttt{indeg}[s_u][v] = 1$ holds, we cannot add e_i to G'. Therefore we prune this case. Otherwise ($\texttt{indeg}[s_u][v] = 0$) we substitute 1 for $\texttt{indeg}[s_u][v]$ and go on the procedure. In case (c'2), we cannot add e_i to G' and prune this case. In case (d), for each s, the following three cases are considered:

(d1) Equation (3.4) is satisfied for e_i and s.

(d2) Equation (3.5) is satisfied for e_i and s.

(d3) Neither Eqs. (3.4) nor (3.5) is satisfied for e_i and s.

Similarly to the above discussion, in case (d1), if $\texttt{indeg}[s][v] = 1$ in N, we cannot add e_i to G'. Therefore, in such cases, we substitute `false` for `valid`$[s][C(v)]$ in N', otherwise 1 for `indeg`$[s][v]$ in N'. Case (d2) is almost the same as case (d1). In case (d3), we substitute `false` for `valid`$[s][C(uv)]$ in N'. The difference between (c') and (d) is that now we do not perform pruning immediately but updating `valid`. Similarly to the discussion in case (c'), if `valid`$[s][C(uv)]$ is `false` in N' for all $s \in S$, we prune the case.

We can deal with the distance constraint by initializing `valid`$[s][\{v\}]$ for all $s \in S$ when a vertex v appears on a frontier. Let `valid`$[s][\{v\}] \leftarrow$ `true` if $d(s, v) \leq \min\{D, R \cdot d^*(v)\}$, otherwise `valid`$[s][\{v\}] \leftarrow$ `false`.

3.3.2 More memory-efficient algorithm

In Section 3.3.1, we store `indeg` into ZDD nodes because we want to know in-degrees of vertices on a frontier in the SPT (under construction) rooted at each $s \in S$. Here, for reducing the memory consumption, we propose not to store `indeg`; we can know in-degrees of vertices in the SPTs from other stored values. In the algorithm of Section 3.3.1, a connected component C can be a part of the SPT rooted at $s \in S$ if `valid`$[s][C] =$ `true`. In other words, when `valid`$[s][C] =$ `true`, we can see C as a part of a directed tree rooted at s. Moreover, the directions of the edges in C in the tree can be determined according to Eqs. (3.4) and (3.5): $u \to v$ holds if $d_G(s, u) < d_G(s, v)$. Thus, we have the only one vertex v such that `indeg`$[s][v] = 0$ in the directed tree of C, which is nearest to s in C. Other vertices u in C have `indeg`$[s][u] = 1$. We can find v by comparing $d_G(s, u)$ among vertices u in C. Note that $d_G(s, u)$ does not change throughout the construction of the ZDD, and thus we can replace individual `indeg` in all ZDD nodes by common $d_G(s, u)$, which can be managed globally. Using this idea, we can realize the same algorithm as Section 3.3.1 without storing `indeg` into ZDD nodes. This reduces memory consumption. Pseudocode is presented in Algorithms 3.1–3.5.

Let us consider the width of a ZDD constructed by our algorithm. As configurations, we store `cmp` and `valid` in each ZDD node. There are B_f different states for `cmp` among ZDD nodes with the same label, and 2^{rf} for `valid` (Recall that r is the number of shelters). Thus, we obtain the following lemma.

Lemma 3.1. *The width of a ZDD constructed by Algorithms 3.1–3.5 is* $\mathcal{O}(B_f 2^{rf})$.

In the algorithm in Section 3.3.1, we store an array `cmp` and matrices `valid` and `indeg` into each ZDD node. `cmp` has f elements and `valid` and indeg have rf elements respectively, and thus we store $(2r + 1)f$ values into each ZDD node in the algorithm in Section 3.3.1. By contrast, in the algorithm proposed in this subsection, we store only $(r+1)f$ values into each ZDD node because we do not store indeg.

3.4 Shelter-capacity constraint

In this section, we propose how to deal with the shelter-capacity constraint efficiently. Kawahara et al. [60] have been proposed an algorithm for the shelter-capacity constraint. Their approach is to store the total population of each connected component into ZDD nodes as an additional configuration. Let A be an algorithm to construct a ZDD for a set of constraints $\mathcal{C}$, where $\mathcal{C}$ is a set of constraints without the shelter-capacity constraint. Then, their approach makes the algorithm B to construct a ZDD for $\mathcal{C}$ and the shelter-capacity constraint. However, when the width of a ZDD constructed by A is $\mathcal{O}(g(f))$, that of B is $\mathcal{O}(g(f)P^f)$, where P is the total population over vertices. This can desperately increase the number of ZDD nodes, which is likely to limit the sizes of solvable instances.

Based on the above observation, we devise a new method to deal with the shelter-capacity constraint. Our idea is that we construct a ZDD representing a set containing all the *forbidden minimal patterns*. In particular, we construct a ZDD Z_2 with the following properties:

1. $\forall G' \in Z_2, G'$ is a tree containing exactly one shelter s,
2. $\forall G' \in Z_2$, the total population over vertices in G' exceeds $cap(s)$,
3. Z_2 contains all the minimal trees violating the shelter-capacity constraint.

Once we construct such Z_2, we can obtain a ZDD Z_3 representing all the solutions satisfying all the constraints using operations between Z_1 and Z_2, obtained in Section 3.3, as we describe later in this section.

We propose an algorithm to construct Z_2 based on frontier-based search. For simplicity, we first consider the case $K = 1$. We now store two configurations into each ZDD node: cmp and sm_popu. The configuration cmp is almost the same as described in Section 3.3.1. However, here we use the new value -1. cmp$[v] = -1$ indicates v has not been adopted yet. We say v is adopted if at least one edge incident to v is adopted. sm_popu is the total populations of adopted vertices. Using these configurations, frontier-based search can be performed as follows: Consider the situation we make a new ZDD node N' as a descendant of 1-arc of a ZDD node N with the label $e_i = \{u, v\}$. Similarly to Section 3.3.1, we pick up a subgraph G' as a representative of a set of subgraphs represented by N. If $\texttt{cmp}[x] = -1$ holds for $x \in e_i$ in N, x is adopted. Therefore we set $\texttt{cmp}[x] \leftarrow x$ in N', to initialize x as an isolated vertex[1]. Because x is adopted, the total population of adopted vertices is updated as $\texttt{sm\_popu} \leftarrow \texttt{sm\_popu} + popu(x)$ in N'. After calculating sm_popu in N', if the current value of sm_popu in N' is never that of a minimal tree, we can prune such a case. To detect this, we calculate two grobal variables in advance: $cap\_max = \max\{cap(v) \mid v \in S\}$ and $popu\_max = \max\{popu(v) \mid v \in V\}$. If $\texttt{sm\_popu} > cap\_max + popu\_max$ holds in N', the solution can never be the minimal tree violating the shelter-capacity constraint. Such a case can be pruned. We should prune the case $\texttt{cmp}[u] = \texttt{cmp}[v] \neq -1$ holds in N' because adding e_i to G' in this case yields a cycle. If all the above pruning did not occur, then we merge two connected components $C(u)$ and $C(v)$ and update cmp.

Next, we consider the situation we make a new ZDD node as a descendant of x-arc ($x \in \{0, 1\}$) of a ZDD node N with the label $e_i = \{u, v\}$. First, if there exists only one connected component C in the frontier, C contains a shelter s, and $\texttt{sm\_popu} > cap(s)$ in N', then C satisfies 1 and 2. So we should make **1** as a new node. Second, if there exists a connected component C leaving the frontier in N', C leaves the frontier before violating the population constraint, and therefore we should make **0**. In the case $i = m$, which indicates G' has no edges, we should also make **0**.

In order to extend the algorithm to cases such that $K > 1$, we only have to set $cap(s) \leftarrow K \cdot cap(s)$ for all $s \in S$ before running the algorithm. Pseudocode is presented in Algorithm 3.6.

Let us consider the width of a ZDD constructed by Algorithm 3.6. Algorithm 3.6 stores cmp and sm_popu into ZDD nodes as configurations. There are $\mathcal{O}(B_f)$ different states for cmp among ZDD nodes with the same label and $\mathcal{O}(P)$ for sm_popu[2]. Therefore we obtain the following lemma.

[1] Since we adopt e_i, x is actually not an isolated vertex (at least it is connected with the other vertex in e_i). However, we update cmp later (in lines 11–14 in Algorithm 3.6), and thus we can simply set $\texttt{cmp}[x] \leftarrow x$ here without loss of correctness.

[2] In practice, if P is big, we can round the values of population. Then the complexity $\mathcal{O}(\mathcal{P})$ changes to $\mathcal{O}(P')$, where P' is the total population of rounded values.

Lemma 3.2. *The width of a ZDD constructed by Algorithm 3.6 is $\mathcal{O}(B_f P)$.*

Now we have ZDDs Z_1 and Z_2. We can obtain the ZDD Z_3 representing the set of all the solutions satisfying all the constraints by

$$Z_3 = Z_1 \searrow Z_2 = \{\alpha \in Z_1 \mid \forall \beta \in Z_2, \alpha \not\supseteq \beta\}. \tag{3.7}$$

This operation is known as *nonsupset* [5]. The operation can be realized as follows by using set difference and restrict operation defined in Section 2.2:

$$Z_3 = Z_1 \setminus (Z_1 \rhd Z_2). \tag{3.8}$$

When we construct $Z_1 \rhd Z_2$, the smaller number of nodes of Z_2 leads to faster calculation. However, as we show in Section 3.5, the number of nodes of Z_2 is sometimes considerably larger than that of Z_1. Thus we give a more efficient procedure. The key point is that some tree in Z_2 may not be an SPT or, even so, it may not satisfy the distance constraint. If we eliminate such trees from Z_2 in advance, the number of nodes of Z_2 may become smaller. Although we can realize this by modifying Algorithm 3.6, it makes the time complexity of the algorithm worse. Therefore we use an operation between ZDDs instead. We use permit operation and modify Eq. (3.8) as follows:

$$Z_3 = Z_1 \setminus (Z_1 \rhd Z_2'), \tag{3.9}$$

where

$$Z_2' = Z_2 \lhd Z_1. \tag{3.10}$$

3.5 Experimental results

We conducted numerical experiments to confirm the efficiency of our proposed algorithm in terms of time and memory. We used a machine with an Intel Xeon Processor E7-8870 (2.4GHz) CPU and a 2 TB memory (Oracle Linux 6.7) for the experiments. All code was implemented in C++ (g++4.4.7 with the -O3 optimization). We used the `TdZdd` library [69] to implement algorithms based on frontier-based search. To perform operations between ZDDs, we adopted the `SAPPOROBDD` library.

Figure 3.3: The map data of the target area. The red circles are the shelters. (© OpenStreetMap contributors)

3.5.1 Dataset

We applied our algorithm to real-world map data. A target area is Higashishiga, Kita Ward, Nagoya City in Japan. We first obtained map data of the target area from `openstreetmap.org`[3], and then created graphs representing road networks within specified ranges of latitude and longitude. The number of vertices is 165 and that of edges is 212 in this graph. We set $w(e) \leftarrow \lceil x_e \rceil$ for all edges e, where x_e is the original length (meter) of e in the map and, for a real number a, $\lceil a \rceil$ is the smallest integer which is not less than a. The locations of shelters are obtained from the official web site of Nagoya City[4]. We assumed that each shelter s is located on the intersection closest to s in the road network. The map data and the locations of shelters are shown in Fig. 3.3. We assumed that $popu(v) = 1$ for all $v \in V$ and set the capacities of shelters proportional to the real capacities so that their summation equals to the number of vertices in the graph, as shown in Table 3.1.

[3] `https://www.openstreetmap.org`

[4] `http://www.city.nagoya.jp/bosaikikikanri/cmsfiles/contents/0000090/90892/ura_03kita.pdf` (in Japanese)

3.5.2 Preprocessing

To enable us to deal with larger networks, we preprocessed graphs and reduced the numbers of vertices and edges. We conducted three types of preprocessing. First, edges which is never contained in a shortest path from any shelter to any vertex can be deleted because such edges can never be contained in any SPT. Therefore, for $e = \{u, v\} \in E$, if $\forall s \in S, |d(s,u) - d(s,v)| \neq w(e)$, we delete e. Second, because of the distance constraint, there may be some vertex v' such that v' can only be assigned to the shelter closest to v'. We can contract such v' to the shelter closest to v' before running the proposed algorithm. Third, a vertex v whose degree is one must be in the same connected component as a vertex u which is adjacent to v. Therefore we can contract v to u. We repeat this until the graph does not have a vertex whose degree is one.

3.5.3 Results

We show the results in Table 3.2. D, R and K are the parameters described in Section 3.2.2, and n and m are the number of vertices and edges in the graph after preprocessing. Groups of columns Z_1, Z_2 and Z_3 show experimental results about constructing ZDDs described in Sections 3.3 and 3.4. Columns "# node" indicate the numbers of ZDD nodes after reduction and "Time" is the time to construct ZDDs including the time to reduce ZDDs (in seconds). The last column "# solution" shows the number of partitions satisfying all the constraints for each parameter.

For all the graphs, our algorithm succeeded in constructing the final ZDD Z_3 within a few minutes. The time to construct Z_1 is always shorter than that to Z_2. This is because less merging of nodes occur in the construction of Z_2, where we maintain the total population of adopted vertices. The time to construct Z_3 from Z_1 and Z_2 is lower than that to construct Z_1 and Z_2. For each graph, although the number of obtained solutions is over 10^8, the number of nodes in Z_3 is a few thousands. This shows that our approach, constructing ZDDs, successfully enumerated partitions as a compressed representation. Using the constructed ZDD and operations between ZDDs, we can deal with more constraints and find good solutions.

3.5.4 Discussion

We have some additional discussions in this subsection. First, we discuss the relationship between the number of ZDD nodes and the time to construct ZDDs. In Table 3.2, it seems that there is no relationship between the number of nodes of Z_2 and its construction time. However, note that the number of nodes in Table 3.2 is that of nodes after reduction. The time to construct Z_2 mainly depends on its number of nodes before reduction. We show the numbers of nodes of Z_2 before reduction in Table 3.3. According to Table 3.3, it is clear that the larger the number of nodes of Z_2 before reduction is, the longer it takes to construct Z_2.

Next, we discuss the relationship between the time to construct Z_3 and the numbers of nodes of Z_1 and Z_2. We show the number of nodes of Z'_2, which is calculated by Eq. (3.10), in Table 3.4. Although the number of nodes of Z_2 is sometimes more than six million, that of Z'_2 is less than a thousand. This leads to the efficient construction of Z_3 in Eq. (3.9).

Finally, we compare our algorithm with the others. We can extend the algorithm of Takizawa et al. [59] in the following way: For each shelter s, we first enumerate SPTs rooted at s satisfying the distance and the shelter-capacity constraints by reverse search [21]. Then we construct ZDDs representing the set of the SPTs and combine ZDDs for each shelter using operations between ZDDs. Note that there are two SPTs which have the same vertex set but different edge sets. The algorithm based on that of Takizawa et al. cannot distinguish two SPTs with the same vertex sets and different edge sets. In contrast, our approach can distinguish SPTs as edge sets, which is useful to design evacuation routes. Therefore we first enumerate SPTs for a shelter as edge sets and then convert them into vertex sets and eliminate duplication. The reverse search for SPTs rooted at s can be designed by defining the root node of the search tree by the empty edge set and the parent of $E' \subseteq E, E' \neq \emptyset$ as $E' \setminus \{e_i\}$, where i is the maximum index such that $E' \setminus \{e_i\}$ is an SPT rooted at s. The algorithm takes $\mathcal{O}(m^2)$ time to output one edge set. However, there may be an exponential number of SPTs with the same vertex set and different edge sets. Therefore the time per SPTs distinguished by vertex sets is not bounded by a polynomial of m.

We implemented the above algorithm based on reverse search. We run the algorithm and measured the total time to enumerate SPTs for all the shelters in each input graph. The timeout is set to 100 hours. We show the results in Table 3.5. In the table, the unit of time is hour and the value is rounded down to the second decimal place. The enumeration finished within 100 hours only in G_1 and G_4. The time for them exceeds 40 hours. In contrast, our approach based on frontier-based search enumerates SSPFs implicitly and thus succeeds in enumerating all the solutions in a few minutes in spite of the big solution space.

3.6 Conclusion

In this chapter, we have dealt with the evacuation planning problem. We reformulate the convexity of components as spanning shortest path forests (SSPFs) to deal with general graphs and have proposed an algorithm to construct a ZDD representing a set of SSPFs. We have also proposed algorithms to deal with the distance and capacity constraints efficiently. As shown in experimental results using real-world map data, the proposed algorithm can construct ZDDs in a few minutes for input graphs with hundreds of edges. As future work, it is important to consider new constraints such as the reliability of roads.

Algorithm 3.1: MakeNewNode1($N, i, take$)

```
Let e_i = {u, v}.
Copy N to N'.
if take = 1 then
    if cmp[u] = cmp[v] then
        return 0                                   // A cycle is generated.
    else if cmp[u] ≤ r and cmp[v] ≤ r then
        return 0                                   // connect shelters
    else if cmp[u] ≤ r and r < cmp[v] and
      valid[cmp[u]][cmp[v]] = false then
        return 0
    else if cmp[v] ≤ r and r < cmp[u] and
      valid[cmp[v]][cmp[u]] = false then
        return 0
    if UpdateState(N', i) returns false then
        return 0
if e_i is the last edge adjacent to C and C does not contain any
  shelter then
    return 0   // A connected component without any shelter
      is generated.
for x ∈ e_i such that x ∉ F_i and cmp[x] > r do
    for s ∈ S do
        if IsNearestInCmp(N', x, s) returns true then
            // a vertex with in-degree zero leaves the
               frontier before it connects to any shelter.
            valid[s][cmp[x]] ← false
    if valid[s][cmp[x]] returns false for all s ∈ S then
        return 0  // cmp[x] can no longer be connected to any
          shelter.
if i = m then
    return 1                  // All the constraints are satisfied.
return N'
```

Algorithm 3.2: UPDATESTATE(N', i)

```
// update information of node N' when we adopt e_i
if CheckIndeg(N', i) returns false then
    return false
if UpdateValid(N', i) returns false then
    return false
// update cmp
C_min = min{cmp[u], cmp[v]}
C_max = max{cmp[u], cmp[v]}
for x ∈ F_{i-1} ∪ e_i such that cmp[x] = C_max do
    cmp[x] ← c_min
return true
```

Algorithm 3.3: CheckIndeg(N', i)

```
// check if we can adopt e_i in N' with respect to the
   constraint of in-degrees
if cmp[u] > r and cmp[v] ≤ r then
    swap u and v.
if cmp[u] ≤ r and cmp[v] > r then
    // cmp[u] contains a shelter and cmp[v] contains no
       shelter.
    s ← cmp[u]
    if d(s, u) + w(e_i) ≠ d(s, v) then
        return false
    else if IsNearestInCmp(N', v, s) returns false then
        return false        // the in-degree of v is not zero.
else
        // Neither cmp[u] nor cmp[v] contains any shelter.
    for s ∈ S do
        if d(s, v) + w(e_i) = d(s, u) then
            swap v and u.
        if d(s, u) + w(e_i) = d(s, v) then
            if IsNearestInCmp(N', v, s) returns false then
                valid[s][cmp[v]] ← false
        else
            valid[s][cmp[u]] ← false
            valid[s][cmp[v]] ← false
return true
```

Algorithm 3.4: UPDATEVALID(N', i)

```
// update valid of the new connected component when we adopt e_i
C_min ← min{cmp[u], cmp[v]}
C_max ← max{cmp[u], cmp[v]}
if C_min > r then
    // merges connected components containing shelters
    for s ∈ S do
        valid[s][C_min] ← valid[s][C_min] and valid[s][C_max]
    if valid[s][C_min] = false for all s ∈ S then
        return false    // C_min can no longer be connected to any shelter
return true
```

Algorithm 3.5: ISNEARESTINCMP(N', x, s)

```
// check if x is the nearest vertex in cmp[x] to s
for y ≠ x such that cmp[y] = cmp[x] do
    if d(s, y) ≤ d(s, x) then
        return false
return true
```

Algorithm 3.6: MakeNewNode2$(N, i, take)$

```
Let e_i = {u, v}.
Copy N to N'.
if take = 1 then
    for x ∈ e_i such that cmp[x] = −1 do
        cmp[x] ← x
        sm_popu ← sm_popu + popu(x)
    if sm_popu > cap_max + popu_max then
        return 0
    if cmp[u] = cmp[v] ≠ −1 then
        return 0
    // update cmp
    C_min = min{cmp[u], cmp[v]}
    C_max = max{cmp[u], cmp[v]}
    for x ∈ F_{i−1} ∪ e_i such that cmp[x] = c_max do
        cmp[x] ← c_min
if C is the only connected component on the frontier and C contains
  a shelter s and sm_popu > cap(s) then
    return 1
if there exists a connected component leaves the frontier or i = m
  then
    return 0
return N'
```

Table 3.1: Capacities of shelters.

shelter	shelter-capacity
s_1	37
s_2	71
s_3	51
s_4	4

Table 3.2: Experimental results for real-world map data.

Graph Name	D	R	K	n	m	Z_1		Z_2		Z_3		# solution
						# node	Time	# node	Time	# node	Time	
G_1	700	2	5	83	117	1888	12.00	530	76.75	1159	0.00	171317520
G_2	700	2.5	4	100	139	1972	0.34	490	3.33	1140	0.00	124372832
G_3	700	3	3	117	157	7123	3.63	6314175	109.44	2207	1.74	596788044
G_4	900	2	5	83	117	1806	11.62	530	76.85	1063	0.01	175309200
G_5	900	2.5	4	100	139	1806	0.32	490	3.33	1053	0.00	125734040
G_6	900	3	3	118	158	7908	4.09	6548955	113.16	2734	2.57	680339404

Table 3.3: The numbers of nodes of Z_2 before reduction.

Graph Name	# node
G_1	25307155
G_2	1970112
G_3	41740598
G_4	25307155
G_5	1970112
G_6	43108192

Table 3.4: The numbers of nodes of Z'_2.

Graph Name	# node
G_1	212
G_2	223
G_3	341
G_4	212
G_5	223
G_6	644

Table 3.5: The time to enumerate shortest path trees by reverse search.

Graph Name	Time
G_1	46.29 h
G_2	$>$ 100 h
G_3	$>$ 100 h
G_4	43.91 h
G_5	$>$ 100 h
G_6	$>$ 100 h

Chapter 4

Balanced Graph Partition

4.1 Introduction

Partitioning a graph is a fundamental problem in computer science and has several important applications such as evacuation planning, political redistricting, VLSI design, and so on. In some applications among them, it is often required to balance the weights of connected components in a partition. For example, the task of the evacuation planning is to design which evacuation shelter inhabitants escape to. This problem is formulated as a graph partitioning problem, and it is important to obtain a graph partition consisting of balanced connected components (each of which contains a shelter and satisfies some conditions). Another example is political redistricting, the purpose of which is to divide a region (such as a prefecture) into several balanced political districts for fairness.

For balanced graph partitioning, Kawahara et al. [60] proposed an algorithm to construct a ZDD representing the set of balanced graph partitions by frontier-based search [4, 5, 6], which is a framework to directly construct a ZDD, and applied it to political redistricting. However, their method stores the weights of connected components, represented as integers, into the ZDD, which generates a not compressed ZDD. As a result, the computation is tractable only for graphs only with less than 100 vertices. Nakahata et al. [70] proposed an algorithm to construct the ZDD representing the set of partitions such that all the weights of connected components are bounded by a given upper threshold (and applied it to evacuation planning). Their approach enumerates connected components with weight more than the upper threshold as a ZDD, say *forbidden components*, and constructs a ZDD representing partitions not containing any forbidden component *as a subgraph* by set operations, which are performed by so-called apply-like methods [7]. However, it seems difficult to directly use their method to obtain balanced partitions by letting connected components with weight less than a lower threshold be forbidden components because partitions not containing any forbidden component *as a connected component* (i.e., one of parts in a partition coincides a forbidden component) cannot be obtained by apply-like methods.

In this chapter, for a ZDD $Z_{\mathcal{A}}$ and an integer L, we propose a novel algorithm to construct the ZDD representing the set of graph partitions such that the partitions are represented by $Z_{\mathcal{A}}$ and all the weights of the connected components in the partitions are at least L. The input ZDD $Z_{\mathcal{A}}$ can be the sets of spanning forests used for evacuation planning (e.g., [70]), rooted spanning forests used for power distribution networks (e.g., [57]), and simply connected components representing regions (e.g., [60]), all of which satisfy complex conditions according to problems. We generically call these structures "partitions." Roughly speaking, our algorithm excludes partitions containing any forbidden component as a connected component from $Z_{\mathcal{A}}$. We first construct the ZDD, say $Z_{\mathcal{S}}$, representing the set of forbidden components, each of which has weight less than L. Then, for a component in $Z_{\mathcal{S}}$, we consider the cutset that separates the input graph into the component and the rest. We represent the set of pairs of every component in $Z_{\mathcal{S}}$ and its cutset as a *ternary decision diagram* (TDD) [71], say $T_{\mathcal{S}^{\pm}}$. We propose a method to construct the TDD $T_{\mathcal{S}^{\pm}}$ from $Z_{\mathcal{S}}$ by frontier-based search. By using the TDD $T_{\mathcal{S}^{\pm}}$, we show how to obtain partitions each of which belongs to $Z_{\mathcal{A}}$, contains all the edges in a component of a pair in $T_{\mathcal{S}^{\pm}}$ and contains no edge in the cutset of the pair. Finally, we exclude such partitions from $Z_{\mathcal{A}}$ and obtain the desired partitions. By numerical experiments, we show that the proposed algorithm runs up to tens of times faster than an existing state-of-the-art algorithm.

This chapter is organized as follows. In Section 4.2, we give preliminaries. We describe an overview of our algorithm in Section 4.3.1, and the detail in the rest of Section 4.3. Section 6.5 gives experimental results.

4.2 Preliminaries

4.2.1 Notation

In this chapter, we deal with a vertex-weighted undirected graph $G = (V, E, p)$, Assume that G is simple and connected. where $V = [n]$ is the vertex set and $E = \{e_1, e_2, \ldots, e_m\} \subseteq \{\{u, v\} \mid u, v \in V\}$ is the edge set. The functions $p\colon V \to \mathbb{Z}^+$ and $w\colon E \to \mathbb{R}^+$ give the weights of the vertices and those of the edges, respectively. We often drop p from (V, E, p) when there is no ambiguity. For an edge set $E' \subseteq E$, we call the subgraph (V, E') a *graph partition*. We often identify the edge set E' with the partition (V, E') by fixing the graph G. For edge sets E', E'' with $E'' \subseteq E' \subseteq E$ and a vertex set $V'' \subseteq V$, we say that (V'', E'') is included in the partition (V, E') *as a subgraph*. The subgraph (V'', E'') is called a *connected component* in the partition (V, E') if $V'' = \mathrm{dom}(E'')$ holds, there is no edge in $E' \setminus E''$ incident with a vertex in V'', and for any two distinct vertices $u, v \in V''$, there is a u-v path on (V'', E''), where $\mathrm{dom}(E'')$ is the set of vertices which are endpoints of at least one edge in E''. In this case, we say that (V'', E'') is included in the partition (V, E') *as a connected component*. We denote the neighborhood of a vertex v in a partition $E' \subseteq E$ by $N(E', v) = \{u \mid \{u, v\} \in E'\}$. For $i \in [m]$, $E^{\leq i}$ denotes the set of edges whose indices are at most i. We define $E^{<i}$, $E^{\geq i}$ and $E^{>i}$ in the same way.

For a set U, let $U^+ = \{+e \mid e \in U\}, U^- = \{-e \mid e \in U\}$ and $U^\pm = U^+ \cup U^-$. A *signed set* is a subset of $U^\pm$ such that, for all $e \in U$, the set contains at most one of $+e$ and $-e$. For example, when $U = [3]$, both $\{+1, -2\}$ and $\{-3\}$ are signed sets but $\{+1, -1, +3\}$ is not. A *signed family* is a family of signed sets. In particular, when $U = E$, we sometimes call a signed set a *signed subgraph* and call a signed family a *set of signed subgraphs*. For a signed set $S^\pm$, we define $\mathrm{abs}(S^\pm) = \{e \mid (+e \in S^\pm) \vee (-e \in S^\pm)\}$.

4.2.2 Ternary decision diagram

A *ternary decision diagram* (TDD) [71] is a directed acyclic graph $T = (N_T, A_T)$ representing a signed family. A TDD shares many concepts with a ZDD, and thus we use the same notation as a ZDD for a TDD. The difference between a ZDD and a TDD is that, while a node of the former has two arcs, that of the latter has three, which are called the *ZERO-arc*, the *POS-arc*, and the *NEG-arc*.

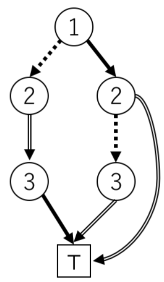


Figure 4.1: The TDD representing the signed family $\{\{+1,-2\},\{+1,-3\},\{-2,+3\}\}$. A dashed arc is a ZERO-arc, a solid single arc is a POS-arc and a solid double arc is NEG-arc. For simplicity, $\perp$ and the arcs pointing at it are omitted.

T represents the signed family in the following way. For a directed path $p = (n_1, a_1, n_2, a_2, \ldots, n_k, a_k, \top) \in \mathcal{P}_T$ with $n_i \in N_Z$, $a_i \in A_T$ and $n_1 = r_T$, we define $S_p^{\pm} = \{+l(n_i) \mid a_i \in A_{T,+}, i \in [k]\} \cup \{-l(n_i) \mid a_i \in A_{T,-}, i \in [k]\}$, where $A_{T,+}$ and $A_{T,-}$ are the set of the POS-arcs of T and the set of the NEG-arcs of T, respectively. We interpret that T represents the signed family $\{S_p^{\pm} \mid p \in \mathcal{P}_T\}$. We illustrate the TDD representing the signed family $\{\{+1,-2\},\{+1,-3\},\{-2,+3\}\}$ in Fig. 4.1 for example. In the figure, a dashed arc ($\dashrightarrow$), a solid single arc ($\rightarrow$), and a solid double arc ($\Rightarrow$) are a ZERO-arc, a POS-arc, and a NEG-arc, respectively. In the figure, $\perp$ and the arcs pointing at it are omitted for simplicity. The TDD in the figure has three directed paths from the root node to $\top$: $1 \rightarrow 2 \Rightarrow \top$, $1 \rightarrow 2 \dashrightarrow 3 \Rightarrow \top$, and $1 \dashrightarrow 2 \Rightarrow 3 \rightarrow \top$, which correspond to $\{+1,-2\}, \{+1,-3\}$, and $\{-2,+3\}$, respectively.

4.3 Algorithms

4.3.1 Overview of the proposed algorithms

In this section, for a ZDD $Z_{\mathcal{A}}$ and $L \in \mathbb{Z}^+$, we propose a novel algorithm to construct the ZDD representing the set of graph partitions such that the partitions are represented by $Z_{\mathcal{A}}$ and each connected component in the partitions has weight at least L. In general, there are two techniques to obtain ZDDs having desired conditions. One is frontier-based search, described in the previous section. The method proposed by Kawahara et al. [60] directly stores the weight of each component into ZDD nodes (as `conf`) and prunes a node when it is determined that the weight of a component is less than L. However, for two nodes, if the weight of a single component on the one node differs from that on the other node, the two nodes cannot be merged. Consequently, node merging rarely occurs in Kawahara et al.'s method and thus the size of the resulting ZDD is too large to construct it if the input graph has more than a hundred of vertices.

The other technique is the usage of the recursive structure of a ZDD. Methods based on the recursive structure are called *apply-like* methods [7]. For each node α of a ZDD, the nodes and arcs reachable from α compose another ZDD, whose root is α. For a ZDD Z and $x \in \{0, 1\}$, let $c_x(Z)$ be the ZDD composed by the nodes and arcs reachable from the x-child of the root. For (one or more) ZDDs F (and G), an apply-like method constructs a target ZDD by recursively calling itself against $c_0(F)$ and $c_1(F)$ (and $c_0(G)$ and $c_1(G)$). For example, the ZDD representing $F \cap G$ can be computed from $c_0(F) \cap c_0(G)$ and $c_1(F) \cap c_1(G)$. Apply-like methods support various set operations [7, 5].

Nakahata et al. [70] developed an algorithm to upperbound the weights of connected components in each partition, i.e., to construct the ZDD representing the set $\mathcal{A}$ of partitions included in a given ZDD and the weights of all the components in the partitions are at most $H \in \mathbb{Z}^+$. Their algorithm first constructs the ZDD $Z_\mathcal{S}$ representing the set of forbidden components (described in the introduction) with weight more than H by frontier-based search. Then, the algorithm constructs the ZDD representing $\{A \in \mathcal{A} \mid \exists S \in \mathcal{S}, A \supseteq S\}$, written as $Z_\mathcal{A}.\mathrm{restrict}(Z_\mathcal{S})$, which means the set of all the partitions each of which includes a component in $\mathcal{S}$ as a subgraph, in a way of apply-like methods. Finally, we extract subgraphs not in $Z_\mathcal{A}.\mathrm{restrict}(Z_\mathcal{S})$ from $Z_\mathcal{A}$ by the set difference operation $Z_\mathcal{A} \setminus (Z_\mathcal{A}.\mathrm{restrict}(Z_\mathcal{S}))$ [8], which is also an apply-like method.

In our case, lowerbounding the weights of components, it is difficult to compute desired partitions by the above approach because a partition including a forbidden component (i.e., weight less than L) *as a subgraph* can be a feasible solution. We want to obtain a partition including a forbidden component *as a connected component*. Although we can perform various set operations by designing apply-like methods, it seems difficult to obtain such partitions by direct set operations.

Our idea in this section is to employ the family of signed sets to represent the set of pairs of every forbidden component and its cutset. We use the

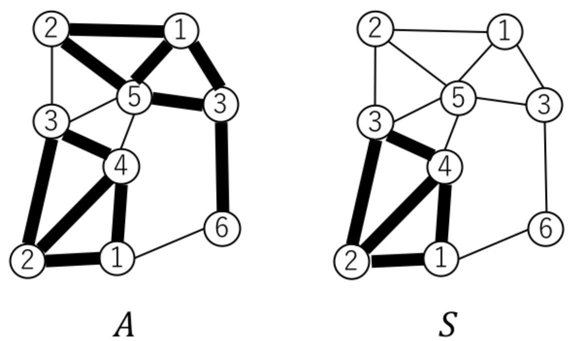


Figure 4.2: Graph partition and its connected component.

following observation.

Observation 4.1. *Let A be a graph partition of $G = (V, E)$ and $S \subseteq E$ be an edge set such that* $(\mathrm{dom}(S), S)$ *is connected. The partition A contains* $(\mathrm{dom}(S), S)$ *as a connected component if and only if both of the following hold.*

1. *A contains all the edges in S.*
2. *A does not contain any edge e in $E \setminus S$ such that e has at least one vertex in* $\mathrm{dom}(S)$.

Based on Observation 4.1, we associate a signed subgraph $S^{\pm}$ with a connected subgraph $(\mathrm{dom}(S), S)$:

$$
\begin{aligned}
S^{\pm} &= S^{+} \cup S^{-}, & (4.1)\\
S^{+} &= \{+e \mid e \in S\}, & (4.2)\\
S^{-} &= \{-e \mid (e = \{u, v\} \in E \setminus S) \wedge (\{u, v\} \cap \mathrm{dom}(S) \neq \emptyset)\}. & (4.3)
\end{aligned}
$$

$S^{\pm}$ is a signed subgraph such that $\mathrm{abs}(S^{+})$ and $\mathrm{abs}(S^{-})$ are sets of edges satisfying Conditions 1 and 2 in Observation 4.1, respectively. Note that $\mathrm{abs}(S^{-})$ is a cutset of G, that is, removing the edges in $\mathrm{abs}(S^{-})$ separates G into the connected component $(\mathrm{dom}(\mathrm{abs}(S^{+})), \mathrm{abs}(S^{+}))$ and the rest. In addition, $\mathrm{abs}(S^{-})$ is minimal among such cutsets. In this sense, we say that $S^{\pm}$ is a *signed subgraph with minimal cutset for S*.

Hereinafter, we call edges in $\mathrm{abs}(S^{+})$ *positive edges*, $\mathrm{abs}(S^{-})$ *negative edges* and the other edges *zero edges*. Fig. 4.2 shows an example of a graph

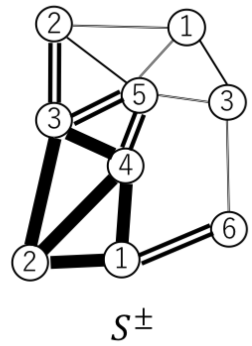


Figure 4.3: Signed subgraph with the minimal cutset. Bold, solid, and double lines indicate positive, zero, and negative edges.

partition A and its connected component S. In the figures, bold lines are edges contained in the partition or the subgraph. Values in vertices are its weights. A contains S as a connected component. The weight of S is $1+2+3+4 = 10$, and thus, when $L > 10$, A does not satisfy the lower bound constraint. Fig. 4.3 shows $S^{\pm}$ associated with S in Fig. 4.2. In the figure, thin single lines, bold single lines, and doubled lines are zero edges, positive edges, and negative edges, respectively. The partition A in Fig. 4.2 indeed contains all the edges in $\mathrm{abs}(S^{+})$ and does not contain any edges in $\mathrm{abs}(S^{-})$. For a graph partition $E' \subseteq E$, when the weights of all the connected components of E' is at least L, we say that E' *satisfies the lower bound constraint.* To extract partitions not satisfying the lower bound constraint from an input ZDD, we compute the set of partitions each of which has all the edges in $\mathrm{abs}(S^{+})$ and no edge in $\mathrm{abs}(S^{-})$ for some $S \in \mathcal{S}$.

The overview of the proposed method is as follows. In the following, let $\mathcal{A}$ be the set of graph partitions represented by the input ZDD and $\mathcal{B}$ be the set of graph partitions each of which belongs to $\mathcal{A}$ and satisfies the lower bound constraint.

1. We construct the ZDD $Z_{\mathcal{S}}$ representing the set $\mathcal{S}$ of forbidden components, where $\mathcal{S}$ is the set of the connected components of G whose weights are less than L.

2. Using $Z_{\mathcal{S}}$, we construct the TDD $T_{\mathcal{S}^{\pm}}$, where $\mathcal{S}^{\pm}$ is a set of signed subgraphs with minimal cutset corresponding to $\mathcal{S}$ by a way of frontier-based search.

3. Using $T_{\mathcal{S}^\pm}$, we construct the ZDD $Z_{\mathcal{S}^\uparrow}$, where $\mathcal{S}^\uparrow$ is the set of partitions each of which contains at least one forbidden component in $\mathcal{S}$ as a connected component.
4. We obtain the ZDD $Z_\mathcal{B}$ by the set difference operation $Z_\mathcal{A} \setminus Z_{\mathcal{S}^\uparrow}$ [8].

In the rest of this section, we describe each step from 1 to 3.

4.3.2 Constructing $Z_\mathcal{S}$

We describe how to construct $Z_\mathcal{S}$, which represents the set $\mathcal{S}$ of forbidden subgraphs whose weights are less than L. In this subsection, we consider only forbidden components with at least one edge. Note that a component with only one vertex cannot be distinguished by sets of edges because all such subgraphs are represented by the empty edge set. We show how to deal with components having only one vertex in Section 4.3.4. In this and the following sections, we show the algorithm and explain the correctness at the same time.

We can construct $Z_\mathcal{S}$ using frontier-based search.We design an algorithm in a similar way as Algorithm 3.6, which deal with the upper-bound constraint. To construct $Z_\mathcal{S}$, in the frontier-based search, it suffices to ensure that every enumerated subgraph has only one connected component and its weight is less than L. The former can be dealt by storing the connectivity of the vertices in the frontier as `comp`. The latter can be checked by managing the total weight of vertices such that at least one edge is incident to as `weight`.

Let us analyze the width of $Z_\mathcal{S}$. For nodes with the same label, there are $\mathcal{O}(B_f)$ different states for `comp` [60], where, for $k \in \mathbb{Z}^+$, B_k is the k-th Bell number and $f = \max\{|F_i| \mid i \in [m]\}$. As for `weight`, when `weight` exceeds L, we can immediately conclude that the subgraphs whose weights are less than L are generated no more. If we prune such cases, there are $\mathcal{O}(L)$ different states for `weight`. As a result, we can obtain the following lemma on the width of $Z_\mathcal{S}$.

Lemma 4.1. *The width of $Z_\mathcal{S}$ is $\mathcal{O}(B_f L)$, where $f = \max\{|F_i| \mid i \in [m]\}$.*

4.3.3 Constructing $T_{\mathcal{S}^\pm}$

In this subsection, we propose an algorithm to construct $T_{\mathcal{S}^\pm}$. First, we show how to construct the TDD representing the set of all the signed subgraphs with minimal cutset, including a disconnected one. Next, we describe the method to construct $T_{\mathcal{S}^\pm}$ using $Z_{\mathcal{S}}$.

Let $S^\pm = S^+ \cup S^-$ be a signed subgraph. Our algorithm uses the following observation on signed subgraphs with minimal cutset.

Observation 4.2. *A signed subgraph $S^\pm$ is a signed subgraph with minimal cutset if and only if the following two conditions hold:*

1. *For all $v \in V$, at most one of a zero edge or a positive edge is incident to v.*
2. *For all the negative edges $\{u, v\}$, a positive edge is incident to at least one of u and v.*

Conditions 1 and 2 in Observation 4.2 ensure that $\mathrm{abs}(S^-)$ is a cutset such that removing it leaves the connected component whose edge set is $\mathrm{abs}(S^+)$ and the minimality of $\mathrm{abs}(S^-)$. This shows the correctness of the observation. We design an algorithm based on frontier-based search to construct a TDD representing the set of all the signed subgraphs satisfying Conditions 1 and 2 in Observation 4.2.

In a similar way to ZDDs, we define configurations to merge equivalent TDD nodes. Here, we define the configuration as a tuple $(\texttt{colors}, \texttt{reserved})$ of two arrays. We explain each array in the following.

First, we consider Condition 1. To ensure Condition 1, we store an array $\texttt{colors} : V \to 2^{\{0,+,-\}}$ into each TDD node. For all $v \in F_{i-1}$, we manage $n_i.\texttt{colors}[v]$ so that it is equal to the set of types of edges incident to v. For example, if a zero edge and a positive edge are incident to v and no negative edges are, $\texttt{colors}[v]$ must be $\{0, +\}$. We can prune the case such that Condition 1 is violated using `colors`, which ensures Condition 1.

Next, we consider Condition 2. Let $\{u, v\}$ be a negative edge. When u and v leave the frontier at the same time, we check if Condition 2 is satisfied from $\texttt{colors}[u]$ and $\texttt{colors}[v]$ and, if not, we prune the case. When one of u or v leaves the frontier (without loss of generality, we assume the vertex is u), if no positive edges are incident to u, at least one positive edge must be incident to v later. To deal with this situation, we store an array $\texttt{reserved} : V \to \{0, 1\}$ into each TDD node. For all $v \in F_{i-1}$, we manage $\texttt{reserved}[v]$ so that $\texttt{reserved}[v] = 1$ if and only if at least one positive edge must be incident to v later. We can prune the cases such that $v \in V$ is leaving the frontier and both $\texttt{reserved}[v] = 1$ and $+ \notin \texttt{colors}[v]$ hold, which violate Condition 2. We show MakeNewNode function and its subroutine Reserve in Algorithms 4.1 and 4.2, respectively.

We give the following lemma on the width of a ZDD constructed by Algorithms 4.1 and 4.2.

Lemma 4.2. *The width W_T of a TDD constructed by Algorithms 4.1 and 4.2 is $W_T = \mathcal{O}(6^f)$.*

Proof. We analyze the number of different non-terminal nodes which are returned by MakeNewNode function and have the label e_i. To this end, we analyze the number of a pair ($\texttt{colors}[w], \texttt{reserved}[w]$) for each $w \in F_{i-1}$. Because of Lines 4–5 in MakeNewNode, $+$ and 0 are never in $\texttt{colors}[w]$ together. In addition, $\texttt{colors}[w]$ is never empty because, when MakeNewNode returns a non-terminal node, there are at least one processed edge incident to w and its type has been added into $\texttt{colors}[w]$ in Line 16. Therefore, there are at most five different states for $\texttt{colors}[w]$: $\{0\}, \{-\}, \{+\}, \{0, -\}$, and $\{-, +\}$. As for $\texttt{reserved}[w]$, it may be 1 only when $\texttt{colors}[w] = \{-\}$ because of Lines 3–4 in Reserve. Thus, there are at most six different states for ($\texttt{colors}[w], \texttt{reserved}[w]$). There are at most f vertices in the frontier, and therefore $W_T = \mathcal{O}(6^f)$. □

Next, we show how to construct $T_{\mathcal{S}^\pm}$ using $Z_{\mathcal{S}}$. We can achieve this goal using *subsetting* technique [69] with Algorithms 4.1 and 4.2. Subsetting technique is a framework to construct a decision diagram corresponding to another decision diagram. We ensure that, for all $S^\pm = S^+ \cup S^- \in \mathcal{S}^\pm$, there exists $S \in \mathcal{S}$ such that $\mathrm{abs}(S^+) = S$ in the construction of $T_{\mathcal{S}^\pm}$ using subsetting technique. For this purpose, we store another configuration `ref`, which is a node of $Z_{\mathcal{S}}$, into each TDD node. We manage $n_T.\texttt{ref}$ in a node n_T of $T_{\mathcal{S}^\pm}$, so that, for any path p_T from r_T to n_T,

(a) there exists a path p_Z from r_Z to $n_T.\texttt{ref}$ in $Z_{\mathcal{S}}$ such that $S_{p_Z} = \mathrm{abs}(S^+_{p_T})$, and

(b) the label of $n_T.\texttt{ref}$ is equal to that of n_T.

To achieve this, we insert the following procedure between Lines 2 and 3 of Algorithm 4.1. We update $n'_i.\texttt{ref}$ by either of two children of $n'_i.\texttt{ref}$ to ensure (b). Let the new value of $n'_i.\texttt{ref}$ be α. If $s = 1$, to ensure (a), α must be the 1-child of $n'_i.\texttt{ref}$ because $s = 1$ implies that we add e_i as a positive edge into all the signed sets represented by n'_i. Otherwise (when $s \in \{0, 2\}$), α must be the 0-child because $s \in \{0, 2\}$ implies that we do not add e_i as a positive edge into any signed set represented by n'_i. If $\alpha = \perp$, we return $\perp$ because we cannot ensure (a) anymore. Otherwise, we go on to Line 3 of Algorithm 4.1. Storing `ref` into each TDD node makes the width of the output TDD larger. The numbers of `ref` in TDD nodes with the same labels are bounded by the width of $Z_{\mathcal{S}}$, so the width of $T_{\mathcal{S}^\pm}$ is bounded by $\mathcal{O}(W_Z 6^f)$, where W_Z is the width of $Z_{\mathcal{S}}$.

4.3.4 Constructing $Z_{\mathcal{S}^\uparrow}$

In this subsection, we show how to construct $Z_{\mathcal{S}^\uparrow}$ and how to deal with forbidden components consisting only of one vertex whose weight is less than L, which was left as a problem in Section 4.3.2. From Observation 4.1 and Eqs. (4.1)–(4.3), $\mathcal{S}^\uparrow$ can be written as

$$\mathcal{S}^\uparrow = \{E' \subseteq E \mid \exists S^\pm \in \mathcal{S}^\pm, (\forall {+e} \in S^\pm, e \in E') \wedge (\forall {-e} \in S^\pm, e \notin E')\}. \quad (4.4)$$

Using $T_{\mathcal{S}^\pm}$, we can construct $Z_{\mathcal{S}}$ by the algorithm of Kawahara et al. [72].

Finally, we show how to deal with a graph partition containing a single vertex v such that $p(v) < L$ as a connected component, i.e., a partition has an isolated vertex with small weight. Let $\mathcal{F}_v$ be the set of graph partitions containing $(\{v\}, \emptyset)$ as a connected component. A graph partition $E' \subseteq E$ belongs to $\mathcal{F}_v$ if and only if E' does not contain any edge incident to v. Using this, we can construct the ZDD Z_v representing $\mathcal{F}_v$ in $\mathcal{O}(m)$ time. For each $v \in V$ such that $p(v) < L$, we construct Z_v and update $Z_{\mathcal{S}^\uparrow} \leftarrow Z_{\mathcal{S}^\uparrow} \cup Z_v$. In this way, we can deal with all the graph partitions containing a connected component whose weight is less than L.

4.4 Experimental results

We conducted computational experiments to evaluate the proposed algorithm and to compare it with the existing state-of-the-art algorithm of Kawahara et al [60]. We used a machine with an Intel Xeon Processor E5-2690v2 (3.00 GHz) CPU and a 64 GB memory (Oracle Linux 6) for the experiments. We have implemented the algorithms in C++ and compiled them by g++ with the `-O3` optimization option. In the implementation, we used the `TdZdd` library [69] and the `SAPPORO_BDD` library.[1] The timeout is set to be an hour.

We used graphs representing some prefectures in Japan for the input graphs. The vertices represent cities and there is an edge between two cities if and only if they have the common border. The weight of a vertex represents the number of residents living in the city represented by the vertex. As for the input ZDD $Z_{\mathcal{A}}$, we adopted three types of graph partitions: graph partitions such that each connected component is an induced subgraph [60], which we call *induced partition*, forests, and rooted forests. There is a one-to-one correspondence between induced partitions and partitions of the vertex set. A rooted forest is a forest such that each tree in the forest has exactly one specified vertex. We chose special vertices for each graph randomly. A summary of input graphs and input graph partitions is in Table 4.1. In the table, we show graph names and the prefecture represented by the graph, the number of vertices (n), edges (m) and connected components (k) in graph partitions. The groups of columns "Induced partition", "Forest", and "Rooted forest" indicate the types of input graph partitions. Inside each of them, we show the size (the number of non-terminal nodes) of $Z_{\mathcal{A}}$ and the cardinality of $\mathcal{A}$.

The lower bounds of weights are determined as follows. Let k be the number of connected components in a graph partition and r be the maximum ratio of the weights of two connected components in the graph partition. From k and r, we can derive the necessary condition that the weight of every connected component must be at least $L(k,r) = P/(r(k-1)+1)$, where $P = \sum_{v \in V} p(v)$ [60]. We used $L(k,r)$ as the lower bound of weights in the experiment. For each graph, we run the algorithms in $r = 1.1, 1.2, 1.3, 1.4,$ and 1.5.

We show the experimental results in Table 4.2. In the table, we show the graph name, the value of r and $L(k,r)$, and the execution time of *Alg. N*, the proposed algorithm, and *Alg. K*, the algorithm of Kawahara et al. The size of $Z_{\mathcal{B}}$ and the cardinality of $\mathcal{B}$ are also shown. "OOM" means *out of memory* and "-" means both algorithms failed to construct the ZDD (due to timeout or out of memory). We marked the values of the time of the algorithm which finished faster as bold.

First, we analyze the results for induced partitions. For the input graphs from G_1 to G_4, both Alg. N and Alg. K succeeded in constructing $Z_{\mathcal{B}}$, except

[1] Although the `SAPPORO_BDD` library is not released officially, you can see the code in `https://github.com/takemaru/graphillion/tree/master/src/SAPPOROBDD`.

when $r = 1.1$ in G_4 for Alg. K. In cases where both algorithms succeeded in constructing $Z_{\mathcal{B}}$, the time for Alg. N to construct the ZDD is 2–32 times shorter than that for Alg. K. In addition, Alg. N succeeded in constructing the ZDD when $r = 1.1$ in G_4, where Alg. K failed to construct the ZDD because of out of memory. These results show the efficiency of our algorithm. In contrast, for G_5, although both algorithms failed to construct the ZDD when $r = 1.1, 1.2, 1.3$ and 1.4, only Alg. K succeeded when $r = 1.5$. In this case, the size of the ZDD constructed by Alg. N did stay in the limitation of memory while, in our algorithm, the size of $Z_{\mathcal{S}^{\uparrow}}$ exceeded the limitation of memory.

Second, we investigate the results for forests. Both Alg. N and Alg. K succeeded in constructing $Z_{\mathcal{B}}$ for the input graph from G_1 to G_4. In all those cases, Alg. N was faster than Alg. K. Comparing the results with those of induced partitions, we found that the execution time of Alg. K depends on the input partitions more than Alg. N does. For example, for G_1, while the execution time of Alg. N is almost irrelevant to the types of input ZDDs, that of Alg. K differ up to about five times. This is because the efficiency of Alg. K strongly depends on the sizes of input ZDDs. This makes the sizes of output ZDDs constructed by Alg. K large, which implies the increase in the execution time of Alg. K. In contrast, the execution time of Alg. N does not depend on the sizes of input ZDDs in many cases because Alg. N uses the input ZDD only in the set difference operation, which is executed in the last of the algorithm (by the existing apply-like method). As we show later, the bottleneck of Alg. N is the construction of $Z_{\mathcal{S}^{\uparrow}}$. Therefore, in many cases, the sizes of input ZDDs do not change the execution time of Alg. N.

Third, we examine the results when the input graph partitions are rooted forests. There are 13 cases such that Alg. K was faster than Alg. N. In the cases, the sizes of input ZDDs and output ZDDs are small, that is, thousands, or even zero. These results show that Alg. K tends to be faster when the sizes of input ZDDs and output ZDDs are small.

In order to assess the efficiency of our algorithm in each step, we show detailed experimental results for G_3 and G_4 when the input graph partitions are induced partitions in Table 4.3. In the table, we show the time to construct decision diagrams, the size of decision diagrams, and the cardinality of the family represented by ZDDs. The cardinality of $S^{\pm}$ is omitted because it is equal to that of $\mathcal{S}$. The size and cardinality for $Z_{\mathcal{A}} \setminus Z_{\mathcal{S}^{\uparrow}}$ are also omitted because they are the same as $|Z_{\mathcal{B}}|$ and $|\mathcal{B}|$, which are shown in Table 4.2. For both G_3 and G_4, the time to construct $Z_{\mathcal{S}}$ and $T_{\mathcal{S}^{\pm}}$ are within one or two seconds. The most time-consuming parts are the construction of $Z_{\mathcal{S}^\uparrow}$ in G_3 and $Z_{\mathcal{S}^\uparrow}$ or $Z_{\mathcal{A}} \setminus Z_{\mathcal{S}^\uparrow}$ in G_4. The set difference operation in G_4 took a lot of time because the sizes of $Z_{\mathcal{A}}$ and $Z_{\mathcal{S}^\uparrow}$ are large, that is, more than a hundred. The reason why the construction of $Z_{\mathcal{S}^\uparrow}$ takes a lot of time is the increase in the sizes of decision diagrams. While the size of $T_{\mathcal{S}^\pm}$ is only 2–7 times larger than that of $Z_{\mathcal{S}}$, that of $Z_{\mathcal{S}^\uparrow}$ is about 10–276 times larger than that of $T_{\mathcal{S}^\pm}$. This also made the execution of the algorithm in G_5 impossible.

4.5 Conclusion

In this chapter, we have proposed an algorithm to construct a ZDD representing all the graph partitions such that all the weights of its connected components are at least a given value. As shown in the experimental results, the proposed algorithm has succeeded in constructing a ZDD representing a set of more than 10^{12} graph partitions in ten seconds, which is 30 times faster than the existing state-of-the-art algorithm. Future work is devising a more memory efficient algorithm that enables us to deal with larger graphs, that is, graphs with hundreds of vertices. It is also important to seek for efficient algorithms to deal with other constraints on weights such that the ratio of the maximum and the minimum of weights is at most a specified value.

Algorithm 4.1: MakeNewNode(n_i, i, s) for constructing a TDD representing the set of signed subgraphs with minimal cutset.

```
// This function returns s(∈ {0,+,−})-child of n_i whose label is e_i.
Let e_i = {u, v}.
Copy n_i to n'_i.
foreach x ∈ {u, v} do
    // violates Condition 1 in Observation 4.2
    if 0 ∈ n'_i.colors[x] and s = + then return ⊥
    if + ∈ n'_i.colors[x] and s = 0 then return ⊥
    if n'_i.colors[x] = {−} and s = 0 then
        // Reserve the vertices in the frontier which are connected to x by the processed edges.
        n'_i ← Reserve(n'_i, N(E^{<i}, x) ∩ (F_{i−1} ∪ F_i))
        if n'_i = ⊥ then return ⊥
    if 0 ∈ n'_i.colors[x] and s = − then
        n'_i ← Reserve(n'_i, e_i \ {x})
        if n'_i = ⊥ then return ⊥
    if n'_i.reserved[x] = 1 and s = 0 then
        return ⊥
    if n'_i.reserved[x] = 1 and s = + then
        n'_i.reserved[x] ← 0          // The reservation is archived.
    n'_i.colors[x] ← n'_i.colors[x] ∪ {s}
foreach x ∈ {u, v} do
    if x ∉ F_i then
        // x is leaving the frontier.
        if n'_i.reserved[x] = 1 and + ∉ n'_i.colors[x] then
            // Although x is reserved, no positive edges are incident to x.
            return ⊥
        if n'_i.colors[x] = {−} then
            // Reserve the vertices in the frontier which are connected to x by the processed edges.
            n'_i ← Reserve(n'_i, N(E^{≤i}, x) ∩ (F_{i−1} ∪ F_i))
            if n'_i = ⊥ then return ⊥
        // Delete the information about the vertices leaving the frontier.
        n'_i.colors[x] ← {}
        n'_i.reserved[x] ← 0
if i = m then
    return ⊤          // All the constraints are satisfied.
return n'_i
```

Algorithm 4.2: RESERVE(n', X)

```
// This function reserves the vertices in X ⊆ V in a TDD
   node n′ and returns the node n″ who has an updated
   state from n′.
Copy n′ to n″.
for x ∈ X do
    // We cannot reserve x if there is a zero edge
       incident to x.
    if 0 ∈ n″.colors[x] then return ⊥
    // Reserve x if there are no positive edges incident
       to x.
    if + ∉ n″.colors[x] then n″.reserved[x] ← 1
return n″
```

Table 4.1: Summary of input graphs and input graph partitions.

				Induced partition		Forest		Rooted forest	
Name	n	m	k	$\|Z_{\mathcal{A}}\|$	$\|\mathcal{A}\|$	$\|Z_{\mathcal{A}}\|$	$\|\mathcal{A}\|$	$\|Z_{\mathcal{A}}\|$	$\|\mathcal{A}\|$
G_1 (Gumma)	37	80	4	10236	1.25×10^{8}	26361	1.01×10^{19}	8957	1.66×10^{16}
G_2 (Ibaraki)	44	95	7	17107	6.38×10^{13}	15553	6.14×10^{23}	3238	1.94×10^{19}
G_3 (Chiba)	60	134	14	301946	6.69×10^{22}	213773	4.86×10^{33}	15741	5.04×10^{25}
G_4 (Aichi)	69	173	17	1598213	9.26×10^{29}	879361	1.78×10^{42}	43465	3.10×10^{30}
G_5 (Nagano)	77	185	5	13203	2.77×10^{17}	44804	2.95×10^{43}	26476	7.66×10^{39}

Table 4.2: Experimental results for three types of input graph partitions.

			Induced partition				Forest				Rooted forest			
	r	$L(r,k)$	Alg. N	Alg. K	$\lvert Z_{\mathcal{B}}\rvert$	$\lvert\mathcal{B}\rvert$	Alg. N	Alg. K	$\lvert Z_{\mathcal{B}}\rvert$	$\lvert\mathcal{B}\rvert$	Alg. N	Alg. K	$\lvert Z_{\mathcal{B}}\rvert$	$\lvert\mathcal{B}\rvert$
G_1	1.1	458947	**4.22**	12.07	4912	1.74×10^4	**4.03**	50.84	29502	8.24×10^{12}	**3.95**	14.96	17920	3.52×10^{11}
	1.2	429016	**2.06**	10.50	3500	5.40×10^4	**2.04**	47.30	21364	3.10×10^{13}	**2.02**	13.34	6331	1.68×10^{12}
	1.3	402750	**1.15**	7.49	2986	9.02×10^4	**1.18**	36.10	18113	7.42×10^{13}	**1.17**	10.54	4655	4.44×10^{12}
	1.4	379514	**0.99**	5.72	3115	2.52×10^5	**1.03**	24.41	20605	3.84×10^{14}	**1.03**	6.97	7677	3.18×10^{13}
	1.5	358813	**0.90**	5.12	3562	2.99×10^5	**0.89**	23.29	20367	7.19×10^{14}	**0.88**	6.52	6719	6.17×10^{13}
G_2	1.1	383928	**3.70**	29.48	27927	1.91×10^6	**3.60**	35.28	47461	2.56×10^{13}	3.53	**2.19**	391	4.32×10^6
	1.2	355836	**3.03**	23.03	83053	1.25×10^8	**2.92**	25.59	143455	2.11×10^{15}	2.95	**1.81**	3103	3.72×10^9
	1.3	331574	**1.73**	16.25	92334	1.02×10^9	**1.70**	18.09	154449	1.41×10^{16}	**1.60**	1.74	5861	1.36×10^{11}
	1.4	310410	**1.21**	12.45	105507	4.54×10^9	**1.30**	14.03	179186	1.02×10^{17}	**1.28**	1.55	5710	1.54×10^{12}
	1.5	291785	**0.73**	8.88	98231	1.25×10^{10}	**0.74**	9.38	149403	3.06×10^{17}	**0.70**	1.21	5855	6.74×10^{12}
G_3	1.1	377742	**83.76**	1008.11	0	0	**77.19**	811.03	0	0	78.68	**66.96**	0	0
	1.2	348159	**32.87**	852.47	6641	2.32×10^5	**27.12**	657.89	17252	1.34×10^{13}	**27.27**	89.75	0	0
	1.3	322874	**23.33**	626.94	261978	3.12×10^{10}	**20.87**	452.10	768876	1.53×10^{19}	**36.20**	36.30	0	0
	1.4	301013	**12.08**	386.91	328581	4.92×10^{11}	**10.88**	266.19	917102	3.23×10^{20}	**9.70**	22.14	0	0
	1.5	281924	**10.81**	315.40	405816	3.02×10^{12}	**9.29**	205.90	1062331	9.94×10^{20}	**7.64**	19.44	606	2.88×10^{10}
G_4	1.1	402370	**155.05**	OOM	190520	1.54×10^{10}	**64.12**	1032.53	374111	5.43×10^{18}	51.95	**0.65**	0	0
	1.2	370499	**86.91**	628.93	739356	1.98×10^{14}	**24.09**	317.44	1374522	1.41×10^{23}	20.82	**0.96**	0	0
	1.3	343307	**125.06**	408.97	1148330	1.98×10^{16}	**14.83**	190.25	2005760	7.27×10^{24}	11.69	**1.48**	0	0
	1.4	319833	**108.25**	281.81	1465722	6.32×10^{17}	**12.18**	134.15	2495000	1.87×10^{26}	8.31	**3.09**	5645	2.19×10^{11}
	1.5	299363	**29.13**	190.59	1761682	1.65×10^{19}	**9.60**	85.84	2434632	4.02×10^{27}	5.55	**3.46**	15587	9.56×10^{14}
G_5	1.1	388844	> 1 h	OOM	-	-	> 1 h	OOM	-	-	> 1 h	$<$ **0.01**	0	0
	1.2	362027	> 1 h	OOM	-	-	> 1 h	OOM	-	-	> 1 h	$<$ **0.01**	0	0
	1.3	338670	OOM	OOM	-	-	> 1 h	OOM	-	-	> 1 h	$<$ **0.01**	0	0
	1.4	318145	OOM	OOM	-	-	OOM	OOM	-	-	OOM	$<$ **0.01**	0	0
	1.5	299965	OOM	**1960.28**	393178	9.20×10^{13}	OOM	OOM	-	-	OOM	$<$ **0.01**	0	0

Table 4.3: Detailed experimental results for G_3 and G_4.

		$Z_{\mathcal{S}}$			$T_{\mathcal{S}^{\pm}}$		$Z_{\mathcal{S}^{\uparrow}}$			$Z_{\mathcal{A}} \setminus Z_{\mathcal{S}^{\uparrow}}$
	r	time	node	card	time	node	time	node	card	time
G_3	1.1	1.90	54745	4.24×10^{8}	0.93	99057	75.88	2117874	2.17532×10^{40}	5.05
	1.2	1.01	39845	1.67×10^{8}	0.69	75581	27.94	977840	2.17528×10^{40}	3.23
	1.3	0.58	31030	6.62×10^{7}	0.51	60034	18.83	814538	2.17498×10^{40}	3.41
	1.4	0.34	24066	3.30×10^{7}	0.38	48818	8.49	490753	2.17490×10^{40}	2.87
	1.5	0.25	19877	1.42×10^{7}	0.34	40340	7.23	410152	2.17486×10^{40}	2.99
G_4	1.1	0.02	2376	2.09×10^{4}	0.32	11109	80.03	3074734	1.19200×10^{52}	74.68
	1.2	0.01	1686	1.03×10^{4}	0.20	8511	22.24	1205320	1.19174×10^{52}	64.46
	1.3	0.01	1235	6.11×10^{3}	0.17	6935	11.51	692798	1.19170×10^{52}	113.37
	1.4	< 0.01	961	3.67×10^{3}	0.14	5808	8.30	529214	1.19164×10^{52}	99.81
	1.5	< 0.01	756	2.67×10^{3}	0.13	4930	5.30	348832	1.19153×10^{52}	23.70

Chapter 5

Planar Subgraph Enumeration

5.1 Introduction

In this chapter, we aim to extend types of subgraphs that can be deal with ZDDs and propose algorithms for planar subgraphs and more. Currently, FBS is known as the framework to construct a DD representing a set of constrained subgraphs. FBS can deal with fundamental constraints on subgraphs such as degrees and connectivity of vertices. Combining these constraints, one can construct DDs representing sets of paths, cycles, trees, and matchings, of a given graph. Recently, Kawahara et al. [72] proposed an extension of FBS, *colorful FBS* (CFBS). CFBS specifies subgraphs by "colored degrees" and "colorwise connectivity" of vertices. Using these constraints, one can construct a DD for more types of subgraphs than ordinary FBS. CFBS is utilized to construct DDs representing sets of chordal subgraphs and interval subgraphs, both of which are characterized by induced subgraphs.

Although many graph classes are characterized by induced subgraphs, another important characterization is by *topological-minor-embeddings* (TM-embeddings) [73]. For graphs G and H, a subgraph G' of G is a TM-embedding of H if G' is isomorphic to a *subdivision* of H. A subdivision of H is a graph obtained by replacing each edge in H with a path with at least one edge. Several important graph classes have *forbidden topological minor characterization* (FTM-characterization) [73]. For example, a graph is planar if and only if it has TM-embeddings of neither K_5 nor $K_{3,3}$ [73], where K_a is the complete graph with a vertices and $K_{b,c}$ is the complete bipartite graph with the two parts of b and c vertices. Other examples are

Table 5.1: Relationship between graph classes and forbidden topological minors. $K_4 - e$ is the graph obtained by removing an arbitrary edge from K_4.

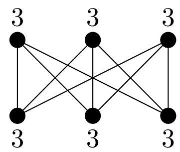
graph class	forbidden topological minors
planar graphs	$K_5, K_{3,3}$
outerplanar graphs	$K_4, K_{2,3}$
series-parallel graphs	K_4
cactus graphs	$K_4 - e$

shown in Table 5.1 (see [74] for details).

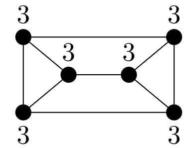
Our contribution In this chapter, when graphs G and H are given, we show a method to implicitly enumerate all TM-embeddings of H in G using CFBS. Combining the method with some additional DD operations, we can also implicitly enumerate subgraphs having FTM-characterizations, including planar, outerplanar, series-parallel, and cactus subgraphs. Our contributions are:

- Given graphs G and H, we show a method to implicitly enumerate all TM-embeddings of H in G using CFBS. We also analyze the complexity of the algorithm, which has not been done in [72]. (Section 5.3.1)
- We show more efficient methods when H is a graph used in FTM-characterizations of graph classes in Table 5.1, that is, complete graphs, complete bipartite graphs, and $K_4 - e$. (Section 5.3.2)
- Combining our method with DD operations, we show how to implicitly enumerate subgraphs having FTM-characterization, including planar, outerplanar, series-parallel, and cactus subgraphs. (Section 5.3.3)
- We evaluate our method by computational experiments. We apply our method to implicitly enumerating all planar subgraphs in a graph. The results show that our method runs up to five orders of magnitude faster than a naive backtracking-based method. We apply our method also for outerplanar, series-parallel, and cactus subgraphs. (Section 5.4)

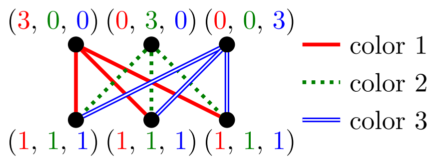
Our techniques We apply CFBS to TM-embeddings. Before TM-embedding enumeration, we explain how to apply CFBS to isomorphic subgraph enu-



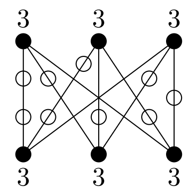
(a) $K_{3,3}$.



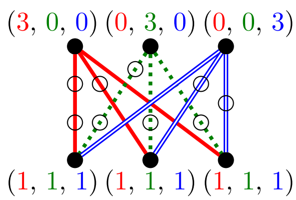
(b) Graph whose degree multiset is the same as $K_{3,3}$ but is not isomorphic to $K_{3,3}$.



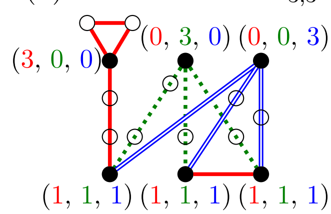
(c) 3-edge-colored graph that is isomorphic to $K_{3,3}$ when the colors are ignored.



(d) Subdivision of $K_{3,3}$.



(e) 3-edge-colored graph that is isomorphic to a subdivision of $K_{3,3}$ when the colors are ignored.



(f) 3-edge-colored graph that is not isomorphic to a subdivision of $K_{3,3}$ when the colors are ignored.

Figure 5.1: Graphs and edge-colored graphs. An integer (resp., tuple) next to a vertex stands for the degree (resp., colored degree) of the vertex. A white vertex stands for a subdividing vertex, whose degree is 2.

meration, which is a special case of [72]. Let us consider $K_{3,3}$ in Figure 5.1(a), which is used in FTM-characterization of planar graphs. $K_{3,3}$ has the degree multiset $\{3^6\}$, where 3^6 means that there are six vertices with degree 3. The graph in Figure 5.1(b) has the same degree multiset although it is not isomorphic to $K_{3,3}$. Thus, the degree multiset is not enough to characterize $K_{3,3}$ uniquely. Let us consider the edge-colored graph in Figure 5.1(c). For an edge-colored graph with k colors, we consider a *colored degree* of a vertex v, that is, a k-tuple of integers such that its i-th element is the number of color-i edges incident to v. The edge-colored graph in Figure 5.1(c) has the colored degree multiset $M = \{(3,0,0),(0,3,0),(0,0,3),(1,1,1)^3\}$. In fact, every edge-colored graph with colored degree multiset M is isomorphic to $K_{3,3}$ when the colors are ignored. Therefore, enumerating subgraphs of G that are isomorphic to $K_{3,3}$ is equivalent to finding all 3-colored subgraphs of G whose degree multisets equal M and then "decolorizing" them.

Now we consider TM-embedding enumeration. Recall that, for graphs G and H, a subgraph G' of G is a TM-embedding of H if G' is isomorphic to a subdivision of H. A graph H' is a subdivision of H if H' is obtained by replacing each edge of H with a path with at least one edge. Replacing an edge of H by a path may introduce a new vertex in H'. Such vertices are *subdividing vertices* and their degrees are 2. Figure 5.1(d) shows a subdivision of $K_{3,3}$. In the figure, white circles stand for subdividing vertices. A subdivision of $K_{3,3}$ has the degree multiset $\{3^6, 2^*\}$, where 2^* means that there are an arbitrary number of vertices with degree 2. However, a graph with the same degree multiset may have an isolated cycle, whose all vertices have degree 2. To forbid such a cycle, we need a constraint that the graph is connected. However, this is not enough because a subdivision of the graph in Figure 5.1(b) satisfies the same constraints.

Using colored constraints, we obtain the following necessary and sufficient condition. A graph is a subdivision of $K_{3,3}$ if and only if its edges can be colored by three colors so that

1. the edge-colored graph has a degree multiset $\{(3,0,0),(0,3,0),(0,0,3),$ $(1,1,1)^3,(2,0,0)^*,(0,2,0)^*,(0,0,2)^*\}$ and,

2. for each $i \in \{1,2,3\}$, the subgraph induced by the color-i edges is connected.

See Figure 5.1(e). Colorwise connectivity is needed because, if we impose only the whole connectivity, an edge-colored graph in Figure 5.1(f) is a counterexample. The above constraints can be handled by CFBS, and thus we can construct a DD representing the set of all TM-embeddings of $K_{3,3}$ in G using CFBS. In this chapter, we prove that a similar approach can be applied to every graph. Since the complexity of CFBS heavily depends on the number of colors used in the constraints, we discuss how to reduce the number of colors.

5.2 Preliminaries

5.2.1 Topological minors and characterization of graphs

In this subsection, we introduce topological minors and explain its application to characterization of graphs. *Subdividing* an edge $\{u, v\}$ of a graph H means removing the edge $\{u, v\}$ from H, introducing a new vertex w, and adding new edges $\{u, w\}$ and $\{v, w\}$. If a graph is obtained by subdividing each edge of H arbitrary times (possibly zero), it is a *subdivision* of H. Note that H itself is also a subdivision of H. A graph F is *homeomorphic* to a graph H if F is isomorphic to some subdivision of H.[1] If a graph F is homeomorphic to a graph H, the original vertices of H are the *branch vertices* of F and the other vertices are the *subdividing vertices*. Note that the degree of a branch vertex equals the original degree in H while the degree of a subdividing vertex is 2. (The degree of a branch vertex can be 2 when its original degree in H is 2.) For graphs G and H, H is a *topological minor* (TM) of G if G contains a subgraph homeomorphic to H. A subgraph G' of G is a *TM-embedding* of H in G if G' is homeomorphic to H. For families $\mathcal{G}$ and $\mathcal{H}$ of graphs, $\mathcal{G}$ is *forbidden-TM-characterized* (FTM-characterized) by $\mathcal{H}$ if, for any graphs $G \in \mathcal{G}$, $H \in \mathcal{H}$, and any subgraph G' of G, G' is not homeomorphic to H. For example, the family of planar graphs is FTM-characterized by $\{K_5, K_{3,3}\}$ [75]. The same characterization goes to several graph classes (Table 5.1).

5.2.2 Edge-colored graphs and tuples

A *c-(edge-)colored graph* $H^c = (H, f)$ is a pair of a graph $H = (V(H), E(H))$ and a function $f\colon E(H) \to [c]$. If $f(e) = i$ holds for an edge $e \in E(H)$ and an integer $i \in [c]$, e is a *color-i edge*. The *color-i degree* of $v \in V(H)$ in H^c is the number of color-i edges incident to v. The *colored degree* of $v \in V(H)$ in H^c is a c-tuple $(\delta_1, \ldots, \delta_c)$ of non-negative integers, where δ_i is the color-i degree of v. The *colored degree multiset* of H^c, which is denoted by $\mathrm{DS}(H^c)$, is the multiset of the colored degrees of all the vertices in H^c. The *color-i subgraph* of H^c is a graph induced by color-i edges of H^c. H is the *underlying*

[1] In another definition, F is homeomorphic to H if some subdivision of F is isomorphic to some subdivision of H. However, we allow subdividing only for H because H is "contracted enough" when it is a forbidden topological minor, that is, H does not contain redundant vertices with degree 2.

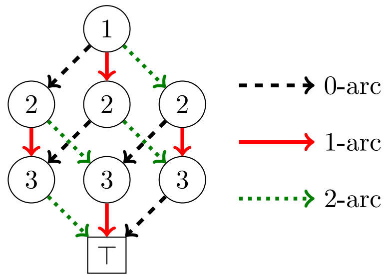


Figure 5.2: 3-DD. A square is a terminal node and circles are non-terminal nodes. An integer in a circle is the label of the node. For simplicity, we omit $\perp$ and the arcs pointing at it.

graph of H^c. A c-colored graph $F^c = (F, f')$ is a *c-colorized graph* of H if F is isomorphic to H. A *c-colored subgraph* of G is a c-colored graph whose underlying graph is isomorphic to a subgraph of G.

Since a colored degree is a tuple, we introduce some notations for tuples. For a c-tuple δ, δ_i denotes the i-th element of δ. For c-tuples δ and γ, we define $\delta \leq \gamma$ if, for all $i \in [c]$, $\delta_i \leq \gamma_i$ holds. When $\delta \leq \gamma$, we say that δ is *dominated* by γ. For a set s of c-colored degrees, $\mathcal{D}(s)$ denotes the set of tuples in $\mathbb{N}^c$ that are dominated by a tuple in s, that is, $\mathcal{D}(s) = \{\chi \in \mathbb{N}^c \mid \exists \delta \in s, \chi \leq \delta\}$.

5.2.3 $(c+1)$-decision diagram

We use a $(c{+}1)$-*decision diagram* ($(c{+}1)$-DD) [72] for implicit TM-embeddings enumeration. Let E be a finite set consisting of m elements $e_1, \ldots, e_m$. A $(c+1)$-*DD over E* is a rooted directed acyclic graph $\mathbf{Z}^{c+1} = (N, A, \ell)$, where N is the set of nodes, $A \subseteq \{(\alpha, \beta) \mid \alpha, \beta \in N, \alpha \neq \beta\}$ is the set of (directed) arcs, and $\ell \colon N \to [m+1]$ is a labeling function for nodes.[2] There is exactly one *root node* in N whose indegree is zero. In addition, N has exactly two *terminal nodes* $\perp$ and $\top$ whose outdegrees are zero. Nodes other than the terminal nodes are called *non-terminal nodes*. Each node α has the *label* $\ell(\alpha) \in [m+1]$. If α is a non-terminal node, its label is an integer in $[m]$. If α is a terminal node, its label is $m+1$. Each non-terminal node α has exactly $c{+}1$ arcs emanating from α. The arcs are called the 0-*arc*, 1-*arc*, ...,

[2]To avoid confusion, we use the terms "node" and "arc" for a $(c+1)$-DD and use "vertex" and "edge" for an input graph. In addition, we represent a node of a $(c+1)$-DD using the Greek alphabet (e.g., α, β) and a vertex of a graph using the English alphabet (e.g., u, v).

and *c-arc* of α. For an integer $j \in \{0, \ldots, c\}$, α_j denotes the node pointed at by the j-arc of α. For each non-terminal node α, $\ell(\alpha_j) = \ell(\alpha) + 1$ or $\ell(\alpha_j) = m + 1$ holds. That is, α_j is either a non-terminal node whose label is one more than α or a terminal node. It follows that $\mathbf{Z}^{c+1}$ is acyclic.

Given a graph $G = (V, E)$, we can represent a family of c-colored subgraphs of G by a $(c+1)$-DD in the following way. In a $(c+1)$-DD, we associate each path from the root node to $\top$ with a c-colored subgraph. For each path and $j \geq 1$, descending the j-arc of a non-terminal node with label i corresponds to assigning color j to e_i. Descending the 0-arc corresponds to excluding e_i from a subgraph. The set of all the paths from the root to $\top$ corresponds to the family of c-colored subgraphs represented by the $(c+1)$-DD. Figure 5.2 shows an example of a 3-DD over $\{e_1, e_2, e_3\}$. In the rest of the chapter, $\mathbf{Z}^{c+1}$ denotes a $(c+1)$-DD and $[\![\mathbf{Z}^{c+1}]\!]$ denotes the family of c-colored subgraphs represented by $\mathbf{Z}^{c+1}$. Note that, when $c = 1$, a 2-DD represents a family of ordinary subgraphs because there is a single color. When $c = 1$, we omit the superscript from $\mathbf{Z}^{c+1}$ and write $\mathbf{Z}$, that is, $\mathbf{Z}$ is a 2-DD.

5.2.4 Colorful frontier-based search (CFBS)

Colorful frontier-based search (CFBS) [72] is a framework of algorithms to construct a DD representing the set of constrained subgraphs. Although FBS constructs a 2-DD directly, CFBS constructs $(c+1)$-DD for some $c \geq 2$ first, and then obtain a 2-DD by *decolorizing* the $(c+1)$-DD. In this way, one can deal with a wider range of subgraphs with CFBS than FBS. Here, for a family $\mathcal{F}^c$ of c-colored subgraphs, its *decolorization* is the family $\mathcal{F}$ of subgraphs obtained by ignoring colors of edges of subgraphs in $\mathcal{F}^c$. For DDs, the decolorization of the $(c+1)$-DD representing $\mathcal{F}^c$ is the 2-DD representing $\mathcal{F}$. We can decolorize a DD by a recursive operation utilizing the recursive structure of the DD [72]. To construct a $(c+1)$-DD efficiently, CFBS uses dynamic programming. The *i-th frontier* W_i is the set of vertices incident to both the edges in $\{e_1, \ldots, e_{i-1}\}$ and $\{e_i, \ldots, e_m\}$. CFBS constructs a DD in a breadth-first manner from the root node and merges two nodes with the same label and states with respect to the frontier. See [72] for details.

5.3 Algorithms

Proofs are deferred to Section 5.6.1.

5.3.1 Implicit enumeration of TM-embeddings

Given graphs G and H, we show a method to construct the 2-DD $\mathbf{Z}(\widehat{H})$, where $\mathbf{Z}(\widehat{H})$ denotes the 2-DD representing the set of all TM-embeddings of H in G. In the following, $\mathcal{S}(H)$ denotes the family of subdivisions of H. Note that H itself is contained in $\mathcal{S}(H)$. Since a subdivision of H is obtained by replacing each edge of H by a path, a subdivision of H can be expressed as $E(H)$ paths with distinct colors. Therefore, we can characterize subdivisions of H using colored degrees and colorwise connectivity.

We define a *smoothed profile* of $\mathcal{S}(H)$ in the following way. Let Δ^c be the set of tuples $(\delta_1, \ldots, \delta_c)$ such that exactly one of δ_i's is zero and the others are two. In other words, δ^c consists of all tuples of the form $(0, \ldots, 0, 2, 0, \ldots, 0)$. Δ^c will be used for representing colored degrees of subdividing vertices. In the following, for a multiset M of c-colored degrees, $\mathcal{C}^*_M$ denotes a function from c-colored graphs to $\{0, 1\}$ such that $\mathcal{C}^*_M(F^c) = 1$ if and only if (a) $\mathrm{DS}(F^c)$ is obtained by adding an arbitrary number of elements (allowing duplication) of Δ^c to M and (b) the color-i subgraph for each i of F^c is connected. We say that a c-colored graph F^c *satisfies* $\mathcal{C}^*_M$ if $\mathcal{C}^*_M(F^c) = 1$.

Definition 5.1 (smoothed profile)**.** *Let c be a positive integer. A multiset M of c-colored degrees is a* smoothed profile *of $\mathcal{S}(H)$ if the following are equivalent:*

(a) A graph F belongs to $\mathcal{S}(H)$.

(b) There exists a c-colorized graph F^c of F that satisfies $\mathcal{C}^*_M$.

Our method of implicit TM-embedding enumeration is written as follows:

1. Find a smoothed profile M of $\mathcal{S}(H)$. Let c be the number of colors in M.

2. Construct $\mathbf{Z}^{c+1}(\mathcal{C}^*_M)$.

3. By decolorizing $\mathbf{Z}^{c+1}(\mathcal{C}^*_M)$, we obtain $\mathbf{Z}(\widehat{H})$.

For Step 1, we discuss how to find a smoothed profile in Theorems 5.2–5.5. Decolorization in Step 3 can be done in the same way as existing CFBS. To construct $\mathbf{Z}^{c+1}(\mathcal{C}^*_M)$ in Step 2, we use Kawahara et al.'s algorithm for the following problem: Given a multiset M of c-colored degrees, construct a DD $\mathbf{Z}^{c+1}(\mathcal{C}^*_M)$. For convenience, we represent a multiset M of c-colored degrees by a set s of c-colored degrees appearing in M and a function $f\colon s \to \mathbb{N}$ such that, for all $\delta \in s$, $f(\delta)$ equals the multiplicity of δ in M.

CFBS (FBS) constructs a DD in a breadth-first manner. To avoid creating redundant nodes, CFBS manages *configuration* of each node. The configuration is the information of subgraphs corresponding to a node. We define the configuration as a tuple $(\mathtt{deg}, \mathtt{dn}, \mathtt{comp}, \mathtt{done})$ of four arrays.[3] The definition of each array is as follows. The first array $\mathtt{deg}$ is an array of colored degrees of the vertices in the frontier. For a vertex v and an integer $j \in [c]$, $\mathtt{deg}[v]$ and $\mathtt{deg}[v][j]$ respectively denote the colored degree of v and the color-j degree of v. The second array $\mathtt{dn}$ is an array of the numbers of fixed vertices having each colored degree in s, where *fixed vertices* means the vertices that have left frontiers. For a colored degree $\delta \in s$, $\mathtt{dn}[\delta]$ denotes the number of fixed vertices having colored degree δ. The third array $\mathtt{comp}$ manages the connectivity of vertices in the frontier in the color-j subgraph for each j. For color $j \in [c]$, $\mathtt{comp}[j]$ is a partition of the frontier such that two vertices u, v are connected in the color-j subgraph if and only if they are contained in the same set in $\mathtt{comp}[j]$. The fourth array $\mathtt{done}$ holds Boolean values indicating which color-j subgraphs are finished. We say that, for each color $j \in [c]$, the color-j subgraph is *finished* when all the vertices in the connected color-j subgraph have left the frontier. For color $j \in [c]$, $\mathtt{done}[j] = \mathit{True}$ if and only if the color-j subgraph is finished.

We show pseudocode in Algorithms 5.2 and 6.2. In the following, we explain the algorithm and show the correctness at the same time. Algorithm 6.2 initializes the configurations of the root node and constructs a DD in a breadth-first manner, which is a usual technique of FBS [4]. When creating nodes α with label i, if there is a node α' that have the same label and configuration as α, the nodes are shared. The subroutine CHILD is a function whose inputs are a node α, its label i, and an integer $j \in \{0, \dots, c\}$ and output is a node α_j that will be the j-th child of α. It is shown in Algorithm 5.2. The procedure of Algorithm 5.2 is as follows. Let u_1, u_2 be the endpoints of e_i and create a node α_j (Lines 1–2). Initialize $(\mathtt{deg}', \mathtt{dn}', \mathtt{comp}', \mathtt{done}')$ by $(\mathtt{deg}, \mathtt{dn}, \mathtt{comp}, \mathtt{done})$ (Line 3). For each endpoint of e_i, we do the following (Lines 4–7). For $k \in [2]$, if u_k is not in the i-th frontier W_i, we initialize the colored degree of u_k by $(0, \dots, 0)$ (Line 6). For each color $j \in [c]$, if color-j subgraph is not finished, we initialize the connectivity of vertices in the color-j subgraph as u_k is the isolated vertex (Line 7).

[3] $\mathtt{comp}$ stands for *component*.

Algorithm 5.1: Constructing the $(c+1)$-DD

input : a set s of c-colored degrees and a function $f\colon s \to \mathbb{N}$
output : a $(c+1)$-DD

let $\texttt{deg} \leftarrow [\,][\,]$ (an empty associative array), $\texttt{dn}[\delta] \leftarrow 0$ for all $\delta \in s$, $\texttt{comp}[j'] \leftarrow \{\{\}\}$ for all $j \in [c]$, and $\texttt{done}[j] \leftarrow \mathit{False}$ for all $j \in [c]$
construct a root node ρ with a configuration $(\texttt{deg}, \texttt{dn}, \texttt{comp}, \texttt{done})$
let $N_1 \leftarrow \{\rho\}$, $N_i \leftarrow \emptyset$ for $i \in \{2, \ldots, m\}$ and $N_{m+1} \leftarrow \{\top, \bot\}$
for $i = 1, \ldots, m$ **do**
  for $\alpha \in N_i$ **do**
    for $j = 0, \ldots, c$ **do**
      $\alpha_j \leftarrow \textsc{Child}(\alpha, j)$
      if $\alpha_j \notin N_{i+1} \cup N_{m+1}$ **then**
        add a new node α_j with label $i+1$ to N_{i+1}
      let α_j be the j-child of α
return the $(c+1)$-DD consisting of nodes of $N_1, \ldots, N_{m+1}$

If $j > 0$, we assign color j to the edge e_i (Lines 8–15). If the color-j subgraph is finished, we cannot assign color j anymore, and thus we return $\bot$ (Line 9). For $k \in [2]$, we add one to the color-j degree of u_k (Line 10). If $\texttt{deg}'[u_k]$ is not in $\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)$, $\texttt{deg}'[u_k]$ cannot be a target colored degree in s or Δ^c, and thus we return $\bot$. Otherwise, we update the connectivity of the vertices including u_k in the color-j subgraph (Lines 13–15). For $k \in [2]$, let $C(u_k)$ be the set containing u_k in the current $\texttt{comp}'[j]$ (Line 13). If u_1 and u_2 are in different components in the color-j subgraph, they are merged by assigning color j to e_i, and thus we update $\texttt{comp}'[j]$ accordingly (Lines 14–15).

Next, we check if the color-j' subgraph is finished for each j'. For each $j' \in [c]$ such that the color-j' subgraph is not finished, we do the following (Lines 16–26). Let L be the set of components of the color-j' subgraph that have no vertices in W_{i+1} and S be the set of the other components (Line 17). If there are multiple components in L, the color-j subgraph will be disconnected, and thus we return $\bot$ (Line 18). Now consider the case where there are exactly one component in L (Lines 19–26). If S the color-j subgraph will be disconnected, and thus we return $\bot$ (Line 20). Otherwise, the color-j subgraph is finished and we update $\texttt{done}'[c]$ by *True* (Line 21). If color-j'' subgraphs for all $j'' \in [c]$ are finished, we check if the multiplicity of colored degrees. If the multiplicity is correct, we return $\top$; otherwise

$\perp$ (Lines 23–25). Since the components in L have no vertices in W_{i+1}, we remove the components in L from $\texttt{comp}'[j']$ (Line 26).

We also check the vertices leaving the frontier (Lines 27–34). For each $k \in [2]$, if u_k is not in W_{i+1}, we check the colored degree of u_k. If $\texttt{deg}'[u_k]$ is in s, we add one to $\texttt{dn}'[\texttt{deg}'[u_k]]$ (Line 30). If $\texttt{dn}'[\texttt{deg}'[u_k]]$ exceeds the target multiplicity $f(\texttt{deg}'[u_k])$, we return $\perp$ (Line 31). If $\texttt{deg}'[u_k]$ is in neither s nor Δ^c, we return $\perp$ (Line 32). Otherwise, since u_k is not in W_{i+1}, we remove u_k from the component of the color-j' subgraph for each $j' \in [c]$ (Lines 33–35).

Finally, if $i = m$, the constraints are not satisfied, and thus return $\perp$ (Line 35). Otherwise, we return a node α_j with a configuration $(\texttt{deg}', \texttt{dn}', \texttt{comp}', \texttt{done}')$ (Lines 36–37).

To assess the efficiency of algorithms based on CFBS, it is usual to analyze the *width* of the output DD [6]. The width of a DD is the maximum number of nodes with the same label. It is a measure of both the size of the DD and the time complexity to construct the DD. Recall that $w = \max_{i\in[m]} |W_i|$, where W_i is the i-th frontier.

Theorem 5.1. *Given a multiset M, let s be the set of c-colored degrees appearing in M and $f\colon s \to \mathbb{N}$ be the function such that, for all $\delta \in s$, $f(\delta)$ equals the multiplicity of δ in M. There is an algorithm to construct a DD $\mathbf{Z}^{c+1}(\mathcal{C}_s^*)$ with width*

$$2^{\mathcal{O}(cw\log w)}\left|\mathcal{D}\left(s\right) \cup \mathcal{D}\left(\Delta^c\right)\right|^w \prod_{\delta\in s}(f(\delta)+1). \tag{5.1}$$

Based on Theorem 5.1, we discuss the complexity for general H. First, we show that there is a smoothed profile for every graph H using $|E(H)|$ colors. Second, we show that the number of colors can be improved to $\tau(H)$, where $\tau(H)$ is the minimum size of vertex covers of H. Although the latter is better in most cases, we show both theorems for comparison.

Theorem 5.2. *Let H be a graph with at least two vertices and $H^{\tau(H)}$ be a $|E(H)$-colorized graph obtained by coloring the edges of H with distinct colors. Then, $M = \mathrm{DS}(H^{|E(H)|})$ is a smoothed profile of $\mathcal{S}(H)$. Moreover, there is an algorithm to construct a DD $\mathbf{Z}^{|E(H)|+1}(\mathcal{C}_M^*)$ with width*

$$2^{\mathcal{O}(|E(H)|w\log w)+|V(H)|}. \tag{5.2}$$

Theorem 5.3. *Let H be a graph with at least two vertices and $H^{\tau(H)}$ be a $\tau(H)$-colorized graph whose color-i subgraph for each i is isomorphic to a* *star and the set of the centers is a minimum vertex cover of H. Then, $M = \mathrm{DS}(H^{\tau(H)})$ is a smoothed profile of $\mathcal{S}(H)$. Moreover, there is an algorithm to construct a DD $\mathbf{Z}^{\tau(H)+1}(\mathcal{C}^*_M)$ with width*

$$2^{\mathcal{O}(\tau(H)w\log w)+|V(H)|}. \tag{5.3}$$

5.3.2 Constraints for forbidden topological minors

We derive specific smoothed profiles for the subdivisions of the graphs in the right column of Table 5.1: complete graphs, complete bipartite graphs, and $K_4 - e$. While the results for complete bipartite graphs and $K_4 - e$ follow directly from Theorem 5.3, we can reduce the number of colors by one for complete graphs. We show the result for $K_4 - e$ in Section 5.6.2. In the following, we discuss complete bipartite graphs first, which is easier than complete graphs.

Theorem 5.4. *Let a, b $(a \le b)$ be positive integers. The multiset $M_{a,b} = M^1_{a,b} \cup M^2_{a,b}$ consisting of a-colored degrees is a smoothed profile of $\mathcal{S}(K_{a,b})$, where*

$$M^1_{a,b} = \left\{ (\delta_1, \ldots, \delta_a)^1 \;\middle|\; \begin{array}{l} \exists i \in [a], \delta_i = b, \\ j \neq i \Rightarrow \delta_j = 0 \end{array} \right\}, \quad M^2_{a,b} = \left\{ (\underbrace{1, \ldots, 1}_{a})^b \right\}.$$

There is an algorithm to construct a DD $\mathbf{Z}^{a+1}(\mathcal{C}^*_{M_{a,b}})$ with width

$$2^{\mathcal{O}(aw\log w)}(2^a + ab)^w b. \tag{5.4}$$

Figure 5.1(e) shows a representation of a subdivision of $K_{3,3}$ based on Theorem 5.4.

Next, we consider the subdivisions of complete graphs. Since the size of a minimum vertex cover of K_a is $a - 1$, there exists a smoothed profile of $\mathcal{S}(K_a)$ with $a-1$ colors by Theorem 5.3. The smoothed profile is obtained by decomposing K_a into $K_{1,1}, K_{1,2}, \ldots,$ and $K_{1,a-1}$ and coloring the subgraphs with distinct colors. In this coloring, if we color $K_{1,2}$ with the same color as $K_{1,1}$, the obtained subgraph is K_3. We show that the colored degree multiset obtained from this coloring is also a smoothed profile of $\mathcal{S}(K_a)$.

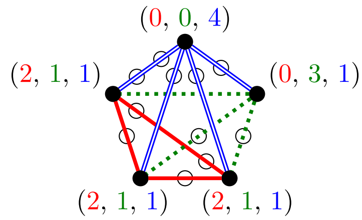


Figure 5.3: Representation of a subdivision of K_5. Filled and non-filled vertices represent branch and subdividing vertices, respectively. A tuple beside a vertex means the colored degree of the vertex.

Theorem 5.5. *Let $a \geq 3$ be an integer. The multiset $M_{a-2} = M^1_{a-2} \cup M^2_{a-2}$ consisting of $(a-2)$-colored degrees is a smoothed profile of $\mathcal{S}(K_a)$, where*

$$M^1_{a-2} = \left\{ (2, \underbrace{1, \ldots, 1}_{a-3})^3 \right\}, M^2_{a-2} = \left\{ (\delta_1, \ldots, \delta_{a-2})^1 \;\middle|\; \begin{array}{l} \exists i \in \{2, \ldots, a-2\}, \\ j < i \Rightarrow \delta_j = 0, \\ \delta_i = i+1, \\ j > i \Rightarrow \delta_j = 1 \end{array} \right\}.$$

There is an algorithm to construct a DD $\mathbf{Z}^{a-1}(\mathcal{C}^*_{M_a})$ with width

$$2^{\mathcal{O}(aw \log w)} \left(3 \cdot 2^{a-2} - a\right)^w . \tag{5.5}$$

Figure 5.3 shows a representation of a subdivision of K_5 based on Theorem 5.5.

5.3.3 Enumerating subgraphs having FTM-characterizations

We show how to implicitly enumerate subgraphs having FTM-characterization. We combine DD operations with our algorithm to implicitly enumerate TM-embeddings. Union [8] is a function whose inputs are two 2-DDs $\mathbf{Z}_1$ and $\mathbf{Z}_2$ and output is the 2-DD representing $[\![\mathbf{Z}_1]\!] \cup [\![\mathbf{Z}_2]\!]$. NonSupset [5] is a function whose input is a 2-DD $\mathbf{Z}$ over a finite set E and output is the 2-DD representing the family $\left\{A \subseteq 2^E \;\middle|\; \forall B \in [\![\mathbf{Z}]\!], A \not\supseteq B\right\}$. $\mathcal{G}(\widehat{H})$ denotes the set of subgraphs of G that are homeomorphic to H and $\mathbf{Z}(\widehat{H})$ denotes the 2-DD representing $\mathcal{G}(\widehat{H})$. The following algorithm constructs the 2-DD representing the set of subgraphs of G that is FTM-characterized by $\mathcal{H}$.

1. Initialize a 2-DD $\mathbf{Z}_{\text{subd}}$ by the 2-DD representing the empty set.
2. Choose an arbitrary graph H from $\mathcal{H}$ and remove it from $\mathcal{H}$.
3. Update $\mathbf{Z}_{\text{subd}}$ by $\textsc{Union}(\mathbf{Z}_{\text{subd}}, \mathbf{Z}(\widehat{H}))$.
4. If $\mathcal{H}$ is not empty, go back to Step 2. If empty, go on to Step 5.
5. We obtain the final 2-DD $\mathbf{Z}_{\text{ans}}$ by $\textsc{NonSupset}(\mathbf{Z}_{\text{subd}})$.

For example, let us consider the case where we want to implicitly enumerate all planar subgraphs of G. In this case, $\mathcal{H}$ is $\{K_5, K_{3,3}\}$. We construct $\mathbf{Z}(\widehat{K_5})$ and $\mathbf{Z}(\widehat{K_{3,3}})$ and take their union, which is $\mathbf{Z}_{\text{subd}}$. Now $\mathbf{Z}_{\text{subd}}$ represents the set of all subgraphs of G that are homeomorphic to K_5 or $K_{3,3}$. $\mathbf{Z}_{\text{ans}} = \textsc{NonSupset}(\mathbf{Z}_{\text{subd}})$ represents the family of all subgraphs of G that is FTM-characterized by $\mathcal{H} = \{K_5, K_{3,3}\}$. Therefore, $\mathbf{Z}_{\text{ans}}$ represents the family of all planar subgraphs of G. Other types of subgraphs such as outerplanar, series-parallel, and cactus subgraphs can be implicitly enumerated only by changing $\mathcal{H}$ according to Table 5.1.

5.4 Computational experiments

5.4.1 Settings

We conducted two experiments. First, we compared several methods to enumerate planar subgraphs (Section 5.4.2). Second, we applied our framework to enumerating all types of subgraphs in Table 5.1 (Section 5.4.3). For input graphs, we used complete graphs K_n and $3 \times b$ king graphs $X_{3,b}$ as synthetic data. $X_{3,b}$ is a graph obtained by, to the $3 \times b$ grid graph, adding diagonal edges in all the cycles of length four. As real data, we used Rome graph[4], which is often used in studies on graph drawing. The edge orderings are determined by breadth-first ordering for complete graphs and king graphs, and an existing method based on path-width optimization [64] for Rome graphs. We implemented all the code in C++ and compiled them by g++5.4.0 with -O3 option. To handle DDs, we used TdZdd [69] and SAPPORO_BDD inside Graphillion [63]. We used a machine with Intel Xeon E5-2637 v3 CPU and 1 TB RAM. For each case, we set the timeout to one day.

[4] `http://www.graphdrawing.org/data.html`

5.4.2 Comparing several methods to enumerate planar subgraphs

We compare the following three methods for planar subgraph enumeration.

- BACKTRACK: It explicitly enumerates subgraphs based on backtracking. The details are described in Section 5.6.3.
- DDEDGE: It implicitly enumerates subgraphs using DDs. It uses $|E(H)|$ colors based on Theorem 5.2. In other words, it uses ten colors for $\mathcal{S}(K_5)$ and nine colors for $\mathcal{S}(K_{3,3})$.
- DDVERTEX: It implicitly enumerates subgraphs using DDs. It uses $\tau(H)$ colors based on Theorems 5.3–5.5. In other words, it uses three colors both for $\mathcal{S}(K_5)$ and $\mathcal{S}(K_{3,3})$.

As a subroutine of BACKTRACK, we used a planarity test in C++ Boost[5]. For fairness, BACKTRACK does not output solutions but only counts the number of solutions. DDEDGE and DDVERTEX construct DDs representing the set of solutions. Once a DD is constructed, we can count the number of solutions in linear time to the number of nodes in the DD [5].

Table 5.2 shows the experimental results. In all the cases, all the methods output the same number of solutions. Among the three methods, DDVERTEX ran fastest except for K_6. BACKTRACK finished in a day only when the number of solutions is small (less than 10^9). Although DDEDGE solved more instances than BACKTRACK, it ran out of memory when the size of input or the number of solutions grows. In contrast, DDVERTEX succeeded even for such instances. For example, for $X_{3,4}$, DDVERTEX is 122,544 and 187 times faster than BACKTRACK and DDEDGE. In addition, for $X_{3,500}$, DDVERTEX succeeded in implicitly enumerating 7.95×10^{1349} planar subgraphs only in 405.04 seconds (less than seven minutes). These results demonstrate the outstanding efficiency of DDVERTEX.

5.4.3 Applying our framework to several types of subgraphs

In this subsection, we apply our framework to enumerating all types of subgraphs in Table 5.1. As stated in Section 5.3.3, to enumerate different types

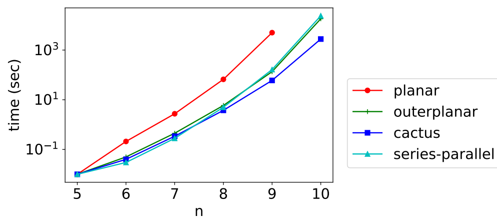
[5]`https://www.boost.org/doc/libs/1_71_0/libs/graph/doc/boyer_myrvold.html`

Table 5.2: Experimental results. Each column shows the name of graphs, the number of vertices and edges, the running time of the three methods (in seconds), and the number of planar subgraphs. "T/O" and "M/O" mean time out and memory out, respectively. "-" means all the methods failed. The number of solutions for K_{10} is from OEIS A066537, which is marked by '*'. We write the fastest time for each input graph in bold.

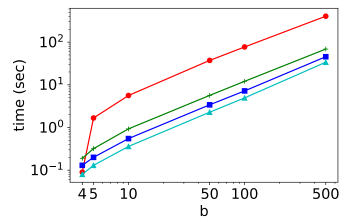
graph	$\lvert V\rvert$	$\lvert E\rvert$	BACKTRACK	DDEDGE	DDVERTEX	# solutions
K_5	5	10	$<$ **0.01**	0.14	$<$ **0.01**	1023
K_6	6	15	$<$ **0.01**	2.12	0.21	32071
K_7	7	21	28.28	35.02	**2.73**	1823707
K_8	8	28	3113.64	620.84	**66.34**	163947848
K_9	9	36	T/O	15623.11	**4694.41**	20402420291
K_{10}	10	45	T/O	T/O	T/O	*3209997749284
$X_{3,4}$	12	29	11029.38	16.83	**0.09**	5.33×10^{8}
$X_{3,5}$	15	38	T/O	53.93	**1.67**	2.70×10^{11}
$X_{3,10}$	30	83	T/O	665.65	**5.62**	8.93×10^{24}
$X_{3,50}$	150	443	T/O	M/O	**37.28**	1.29×10^{133}
$X_{3,100}$	300	893	T/O	M/O	**76.99**	2.03×10^{268}
$X_{3,500}$	1500	4493	T/O	M/O	**405.04**	7.95×10^{1349}
$X_{3,1000}$	3000	8993	T/O	M/O	M/O	-
G_1 (grafo1764.20)	20	25	792.16	1.09	**0.06**	3.35×10^{7}
G_2 (grafo1760.28)	28	39	T/O	96.81	**3.76**	5.49×10^{11}
G_3 (grafo10000.38)	38	52	T/O	787.98	**29.43**	4.50×10^{15}
G_4 (grafo10008.42)	42	61	T/O	38647.96	**668.15**	2.30×10^{18}
G_5 (grafo1378.46)	46	62	T/O	M/O	**796.48**	4.61×10^{18}
G_6 (grafo1395.61)	61	78	T/O	M/O	**11992.12**	3.02×10^{23}
G_7 (grafo5287.61)	61	88	T/O	M/O	M/O	-
G_8 (grafo9798.76)	76	91	T/O	M/O	**1709.64**	2.48×10^{27}
G_9 (grafo10006.98)	98	136	T/O	M/O	M/O	-

of subgraphs, it is enough to change $\mathcal{H}$, the set of forbidden topological minors.

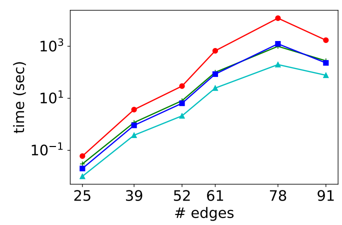
Figures 5.4(a)–5.4(c) show the results. We call an algorithm to enumerate planar subgraphs PLANAR, and so on. The results for king graphs (Figure 5.4(b)) are easiest to understand. We observe that PLANAR takes the most time because it uses three colors while the others use two colors. Among the algorithms using two colors, OUTERPLANAR is most time-consuming because it needs two topological minors. The reason why SERIES-PARALLEL runs faster than CACTUS is that K_4 has better "regularity" than $K_4 - e$, which makes the size of the output DD smaller. Indeed, for $X_{3,500}$, the size (number of nodes) of the DD constructed by SERIES-PARALLEL was



(a) Complete graph K_n.



(b) King graph $X_{3,b}$.



(c) Rome graph.

Figure 5.4: Results of applying our framework to enumerating several types of subgraphs. For each figure, its horizontal axis shows the size of an input graph and vertical one running time (in seconds). Note that all the vertical axes and the horizontal axis of Figure 5.4(b) are logarithmic.

4,582,909 while that by CACTUS was 7,289,225. The similar relation holds both for Figs. 5.4(a) and 5.4(c). For Rome graphs (Fig. 5.4(c)), the time for G_8 was smaller than G_6 although G_8 has more edges than G_6. It is because w of G_8 was smaller than that of G_6.

5.5 Conclusion

Given graphs G and H, we have shown a method to implicitly enumerate topological-minor-embeddings of H in G using decision diagrams. We also have shown a useful application of our method to enumerating subgraphs characterized by forbidden topological minors, including planar, outerplanar, series-parallel, and cactus subgraphs. Computational experiments show that our method can find all planar subgraphs up to 122,544 times faster than a naive backtracking-based method and could solve more problems than the backtracking-based method. We have applied our method also for outerplanar, series-parallel, and cactus subgraphs. Future work is extending our method from topological minors to general minors.

5.6 Appendix for this chapter

5.6.1 Proofs omitted from Section 5.3

In this section, we show appendix for this chapter. We show the proofs omitted from Section 5.3 in this subsection. For Theorems 5.2–5.5, there are two parts in the proofs: correctness of smoothed profiles and widths of the output DDs. The titles of the paragraphs indicate them.

Proof of Theorem 5.1

In the following, by "Line", we refer to lines in Algorithm 5.2. For a vertex v, the number of different values for $\texttt{deg}[v]$ is at most $|\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)|$ because we return $\perp$ if $\texttt{deg}[v]$ is not in $\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)$ (Line 12). Since every frontier has at most w vertices, the number of distinct sequences of values appearing in $\texttt{deg}$ is at most $|\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)|^w$. For a colored degree $\delta \in s$, the value of $\texttt{dn}[\delta]$ is in $\{0, \ldots, f(\delta)\}$ because we return $\perp$ if $\texttt{dn}[\delta]$ exceeds $f(\delta)$ (Line 31). Therefore, the number of different sequences of values appearing in $\texttt{dn}$ is at most $\prod_{\delta \in s}(f(\delta)+1)$. For each color $j' \in [c]$, $\texttt{comp}[j']$ maintains the partition of at most w vertices in the frontier. Since the number of partitions of w elements is $\mathcal{O}(w^w) = 2^{\mathcal{O}(w \log w)}$, the number of different values for $\texttt{comp}$ is $\left(2^{\mathcal{O}(w \log w)}\right)^c = 2^{\mathcal{O}(cw \log w)}$. For each color $j' \in [c]$, $\texttt{done}[j']$ is either *True* or *False*, and thus the number of different sequences of values appearing in $\texttt{done}$ is 2^c. Since we share nodes with the same label and configuration, the width of the constructed DD (the number of nodes with the same label) is at most the number of different configurations. The number is at most the product of the numbers of $\texttt{deg}$, $\texttt{dn}$, $\texttt{comp}$, and $\texttt{done}$. Therefore, the width of the constructed DD is at most

$$\begin{aligned}
&|\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)|^w \cdot \left(\prod_{\delta \in s}(f(\delta)+1)\right) \cdot 2^{\mathcal{O}(cw \log w)} \cdot 2^c \\
&= 2^{\mathcal{O}(cw \log w)} \, |\mathcal{D}(s) \cup \mathcal{D}(\Delta^c)|^w \prod_{\delta \in s}(f(\delta)+1).
\end{aligned}$$

Proof of Theorem 5.2

In the following, the proofs consist of two parts. We first show that the colored degree multiset is indeed the smoothed profile, and next derive the width of the DD.

Smoothed profile Let F be a graph that is homeomorphic to H. Observe that a subdivision of a graph H is obtained by replacing each of its edges by a path of length one or more. Let us color the paths with distinct colors. Each edge in F is associated, through the bijective mapping, with an edge in H. If an edge e in F is associated with an edge e' in H and e' has the color i in $H^{\tau(H)}$, we color e with i in F. The obtained $\tau(H)$-colorized graph of F has the

The colored degree multiset of the colorized graph, with the constraint "the color-i subgraph is connected for each i," suffices to ensure that the graph is homeomorphic to H. The color-i subgraph for each i must be a path because there are two vertices with degree 1 and an arbitrary number of vertices with degree 2 and is connected. In addition, two paths with different colors i and j share their endpoints if and only if there is a vertex whose color-i and color-j degrees are both 1. Therefore, for every graph H, we can identify $\mathcal{S}(H)$ by the constraints with $|E(H)|$ colors.

Width We derive Formula (5.2) from (5.1). Now, let $c = |E(H)|$ and $s = M$. All the tuples in s are dominated by $(1, \ldots, 1)$ because at most one edge with each color is incident to a vertex. Since no tuples in $\Delta^{|E(H)|}$ are dominated by $(1, \ldots, 1)$, $\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{|E(H)|}\right) \subseteq \mathcal{D}(\{(1, \ldots, 1)\}) \cup \mathcal{D}\left(\Delta^{|E(H)|}\right) = \mathcal{D}(\{(1, \ldots, 1)\}) \cup \Delta^{|E(H)|}$. Therefore, $\left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{|E(H)|}\right)\right| = |\mathcal{D}(\{(1, \ldots, 1)\})| + \left|\Delta^{|E(H)|}\right| = 2^{|E(H)|} + |E(H)|$. In addition, $\prod_{\delta \in s}(f(\delta)+1) \leq 2^{|V(H)|}$ because there are at most $|V(H)|$ different tuples in s. Based on the above discussion,

$$
\begin{aligned}
&2^{\mathcal{O}(cw \log w)} \left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{|E(H)|}\right)\right|^{w} \prod_{\delta \in s}(f(\delta)+1) \\
&\leq 2^{\mathcal{O}(|E(H)| w \log w)} \left(2^{|E(H)|} + |E(H)|\right)^{w} 2^{|V(H)|} \\
&= 2^{\mathcal{O}(|E(H)| w \log w) + |V(H)|} \left(2^{|E(H)|} + |E(H)|\right)^{w} \\
&= 2^{\mathcal{O}(|E(H)| w \log w) + |V(H)|} 2^{\mathcal{O}(|E(H)| w)} \\
&= 2^{\mathcal{O}(|E(H)| w \log w) + |V(H)|}.
\end{aligned}
$$

Proof of Theorem 5.3

We use the following lemma regarding characterization of isomorphic subgraphs by colored degrees. For a graph H, a subset $S \subseteq V(H)$ is a *vertex cover* of H if, for every edge $e \in E(H)$, at least one of its endpoints belongs to S. We denote the minimum size of vertex covers in H by $\tau(H)$. A *star* is a graph isomorphic to $K_{1,a}$ for some positive integer a.

Lemma 5.1 ([76]). *Let H be a graph and H^c be a c-colored graph of H such that, for every $i \in [c]$, the subgraph of H^c induced by color-i edges is isomorphic to a star. A graph F is isomorphic to H if and only if there exists a c-colored graph F^c of F such that $\mathrm{DS}(F^c) = \mathrm{DS}(H^c)$. It follows that, for every H, there is a profile using $\tau(H)$-colored degrees.*

Smoothed profile Let F be a graph that is homeomorphic to H. Each edge in F is associated, through the bijective mapping, with an edge in H. If an edge e in F is associated with an edge e' in H and e' has the color i in $H^{\tau(H)}$, we color e with i in F. The obtained $\tau(H)$-colorized graph of F satisfies (a) and (b) in Definition 5.1.

Let F be a graph and F^c be a c-colorized graph of F such that it satisfies (a) and (b) in Definition 5.1. First, we show that, in F^c, the color-i subgraph for each i is homeomorphic to a star. For each integer i in $[c]$, let M_i be the multiset of degrees of vertices in the color-i subgraph of F^c. By (a) in Definition 5.1, the multiset M_i satisfies one of the following:

1. $M_i = \{1^2, 2^y\}$ for an integer $y \geq 0$, or

2. $M_i = \{x^1, 1^x, 2^y\}$ for integers $x \geq 3$ and $y \geq 0$.

If $M_i = \{1^2, 2^y\}$ for an integer $y \geq 0$, the color-i subgraph of F^c is a path. Thus, it is homeomorphic to $K_{1,1}$. If $M_i = \{x^1, 1^x, 2^y\}$ for integers $x \geq 3$ and $y \geq 0$, the color-i subgraph of F^c is isomorphic to $K_{1,x}$. For each color $i \in [c]$, we process the color-i subgraph of F^c as follows:

- If $M_i = \{x^1, 1^x, 2^y\}$ for integers $x \geq 3$ and $y \geq 0$, we smooth all the vertices with degree 2, where *smoothing* a vertex v with degree 2 means removing vertex v and edges incident to it and connecting two vertices that were adjacent with v by a new edge.

- If $M_i = \{1^2, 2^y\}$ for an integer y, we check $\mathrm{DS}(H^{\tau(H)})$. If $\mathrm{DS}(H^{\tau(H)})$ contains a vertex with color-i degree 2 (note that there exists at most one such vertex in $\mathrm{DS}(H^{\tau(H)})$), we smooth all the vertices but one with degree 2. If not, we smooth all the vertices with degree 2.

Let I^c be the c-colorized graph obtained by the above procedure and I be its underlying graph. In I^c, the color-i subgraph for each i is isomorphic to a star and $\mathrm{DS}(I^c) = \mathrm{DS}(H^c)$. Therefore, by Lemma 5.1, the graph I is isomorphic to H. Since F is obtained by inserting smoothed vertices into edges in I, the graph F is a subdivision of I. It follows that F is homeomorphic to H.

Width We derive Formula (5.3) from (5.1). Now, let $c = \tau(H)$ and $s = M$. Since the color-i subgraph for each i of $H^{\tau(H)}$ is a star, every colored degree χ in $\mathrm{DS}(H^{\tau(H)})$ satisfies that there exists at most one color i such that χ_i exceeds 1. Therefore,

$$\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{\tau(H)}\right) \subseteq \left\{ (\chi_1, \ldots, \chi_{\tau(H)}) \in \mathbb{N}^{\tau(H)} \;\middle|\; \begin{array}{l} \exists i \in [\tau(H)], \\ \chi_i \leq |V(H)| - 1, \\ j \neq i \Rightarrow \chi_j \leq 1 \end{array} \right\} \tag{5.6}$$

Note that $|V(H)| - 1$ is an upper bound of the maximum degree of H. From (5.6), we obtain $\left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{\tau(H)}\right)\right| \leq 2^{\tau(H)}\tau(H)$. Combining the above with $\prod_{\delta \in s}(f(\delta) + 1) \leq 2^{|V(H)|}$, we obtain

$$\begin{aligned}
&2^{\mathcal{O}(cw \log w)} \left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{\tau(H)}\right)\right|^w \prod_{\delta \in s}(f(\delta) + 1) \\
&\leq 2^{\mathcal{O}(\tau(H) w \log w)} \left(2^{\tau(H)}\tau(H)\right)^w 2^{|V(H)|} \\
&= 2^{\mathcal{O}(\tau(H) w \log w) + |V(H)|} \left(2^{\tau(H)}\tau(H)\right)^w \\
&= 2^{\mathcal{O}(\tau(H) w \log w) + |V(H)|}\, 2^{\mathcal{O}(\tau(H) w)} \\
&= 2^{\mathcal{O}(\tau(H) w \log w) + |V(H)|}.
\end{aligned}$$

Proof of Theorem 5.4

Smoothed profile Let A and B be the parts of $K_{a,b}$ ($a \leq b$) consisting of a and b vertices, respectively. The set A is a minimum vertex cover of $K_{a,b}$. Let us decompose $K_{a,b}$ into a stars such that their centers are A and the leaves are B. We color the stars with distinct colors from $[a]$. In the colorized graph, the multisets of colored degrees of the vertices in A and B are $M_{a,b}^1$ and $M_{a,b}^2$, respectively. By Theorem 5.3, $M_{a,b} = M_{a,b}^1 \cup M_{a,b}^2$ is a smoothed profile of $\mathcal{S}(K_{a,b})$.

Width We derive Formula (5.4) from (5.1). Now $c = a$ and $s = M_{a,b}$. Since $\mathcal{D}(s) \cup \mathcal{D}(\Delta^a) = \mathcal{D}\left(M^1_{a,b}\right) \cup \mathcal{D}\left(M^2_{a,b}\right) \cup \mathcal{D}(\Delta^a) = \mathcal{D}\left(M^1_{a,b}\right) \cup \mathcal{D}\left(M^2_{a,b}\right)$, we obtain $|\mathcal{D}(s) \cup \mathcal{D}(\Delta^a)| \leq \left|\mathcal{D}\left(M^1_{a,b}\right)\right| + \left|\mathcal{D}\left(M^2_{a,b}\right)\right| = ab + 2^a = 2^a + ab$. Combining the above with $\prod_{\delta \in s}(f(\delta)+1) = 2^a(b+1)$,

$$\begin{aligned}
&2^{\mathcal{O}(cw \log w)} \left|\mathcal{D}(s) \cup \mathcal{D}(\Delta^a)\right|^w \prod_{\delta \in s}(f(\delta)+1) \\
&\leq 2^{\mathcal{O}(aw \log w)} \left(2^a + ab\right)^w 2^a(b+1) \\
&= 2^{\mathcal{O}(aw \log w)} \left(2^a + ab\right)^w b.
\end{aligned}$$

Proof of Theorem 5.5

Smoothed profile Let us decompose a subdivision F of K_a into subdivisions of $K_3, K_{1,3}, \ldots,$ and $K_{1,a-1}$ so that their centers and leaves are the branch vertices of F and color them with distinct colors. We denote the colorized graph by J. J is an $(a-2)$-colored graph and the multiset of colored degrees of the branch vertices in J is $M_{a-2} = M^1_{a-2} \cup M^2_{a-2}$, where M^1_{a-2} and M^2_{a-2} are the multisets of the colored degrees of (three arbitrarily chosen) branch vertices of a subdivision of K_3 and the centers of the subdivisions of the stars, respectively. Therefore, J satisfies the constraint $\mathcal{C}^*_M$, where M is M_{a-2}.

We show that the converse is true by induction. When $a = 3$, for a graph F, assume that there exists a $3 - 2 = 1$-colorized graph F^1 satisfying the constraint $\mathcal{C}^*_M$, where $M = M_1$. Since F_1 has an arbitrary number of vertices of degree 2 and is connected, F_1 is a cycle, that is, a subdivision of K_3. Next, for an integer $a \geq 3$, assume that "For a graph I, if there exists an $(a-2)$-colored graph I^{a-2} satisfying the constraint $\mathcal{C}^*_M$, where $M = M_{a-2}$, I belongs to $\mathcal{S}(K_a)$" is true. For a graph F, assume that there exists an $(a-1)$-colored graph F^{a-1} satisfying the constraint $\mathcal{C}^*_{M'}$, where $M' = M_{a-1}$. Among the colored degree multiset of F^{a-1}, the part of colors from 1 to $a-2$ is M_{a-2} plus an arbitrary number of elements of Δ^{a-2}. Therefore, by the assumption, the underlying graph of the colored graph from color 1 to $a-2$ in F^{a-1} forms a subdivision of K_a. The remaining part, the color-$(a-1)$ subgraph of F^{a-1}, has one vertex with degree a, a vertices with degree 1, and an arbitrary number of vertices with degree 2 and is connected. Therefore, the color-$(a-1)$ subgraph of F^{a-1} forms a subdivision of $K_{1,a}$. As for its center, its color-$(a-1)$ degree is a and the degrees of the other colors are 0. As for its leaves, their color-$(a-1)$ degrees are 1. If the colored degree of a leaf belongs to M^1_{a-1}, it can be a branch vertex of K_3. Otherwise, $\delta_i = i+1$ implies that it is the center of a subdivision of $K_{1,i+1}$. Therefore, F^{a-1} is a graph obtained by merging the branch vertices of a subdivision of K_a and the leaves of a subdivision of $K_{1,a}$. It follows that the underlying graph of F^{a-1} is homeomorphic to K_{a+1}.

Width We derive Formula (5.5) from (5.1). Now $c = a-2$ and $s = M_{a-2}$. For $\mathcal{D}(s) \cup \mathcal{D}(\Delta^{a-2})$, the following holds:

$$\begin{aligned}
\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{a-2}\right) &= \mathcal{D}\left(M^1_{a-2}\right) \cup \mathcal{D}\left(M^2_{a-2}\right) \cup \mathcal{D}\left(\Delta^{a-2}\right) \\
&= \mathcal{D}\left(\left\{(\underbrace{1,\ldots,1}_{a-2})\right\}\right) \cup \left\{(\delta_1,\ldots,\delta_{a-2}) \;\middle|\; \begin{array}{l} \exists i \in [a-2], \\ j < i \Rightarrow \delta_j = 0, \\ 2 \le \delta_i \le i+1, \\ j > i \Rightarrow \delta_j = 1 \end{array}\right\} \\
&= 2^{a-2} + \sum_{i=1}^{a-2}\left(i \cdot 2^{a-2-i}\right) \\
&= 2^{a-2} + \left(2^{a-1} - a\right) \\
&= 3 \cdot 2^{a-2} - a.
\end{aligned}$$

In addition, $\prod_{\delta \in s}(f(\delta)+1) = (3+1)\cdot(1+1)^{a-3} = 2^{a-1}$ holds. Thus, we obtain

$$\begin{aligned}
&2^{\mathcal{O}(cw\log w)} \left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^{a-2}\right)\right|^w \prod_{\delta\in s}(f(\delta)+1) \\
&\le 2^{\mathcal{O}((a-2)w\log w)} \left(3\cdot 2^{a-2} - a\right)^w 2^{a-1} \\
&= 2^{\mathcal{O}(aw\log w)} \left(3\cdot 2^{a-2} - a\right)^w.
\end{aligned}$$

5.6.2 Smoothed profile of $\mathcal{S}(K_4 - e)$

Recall that $K_4 - e$ is the graph obtained by removing an arbitrary edge from K_4.

Theorem 5.6. *A multiset $M = \{(3,0),(1,2),(1,1)^2\}$ of 2-colored degrees is a smoothed profile of $\mathcal{S}(K_4 - e)$. There is an algorithm to construct a DD representing $\mathbf{Z}^3(\mathcal{C}^*_M)$ with width $2^{\mathcal{O}(w\log w)}$.*

Proof Let $K_4 - e = (\{a, b, c, d\}, \{\{a, b\}, \{a, c\}, \{a, d\}, \{c, b\}, \{c, d\}\})$. The set $\{a, c\}$ of vertices is a minimum vertex cover of the graph. We color the star with edges $\{a, b\}$, $\{a, c\}$, and $\{a, d\}$ by color 1 and that with $\{c, b\}$ and $\{c, d\}$ by color 2. The colored degree multiset of the colorized graph is M. By Theorem 5.3, M is a smoothed profile of $\mathcal{S}(K_4 - e)$. We obtain the width as follows:

$$
\begin{aligned}
&2^{\mathcal{O}(cw \log w)} \left|\mathcal{D}(s) \cup \mathcal{D}\left(\Delta^2\right)\right|^w \prod_{\delta \in s} (f(\delta) + 1) \\
&= 2^{\mathcal{O}(2w \log w)} \cdot 8^w \cdot (2 \cdot 2 \cdot 3) \\
&= 2^{\mathcal{O}(w \log w)}.
\end{aligned}
$$

5.6.3 Details of backtracking-based method

In this subsection, we show the details of an algorithm to explicitly enumerate planar subgraphs based on backtracking, which we used in Section 5.4. Pseudocode is given in Algorithm 5.3. Given a graph G, we first call $\textsc{Main}(G)$ (Line 1). It calls a subfunction $\textsc{Rec}$. Its inputs are a graph G, a subset of edges S that forms a planar subgraph of G, and the index of the edge that should be processed next. If $i = |E(G)| + 1$, we can add no edges, and thus we output S (Line 3). Otherwise, we guess whether e_i is adopted for a solution of not. We always call $\textsc{Rec}(G, S, i + 1)$ because $G[S]$ is planar. In contrast, we call $\textsc{Rec}(G, S \cup \{e_i\}, i + 1)$ only if $G[S \cup \{e_i\}]$ is planar. Since planar graphs are closed under taking subgraphs, the algorithm correctly outputs all the planar subgraphs. The algorithm runs a planarity test $\mathcal{O}(|E(G)|)$ times for each solution. Since a planarity test can be done in $\mathcal{O}(|V(G)|)$ time [77], the time complexity of the algorithm is $\mathcal{O}(N \cdot |E(G)| \cdot |V(G)|)$, where N is the number of solutions.

Algorithm 5.2: CHILD(α, i, x)

input : node α with configuration (deg, dn, comp, done) and a child number j

output : a node α_j that will be the j-th child of α

```
let {u_1, u_2} ← e_i
generate α_j
let deg′ ← deg, dn′ ← dn, comp′ ← comp, and done′ ← done
for k ∈ [2] do
    if u_k ∉ W_i then
        deg′[u_k] ← (0, . . . , 0)
        for j′ ∈ [c] such that done′[j′] = False do
            comp[j′] ← comp[j′] ∪ {{u_k}}
if j > 0 then
    if done′[j] = True then return ⊥
    for k ∈ [2] do
        deg′[u_k][j] ← deg′[u_k][j] + 1
        if deg′[u_k] ∉ D(s) ∪ D(Δ^c) then return ⊥
    for each k ∈ [2], let C(u_k) be the set containing u_k in the current
      comp′[j]
    if C(u_1) ≠ C(u_2) then
        comp′[j] ← (comp′[j] \ {C(u_1), C(u_2)}) ∪ {C(u_1) ∪ C(u_2)}
for j′ ∈ [c] such that done′[j′] = False do
    let L ← {C ∈ comp′[j′] | C ∩ W_{i+1} = ∅} and S ← comp′[j] \ L
    if |L| > 1 then return ⊥
    else if |L| = 1 then
        if |S| > 0 then return ⊥
        else
            done′[j′] ← True
            if for all j″ ∈ [c], done′[j″] = True then
                if for all δ ∈ s, dn′[δ] = f(δ) then return ⊤
                else return ⊥
            comp′[j′] ← comp′[j′] \ L
    for k ∈ [2] do
        if u_k ∉ W_{i+1} then
            if deg′[u_k] ∈ s then
                dn′[deg′[u_k]] ← dn′[deg′[u_k]] + 1
                if dn′[deg′[u_k]] > s(deg′[u_k]) then return ⊥
            else if deg′[u_k] ∉ Δ^c then return ⊥
            for j′ ∈ [c] such that done′[j′] = False do
                let C(u_k) be the set containing u_k in the current comp′[j′]
                comp′[j′] ← (comp′[j′] \ {C(u_k)}) ∪ ({C(u_k) \ {u_k}})
    if i = m then return ⊥
    let (deg′, dn′, comp′, done′) be the configuration of α_j
    return α_j
```

Algorithm 5.3: Enumerating planar subgraphs based on backtracking

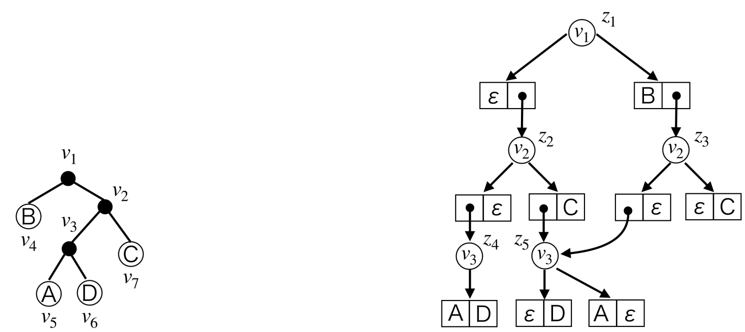
input : a graph G
output : all planar subgraphs in G
def MAIN(G)**:**
    REC($G, \emptyset, 1$)
def REC(G, S, i)**:**
    if $i = |E(G)| + 1$ **then** output S
    else
        REC($G, S, i + 1$)
        if $G[S \cup \{e_i\}]$ *is planar* **then**
            REC($G, S \cup \{e_i\}, i + 1$);

Chapter 6

Frontier-Based Search for ZSDDs

6.1 Introduction

Until the previous chapter, we have been used ZDDs for implicit enumeration. Although ZDDs can represent set families in a compact way, the size of a ZDD can be prohibitively large, which leads to the limitation of the application of ZDDs to relatively small graphs. Recently, Zero-suppressed Sentential Decision Diagrams (ZSDDs) [49] have been proposed as different representations of set families. Since ZSDDs are generalizations of ZDDs, ZSDDs are at least as compact as ZDDs. In theory, there exist set families that have polynomial ZSDD sizes but exponential ZDD sizes [65]. In addition, ZSDDs inherit some poly-time queries of ZDDs: counting, random sampling, and Apply operations. Thus, a natural question is: Can we design top-down construction algorithms for ZSDDs representing sets of subgraphs? The question is partially answered in an affirmative way by Nishino et al. [78]. They proposed top-down construction algorithms for ZSDDs representing sets of specific types of subgraphs: matchings and paths. The sizes of constructed ZSDDs by their algorithms are bounded by the *branch-width* of the input graph [78], while those of ZDDs are bounded by the *path-width* [64]. Since the branch-width of a graph never exceeds the path-width [79], ZSDDs have tighter upper bounds than ZDDs. The efficiency of their algorithms was confirmed in experiments. Despite such striking results, their algorithms are specific to matchings and paths.



(a) A vtree.

(b) A ZSDD.

Figure 6.1: A vtree and a ZSDD that respects the vtree.

In this chapter, we propose a novel framework of top-down construction algorithms for ZSDDs. To design a top-down construction algorithm using our framework, one only has to prove a recursive formula for the desired set of subgraphs. Using the recursive formula, we can theoretically show the correctness and the complexity of the algorithm, which was difficult with the existing method. We apply our framework to the three fundamental constraints used in ZDDs: the number of edges, degrees of vertices, and connectivity of vertices. We show that the sizes of constructed ZSDDs are bounded by the branch-width of the input graph, not only for matchings and paths. Experiments show that proposed methods can construct ZSDDs faster than ZDDs and that the constructed ZSDDs are smaller than ZDDs representing the same sets of subgraphs.

6.2 Preliminaries

6.2.1 $(\mathbf{X}, \mathbf{Y})$-partition and vtree

To introduce ZSDDs, we define $(\mathbf{X}, \mathbf{Y})$-partition and vtree in this subsection.

Definition 6.1. *Let f be a set family, and $\mathbf{X}, \mathbf{Y}$ be a partition of the universe of f. Set family f can be written as*

$$f = \bigcup_{i=1}^{h} [p_i \sqcup s_i], \tag{6.1}$$

*where p_i and s_i are the set families whose universes are $\mathbf{X}$ and $\mathbf{Y}$, respectively. The equation is an $(\mathbf{X}, \mathbf{Y})$-*decomposition*. We call $p_1, \ldots, p_h$* primes *and $s_1, \ldots, s_h$* subs*. If the primes are* exclusive *($p_i \cap p_j = \emptyset$ for all $i \neq j$), the decomposition is an $(\mathbf{X}, \mathbf{Y})$-*partition*.* [1]

Example 6.1. *Let f_1 be the family of subsets of $U_1 = \{A, B, C, D\}$ that contain exactly two elements. It follows that $f_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}$. For $\mathbf{X}_1 = \{B\}$ and $\mathbf{Y}_1 = \{A, C, D\}$, an $(\mathbf{X}_1, \mathbf{Y}_1)$-partition of f_1 is*

$$f_1 = [\underbrace{\{\emptyset\}}_{\text{prime}} \sqcup \underbrace{f_2^1}_{\text{sub}}] \cup [\underbrace{\{\{B\}\}}_{\text{prime}} \sqcup \underbrace{f_2^2}_{\text{sub}}], \tag{6.2}$$

where $f_2^1 = \{\{A, C\}, \{A, D\}, \{C, D\}\}$ and $f_2^2 = \{\{A\}, \{C\}, \{D\}\}$.

The universe of f_2^1 and f_2^2 is $U_2 = \{A, C, D\}$. For $\mathbf{X}_2 = \{A, D\}$ and $\mathbf{Y}_2 = \{C\}$, an $(\mathbf{X}_2, \mathbf{Y}_2)$-partition of f_2^1 is

$$f_2^1 = [\underbrace{\{\{A, D\}\}}_{\text{prime}} \sqcup \underbrace{\{\emptyset\}}_{\text{sub}}] \cup [\underbrace{\{\{A\}, \{D\}\}}_{\text{prime}} \sqcup \underbrace{\{\{C\}\}}_{\text{sub}}]. \tag{6.3}$$

A ZSDD represents a set family by recursively applying $(\mathbf{X}, \mathbf{Y})$-partitions to decompose the family into sub-families, where the order of partitions is determined by a *vtree*. A vtree is a rooted, ordered, and full binary tree whose leaves correspond to elements of the universe. Fig. 6.1(a) shows an example. Symbols appearing in leaves represent corresponding elements, and symbols beside nodes represent their names. Each internal node represents a partition of the universe into two subsets: elements appearing in the left and right subtrees. We denote the left and right children of node v by v^l and v^r, respectively. In the figure, root node v_1 represents the $(\mathbf{X}_1, \mathbf{Y}_1)$-partition of the universe $U_1 = \{A, B, C, D\}$ where $\mathbf{X}_1 = \{B\}$ and $\mathbf{Y}_1 = \{A, C, D\}$. Similarly, node v_2 represents the $(\mathbf{X}_2, \mathbf{Y}_2)$-partition of the universe $U_2 = \{A, C, D\}$ where $\mathbf{X}_2 = \{A, D\}$ and $\mathbf{Y}_2 = \{C\}$. To avoid confusion, we call vtree nodes *vnodes*, ZSDD nodes *znodes*, and graph nodes *vertices*. We represent them as v_i, z_i, and u_i.

[1] In [49], an $(\mathbf{X}, \mathbf{Y})$-decomposition is called an $(\mathbf{X}, \mathbf{Y})$-partition if the primes are exclusive and *consistent* ($p_i \neq \emptyset$ for all i). For simplicity, we do not require consistency for $(\mathbf{X}, \mathbf{Y})$-partitions. If we construct a ZSDD without consistency, we can make their primes consistent in linear time to the ZSDD size [78].

6.2.2 Zero-suppressed Sentential Decision Diagrams

A ZSDD is recursively defined as follows. ZSDD α *respects* vnode v if the order of $(\mathbf{X}, \mathbf{Y})$-partitions in α follows the vtree whose root is v. $\langle \alpha \rangle$ denotes the set family that α represents.

Definition 6.2. *α is a ZSDD that respects vnode v if and only if:*

- *$\alpha = \varepsilon$ or $\alpha = \bot$. (Semantics: $\langle \varepsilon \rangle = \{\emptyset\}$ and $\langle \bot \rangle = \emptyset$.)*
- *$\alpha = X$ or $\alpha = \pm X$ and v is a leaf with element X. (Semantics: $\langle X \rangle = \{\{X\}\}$ and $\langle \pm X \rangle = \{\{X\}, \emptyset\}$.)*
- *$\alpha = \{(p_1, s_1), \ldots, (p_h, s_h)\}$, v is internal, $p_1, \ldots, p_h$ are ZSDDs that respect a vnode in the subtree whose root is v^l, $s_1, \ldots, s_h$ are ZSDDs that respect a vnode in the subtree whose root is v^r, and $\langle p_1 \rangle, \ldots, \langle p_h \rangle$ are exclusive. (Semantics: $\langle \alpha \rangle = \bigcup_{i=1}^{h} [\langle p_i \rangle \sqcup \langle s_i \rangle]$.)*

If a ZSDD is either $\varepsilon, \bot, X$, or $\pm X$, it is a *terminal*. Otherwise, it is a *decomposition*. Fig. 6.1(b) shows an example ZSDD that represents set family f_1 in Example 6.1 and respects the vtree in Fig. 6.1(a). A circle node and its child rectangle nodes represent an $(\mathbf{X}, \mathbf{Y})$-partition. The symbol in a circle node indicates the vnode that the decomposition respects. A pair of rectangle nodes represent a prime-sub pair in an $(\mathbf{X}, \mathbf{Y})$-partition where the left and right are prime p and sub s, respectively. Every p and s is either a terminal ZSDD or a pointer to a decomposition ZSDD. Circle nodes are *decomposition znodes*, and rectangle nodes are *element znodes*. For example, znodes z_1 and z_2 represent the $(\mathbf{X}, \mathbf{Y})$-partitions in Eqs. (6.2) and (6.3), respectively. The *size* of a ZSDD is the sum of the sizes of $(\mathbf{X}, \mathbf{Y})$-partitions in the ZSDD. The size of the ZSDD in Fig. 6.1(b) is 9. [2]

6.3 A novel framework of top-down ZSDD construction

We present a novel framework of top-down ZSDD construction. Our framework is partially identical to that of Nishino et al.'s [78], but we modify it so

[2]The size of a ZDD is defined as the number of nodes. [8] This is because, every node of a ZDD has exactly two children. In contrast, nodes of a ZSDD may have different number of children, and thus the size of a ZSDD is defined as the number of arcs.

Algorithm 6.1: A top-down construction algorithm

input : A graph $G = (V, E)$ and the root vtree node v
output : A ZSDD representing a set of subgraphs of G

$Z[v] \leftarrow \mathsf{rootState}()$
$\mathsf{construct}(v, Z)$
$Z \leftarrow \mathsf{reduce}(Z)$
return Z

Algorithm 6.2: $\mathsf{construct}(v, Z)$

for $z \in Z[v]$ **do**
  $\texttt{elems} \leftarrow \emptyset$
  for $(m^l, m^r) \in \mathsf{decomp}(v, z)$ **do**
    for $\circ \in \{l, r\}$ **do**
      if v° *is a leaf vnode* **then** $z^\circ \leftarrow \mathsf{terminal}(v^\circ, m^\circ)$
      else $z^\circ \leftarrow \mathsf{unique}(v^\circ, m^\circ, Z)$
    $\texttt{elems} \leftarrow \texttt{elems} \cup \left\{(z^l, z^r)\right\}$
  Set `elems` as the child znodes of z
  for $\circ \in \{l, r\}$ **do**
    if v° *is an internal vnode* **then** $\mathsf{construct}(v^\circ, Z)$

that we can design algorithms easily for several constraints. Algorithm 6.1 shows the framework. The algorithm takes graph G and the root vnode as its inputs and returns a ZSDD representing a set of subgraphs of G. $Z[v]$ stores a set of decomposition znodes that respect vnode v. Since a ZSDD is represented as a set of decomposition znodes, the set of $Z[v]$'s for all internal vnodes v can be seen as a ZSDD. The algorithm first calls $\mathsf{rootState}()$, which returns the root znode. The procedure depends on the types of subgraphs. The algorithm next calls $\mathsf{construct}(v, Z)$, which recursively construct child znodes of znodes respecting v. If we naively construct znodes, the number of child znodes grows exponentially. We thus merge *equivalent* znodes during the construction of a ZSDD. Here, two znodes are equivalent if they respect the same vnode and represent the same family of sets. To detect equivalent znodes efficiently, we attach a *label* to each znode. The labels must be defined depending on the types of subgraphs so that two znodes are equivalent if they respect the same vnode and have the same label. We explain how to design labels in Section 6.4. The constructed ZSDD may have redundant znodes. Function $\mathsf{reduce}(Z)$ deletes such znodes.

Algorithm 6.2 shows function $\mathsf{construct}(v, Z)$. The function is called only for internal vnodes. In [78], the procedure of $\mathsf{construct}(v, Z)$ was designed depending on whether v^l is a leaf or not. Instead, we treat all internal vnodes in the same way, which makes it easy to design algorithms for several constraints. For each znode z in $Z[v]$, the function calculates the prime-sub pairs corresponding to z. We first initialize the set of prime-sub pairs `elems` to the empty set (Line 2). Function $\mathsf{decomp}(v, z)$ receives vnode v and znode z that respects v, and returns the set of pairs of labels corresponding to the prime-sub pairs (Line 3). For each $\circ \in \{l, r\}$, if v° is a leaf vnode, we set znode z° to a terminal (Line 5). Function $\mathsf{terminal}(v, m)$ receives leaf vnode v and label m, and returns an appropriate terminal depending on the types of subgraphs. If v° is an internal vnode, we call $\mathsf{unique}(v, m, Z)$ (Line 6). The function receives vnode v and label m, and checks whether $Z[v]$ contains a znode with label m. If such a znode exists, the function returns its address. Otherwise, the function creates a new znode that respects v and has label m, stores it into $Z[v]$, and returns its address. We add the prime-sub pair (z^l, z^r) into `elems` (Line 7). After generating all the prime-sub pairs, we set `elems` as the child znodes of z (Line 8). Finally, for each $\circ \in \{l, r\}$ such that v° is an internal vnode, we call $\mathsf{construct}(v^\circ, Z)$ to recursively construct sub-ZSDDs (Lines 9–10).

The functions $\mathsf{reduce}(Z)$ and $\mathsf{unique}(v, m, Z)$ can be designed regardless of the types of subgraphs [78]. In contrast, the definition of labels and the procedures of $\mathsf{rootState}()$, $\mathsf{terminal}(v, m)$, and $\mathsf{decomp}(v, z)$ heavily depend on the types of subgraphs. To easily design them for several constraints, we relate a recursive formula for the desired set of subgraphs to top-down ZSDD construction. Intuitively, in our framework, internal vnodes correspond to recursion steps, while leaf vnodes correspond to base cases. Therefore, we only have to prove a recursive formula for the desired set of subgraphs. The recursive formula directly leads to the definition of labels and the procedures of subroutines. We can also show the correctness of the algorithm and the bound of the constructed ZSDD size from the recursive formula.

6.4 Subroutines for several constraints

We apply our framework to three fundamental constraints: the number of edges, degrees of vertices, and connectivity of vertices. By combining these constraints, we can specify several types of subgraphs. For each constraint, we show a recursive formula for the set of subgraphs satisfying the constraint. Using the recursive formula, we derive subroutines and bound the sizes of constructed ZSDDs.

6.4.1 Cardinality

Given graph $G = (V, E)$, vtree T whose leaves are labeled by the elements of E, and non-negative integer k^*, we construct a ZSDD that represents the family of sets with exactly k^* elements. We can also construct a ZSDD that represents the family of sets with at most or at least k^* elements. In the following, we focus on the "exactly k^*" constraint. For vnode v, let $E(v) \subseteq E$ be the set of graph edges that correspond to the leaf vnodes of the sub-vtree whose root is v. For vnode v and non-negative integer k, let $f(v, k)$ be the family of subsets of $E(v)$ with k elements, that is, $f(v, k) = \{S \mid S \subseteq E(v), |S| = k\}$. The desired family is $f(v^{\mathrm{root}}, k^*)$, where v^{root} is the root vnode of T. For leaf vnode v, $\ell(v)$ denotes the element corresponding to v. We show a recursive formula for $f(v, k)$.

Lemma 6.1. *Let v be a vnode, and k be a non-negative integer. If v is a leaf vnode, then the following hold:*

$$f(v,k) = \begin{cases} \{\emptyset\} & (k = 0), \\ \{\{\ell(v)\}\} & (k = 1), \\ \emptyset & (\text{otherwise}). \end{cases} \tag{6.4}$$

If v is internal, the following is an $(E(v^l), E(v^r))$-partition:

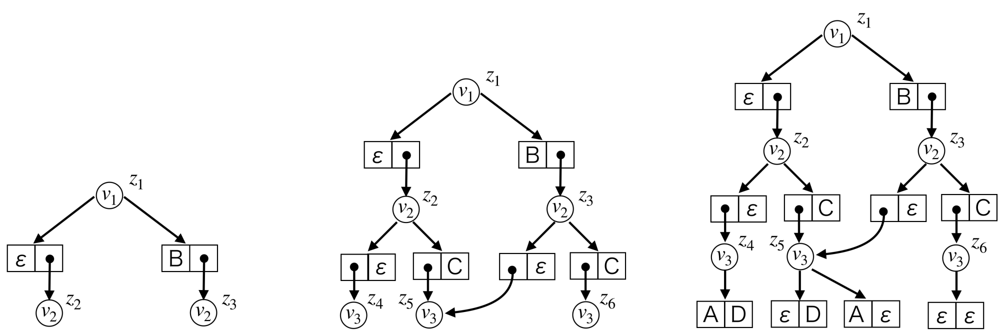
$$f(v,k) = \bigcup_{i=0}^{k} \left[f(v^l, i) \sqcup f(v^r, k-i) \right]. \tag{6.5}$$

Using the recursive formula, we can design the subroutines of the framework. In the following, we show the subroutines and proof the correctness at the same time. We use non-negative integers as znode labels. For znode z that respects vnode v, the label of z indicates the number of elements that should be adopted from $E(v)$. Function $\mathsf{rootState}()$ returns the root znode with label k^*, since the desired family is $f(v^{\mathrm{root}}, k^*)$. Algorithm 6.3 shows the subroutines $\mathsf{terminal}(v, k)$ and $\mathsf{decomp}(v, z)$. $\mathsf{terminal}(v, k)$ is obtained from Eq. (6.4). If $k = 0$, it returns ε since $\langle \varepsilon \rangle = \{\emptyset\}$ (Line 1). If $k = 1$, it

Algorithm 6.3: Subroutines for the cardinality constraint

Function : terminal(v, k)
1 **if** $k = 0$ **then return** ε
2 **else if** $k = 1$ **then return** $\ell(v)$
3 **else return** $\perp$

Function : decomp(v, z)
4 `elems` $\leftarrow \emptyset$
5 Let k be the label of z
6 **for** $i \in [0, k]$ **do**
7   `elems` $\leftarrow$ `elems` $\cup \{(i, k-i)\}$
8 **return** `elems`

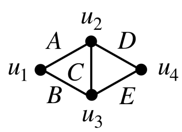


(a) After construct(v_1, Z). (b) After construct(v_2, Z). (c) After construct(v_3, Z).

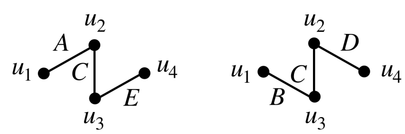
Figure 6.2: Intermediate ZSDDs for the cardinality constraint.

returns $\ell(v)$ since $\langle \ell(v) \rangle = \{\{\ell(v)\}\}$ (Line 2). Otherwise, it returns $\perp$ since $\langle \perp \rangle = \emptyset$ (Line 3). Similarly, decomp(v, z) is obtained from Eq. (6.5). The function initializes `elems` to the empty set (Line 4). Let k be the label of z (Line 5). If the prime has label $0 \leq i \leq k$, then the sub has label $k - i$. Thus, we add the pair $(i, k-i)$ to `elems` (Lines 6–7). Finally, we return `elems` (Line 8). The correctness of the algorithm directly follows from the correctness of Lemma 6.1.

Example 6.2. *Let us construct a ZSDD that represents the family of subsets of $\{A, B, C, D\}$ with exactly two elements. We use the vtree in Fig. 6.1(a). First,* rootState() *creates root znode z_1 with label 2 and stores it into $Z[v_1]$. The function then calls* construct(v_1, Z). *$Z[v_1]$ contains only one znode z_1. Since z_1 has label 2,* decomp(v_1, z_1) *returns $\{(0,2), (1,1), (2,0)\}$. The function first processes label pair $(0, 2)$. Since $v_1^l = v_4$ is a leaf vnode, the function calls* terminal$(v_4, 0)$*, which returns ε. Since $v_1^r = v_2$ is not a leaf vnode, the function calls* unique$(v_2, 2, Z)$*. It creates new decomposition znode z_2 that*



(a) A graph.



(b) Subgraphs.

Figure 6.3: A graph and its subgraphs satisfying a degree constraint.

respects v_4 and has label 2, stores it into $Z[v_4]$, and returns its address. Similarly, for label pair $(1,1)$, the corresponding prime-sub pair is calculated as (B, z_3), where z_3 is a new decomposition znode that respects v_2 and has label 1. As for label pair $(2,0)$, since the universe of the prime contains only one element, we discard this pair. As a result, the function set the prime-sub pairs (ε, z_2) and (B, z_3) as child znodes of z_1. Fig. 6.2(a) shows the current intermediate ZSDD. Since $v_1^l = v_4$ is a leaf vnode and $v_1^r = v_2$ is an internal vnode, the function calls only $\mathsf{construct}(v_2, Z)$.

We go on to $\mathsf{construct}(v_2, Z)$. *$Z[v_2]$ contains two znodes z_2 and z_3. The function processes z_2 first. Since z_2 has label 2,* $\mathsf{decomp}(v_2, z_2)$ *returns $\{(2,0),(1,1),(0,2)\}$. However, $(0,2)$ is discarded because the universe of the sub only contains one element. As a result, the prime-sub pairs are calculated as $\{(z_4,\varepsilon),(z_5,C)\}$, where z_4 and z_5 are new decomposition znodes that respect v_3. The labels of z_4 and z_5 are 2 and 1, respectively. The function processes z_3 next.* $\mathsf{decomp}(v_2, z_3)$ *returns $\{(1,0),(0,1)\}$. Here, znode z_5 with label 1 already exists in $Z[v_3]$, and thus* $\mathsf{unique}(v_3, 1, Z)$ *returns z_5. As a result, the set of prime-sub pairs is $\{(z_5,\varepsilon),(z_6,C)\}$, where z_6 is a new znode that respects v_3 and has label 0. Fig. 6.2(b) shows the current intermediate ZSDD. Finally,* $\mathsf{construct}(v_3, Z)$ *is called and Fig. 6.2(c) shows the resulting ZSDD. By calling* $\mathsf{reduce}(Z)$*, the ZSDD can be trimmed as Fig. 6.1(b).*

Using Lemma 6.1, we can also bound the size of the constructed ZSDD.

Theorem 6.1. *If α is the ZSDD obtained by Algorithm 6.3, the size of α is $\mathcal{O}(|E|k^2)$.*

6.4.2 Degree

We denote a given degree constraint by function $\delta^*\colon V \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. For subgraph $S \subseteq E$, we say that S *satisfies* δ^* if $\deg_S(u) = \delta^*(u)$ holds for all $u \in V$. For example, for the graph shown in Fig. 6.3(a) and degree constraint δ^* such that $\delta^*(u_1) = \delta^*(u_4) = 1$ and $\delta^*(u_2) = \delta^*(u_3) = 2$, there are two subgraphs satisfying δ^* as shown in Fig. 6.3(b). Given G, T, and δ^*, we construct a ZSDD representing the set of all subgraphs satisfying δ^*. When a subgraph satisfies δ^*, for every vertex u, the degree of u in a subgraph must be "exactly" $\delta^*(u)$. Although we mainly discuss this "exact" constraint, we can easily modify the algorithm to deal with "at most" or "at least" constraints.

Similarly to Lemma 6.1, we show a recursive formula for the set of subgraphs satisfying the degree constraint. For vnode v, $V(v)$ denotes the set of vertices to which an edge in $E(v)$ is incident. Let us consider a degree constraint whose domain is limited to $V(v)$ as function $\delta\colon V(v) \to \mathbb{N}$. We define $f(v,\delta)$ as the family of subsets of $E(v)$ such that, for all $u \in V(v)$ and $S \in f(v,\delta)$, degree $\deg_S(u)$ equals $\delta(u)$. We show a recursive formula for $f(v,\delta)$.

Lemma 6.2. *Let v be a vnode, and δ be a function from $V(v)$ to $\mathbb{N}$. If v is a leaf vnode, let u_1 and u_2 be the endpoints of graph edge $\ell(v)$. Then, the following hold:*

$$f(v,\delta) = \begin{cases} \{\emptyset\} & (\delta(u_1) = \delta(u_2) = 0), \\ \{\{\ell(v)\}\} & (\delta(u_1) = \delta(u_2) = 1), \\ \emptyset & (\text{otherwise}). \end{cases} \tag{6.6}$$

If v is internal, the following is an $(E(v^l), E(v^r))$-partition:

$$f(v,\delta) = \bigcup_{(\delta^l,\delta^r)\in P(v,\delta)} [f(v^l,\delta^l) \sqcup f(v^r,\delta^r)], \tag{6.7}$$

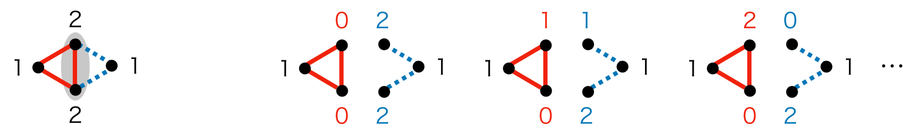
where $P(v,\delta)$ is the set of pairs of functions $\delta^l\colon V(v^l) \to \mathbb{N}$ and $\delta^r\colon V(v^r) \to \mathbb{N}$ such that

$$\forall u \in V(v^l) \cap V(v^r), \quad \delta^l(u) + \delta^r(u) = \delta(u), \tag{6.8}$$

$$\forall u \in V(v^l) \setminus V(v^r), \quad \delta^l(u) = \delta(u), \tag{6.9}$$

$$\forall u \in V(v^r) \setminus V(v^l), \quad \delta^r(u) = \delta(u). \tag{6.10}$$

For vnode v, the *frontier* of v is $F(v) = V(v^l) \cap V(v^r)$. Let us consider the graph shown in Fig. 6.3(a) and the degree constraint δ^*, which we defined above. For vnode v, let $E(v^l) = \{A, B, C\}$ and $E(v^r) = \{D, E\}$. It follows



(a) A graph. (b) Prime-sub pairs.

Figure 6.4: A graph and corresponding prime-sub pairs.

that $F(v)$ is $\{u_2, u_3\}$. Fig. 6.4(a) shows the current situation. The set of red (solid) and blue (dashed) edges are $E(v^l)$ and $E(v^r)$, respectively. The set of vertices in the shaded area is $F(v)$. We can interpret Eqs. (6.7) to (6.10) as follows. For vertex $u \in V(v^l) \setminus V(v^r)$, $\delta(u)$ edges in $E(v^l)$ must be incident to u, and thus $\delta^l(u) = \delta(u)$ (Eq. (6.9)). A similar statement holds for vertices in $V(v^r) \setminus V(v^l)$ (Eq. (6.10)). The remaining vertices are in $F(v)$. For vertex $u \in F(v)$, both edges in $E(v^l)$ and $E(v^r)$ are incident to u. Here, we guess how many edges in $E(v^l)$ are incident to u. This results in generating nine prime-sub pairs, as shown in Fig. 6.4(b). We can construct the ZSDD by recursively applying Lemma 6.2. Here we use δ as a label of a znode.

Let us analyze the sizes of ZSDDs constructed by our algorithm. The *width* of a vtree is $\max_{v \in \mathrm{in}(T)} V(v^l) \cap V(v^r)$, where $\mathrm{in}(T)$ is the set of internal vnodes.

Theorem 6.2. *If α is the ZSDD representing $f(v^{\mathrm{root}}, \delta^*)$ obtained by our algorithm, the size of α is $\mathcal{O}(|E|d^{2W})$, where $d = \max_{u \in V} \delta^*(u) + 1$ and W is the width of the input vtree.*

There exists a vtree whose width equals the *branch-width* of the graph [78]. Given such a vtree, the ZSDD size is $\mathcal{O}(|E|d^{2\mathrm{bw}(G)})$, where $\mathrm{bw}(G)$ is the branch-width of G.

6.4.3 Spanning tree

We construct a ZSDD representing the set of all spanning trees of G. With a few modifications, we can also construct a ZSDD representing the set of all connected subgraphs. We introduce some notation. If vertices u, u' are connected in subgraph $S \subseteq E$, we write $u \overset{S}{\sim} u'$. Note that $\overset{S}{\sim}$ is an equivalence relation on V; an equivalence class (a set of vertices) is a *connected component* of S. Two vertex subsets $C, C' \subseteq V$ are *connected* if there exist $u \in C$ and

$u' \in C'$ with $u \overset{S}{\sim} u'$; we write this as $C \overset{S}{\sim} C'$. We also write $u \overset{S}{\sim} C'$ if $C \overset{S}{\sim} C'$ for $C = \{u\}$.

For vnode v, let $\mathcal{C}$ be a partition of vertex set $F(v)$, that is, $\mathcal{C} = \{C_1, \ldots, C_g\}$ where $C_i \subseteq F(v)$ is a vertex set satisfying $C_i \cap C_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{g} C_i = F(v)$. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a disjoint set family defined over vertex sets in $\mathcal{C}$, that is, $R_i \subseteq \mathcal{C}$ and $R_i \cap R_j = \emptyset$ for all $i \neq j$. Let $U(\mathcal{R}) = \{C \mid \exists i : C \in R_i\}$. Function $\mathsf{Same}(\mathcal{R}, C, C')$ returns `true` if there exists $R_i \in \mathcal{R}$ such that $C, C' \in R_i$, otherwise `false`. To represent the set of all spanning trees, we define $f(v, \mathcal{C}, \mathcal{R})$ as the set of subgraphs $S \subseteq E(v)$ satisfying the following:

- for every $C_1, C_2 \in U(\mathcal{R})$, $C_1 \overset{S}{\sim} C_2$ holds if and only if $\mathsf{Same}(\mathcal{R}, C_1, C_2) =$ `true`,
- for every $C \in \mathcal{C} \setminus U(\mathcal{R})$, there exists a unique $C' \in U(\mathcal{R})$ such that $C \overset{S}{\sim} C'$. Similarly, for every $u \in V(v) \setminus F(v)$, there exists a unique $C' \in U(\mathcal{R})$ such that $u \overset{S}{\sim} C'$, and
- S does not contain a cycle.

Intuitively, $\mathcal{C}$ represents the sets of equivalent vertices. That is, vertices in the same vertex group $C \in \mathcal{C}$ are regarded to be connected. $\mathcal{R}$ represents the connectivity constraints over such equivalent sets of vertices. The first condition above requires that two vertex subsets C and C' must be connected in S if and only if they appear in the same $R \in \mathcal{R}$. The second condition requires that, every equivalent vertex subset appearing in $V(v)$ but does not appear in $\mathcal{R}$ must be connected to a vertex subset C' appearing in $\mathcal{R}$. The third condition is for acyclicity. The set of all spanning trees of G is $f(v^{\mathrm{root}}, \mathcal{C}^*, \mathcal{R}^*)$, where $\mathcal{C}^* = \{\{u\} \mid u \in F(v^{\mathrm{root}})\}$ and $\mathcal{R}^* = \{\{C\}\}$ for an arbitrary $C \in \mathcal{C}^*$ since initially there are no equivalent vertices and all vertices must be connected to form a spanning tree.

Unfortunately, it is quite complicated to show a recursive formula for $f(v, \mathcal{C}, \mathcal{R})$ and prove it theoretically. Thus, we show pseudo-code of subroutines and explain the behavior using an example. We use $(\mathcal{C}, \mathcal{R})$ as a znode label. $\mathsf{rootState}()$ returns the root znode label $(\mathcal{C}^*, \mathcal{R}^*)$. Algorithm 6.4 shows functions $\mathsf{terminal}(v, (\mathcal{C}, \mathcal{R}))$ and $\mathsf{decomp}(v, z)$. $\mathsf{terminal}(v, (\mathcal{C}, \mathcal{R}))$ returns an appropriate terminal with respect to the label of z. Let u_1 and u_2 be the endpoints of edge $\ell(v)$. We first consider the case that u_1 and u_2 are contained in the same vertex group $C \in \mathcal{C}$ (Lines 2–4). If $C \notin U(\mathcal{R})$, C must

Algorithm 6.4: Subroutines for spanning trees

Function : $\mathsf{terminal}(v, (\mathcal{C}, \mathcal{R}))$
1 Let u_1 and u_2 be the endpoints of the graph edge $\ell(v)$
2 **if** $\mathsf{Same}(\mathcal{C}, u_1, u_2) = \mathit{True}$ **then**
3   Let $C \in \mathcal{C}$ be the set containing u_1 and u_2
4   **if** $C \in U(\mathcal{R})$ **then return** ε **else return** $\perp$
5 **else**
6   Let $C_1, C_2 \in \mathcal{C}$ be the sets containing u_1 and u_2, respectively
7   **if** *neither* C_1 *nor* C_2 *is in* $U(\mathcal{R})$ **then return** $\perp$
8   **else if** *exactly one of* C_1 *or* C_2 *is in* $U(\mathcal{R})$ **then return** $\ell(v)$
9   **else**
10   **if** $\mathsf{Same}(\mathcal{R}, C_1, C_2) = \mathit{True}$ **then return** $\ell(v)$ **else return** ε

Function : $\mathsf{decomp}(v, z)$
11 $\texttt{elems} \leftarrow \emptyset$
12 Let $(\mathcal{C}, \mathcal{R})$ be the label of z
13 $\mathcal{C}^l \leftarrow \{C \cap F(v^l) \mid C \in \mathcal{C}, C \cap F(v^l) \neq \emptyset\} \cup \{\{u\} \mid u \in F(v^l) \setminus F(v)\}$
14 **for** $\mathcal{R}^l \in \mathsf{enumPartition}(\mathcal{C}^l)$ **do**
15   **if** $\mathsf{isCompatible}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l) = \mathit{True}$ **then**
16         $\mathcal{C}^r, \mathcal{R}^r \leftarrow \mathsf{calcSubState}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$
17         $\texttt{elems} \leftarrow \texttt{elems} \cup \{((\mathcal{C}^l, \mathcal{R}^l), (\mathcal{C}^r, \mathcal{R}^r))\}$

be connected to some $C' \in U(\mathcal{R})$. However, now we have $\mathcal{C} = \{C\}$, and thus there is no such C'. Therefore, we return $\perp$. If $C \in U(\mathcal{R})$, to avoid generating a cycle, we must not adopt edge $\ell(v)$. Thus we return ε. We next consider the case that u_1 and u_2 are contained in different sets $C_1, C_2 \in \mathcal{C}$ (Lines 5–10). If neither C_1 nor C_2 appear in constraints $\mathcal{R}$, they must be connected to some $C' \in U(\mathcal{R})$, but there are no such C'. Thus we return $\perp$ (Line 7). If either of C_1 or C_2 appears in $\mathcal{R}$, the unconstrained one must be connected with the other one, which has a constraint in $\mathcal{R}$. Thus we return $\ell(v)$ (Line 8). If both C_1 and C_2 appear in $\mathcal{R}$, we return the corresponding terminal depending on whether they appear in the same $R_i \in \mathcal{R}$ or not. If so, edge $\ell(v)$ must be adopted, and thus we return $\ell(v)$. Otherwise, the edge must not be adopted, and thus we return ε (Lines 9–10).

We go on to $\mathsf{decomp}(v, z)$. We first enumerate all possible set of constraints $\mathcal{R}^l$ of the prime. Since $\mathcal{R}^l$ is a partition of vertex groups $\mathcal{C}$, function $\mathsf{enumPartition}(\mathcal{C}^l)$ enumerates all partitions of $\mathcal{C}^l$. There may be partitions of $\mathcal{C}^l$ that are not *compatible* with $(\mathcal{C}, \mathcal{R})$; If $C_1 \in R_i$ and $C_2 \in R_j$ for

$R_i, R_j \in \mathcal{R}$ where $i \neq j$, they must not appear in the same $R \in \mathcal{R}^l$. In addition, for every constraint $R \in \mathcal{R}^l$, a vertex in $F(v^l)$ must appear in some $C \in R$ in order to obtain a spanning tree. If both conditions are satisfied, $\mathcal{R}^l$ is compatible with $(\mathcal{C}, \mathcal{R})$. Function $\mathsf{isCompatible}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$ returns *True* if $\mathcal{R}^l$ is compatible with $(\mathcal{C}, \mathcal{R})$, otherwise *False*. $\mathsf{calcSubState}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$ calculates $\mathcal{C}^r$ and $\mathcal{R}^r$ from its arguments. Intuitively, $\mathcal{C}^r$ and $\mathcal{R}^r$ are obtained by updating equivalent vertex groups in $\mathcal{C}$ by assuming constraints in $\mathcal{R}^l$ are satisfied. Let us give an example. Fig. 6.5(a) shows a label and Fig. 6.5(b) shows the corresponding prime-sub pairs. Five vertices $u_1, \ldots, u_5$ are on the frontier. We assume $F(v^l) = F(v^r) = F(v)$ in this example. In Fig. 6.5(a), the vertices are partitioned into three equivalency groups $\mathcal{C} = \{C_1, C_2, C_3\}$, where $C_1 = \{u_1, u_2\}$, $C_2 = \{u_3, u_4\}$, and $C_3 = \{u_5\}$. $\mathcal{C}$ is further partitioned into $\mathcal{R} = \{\{C_1\}, \{C_2, C_3\}\}$. $\mathcal{C}$ and $\mathcal{R}$ are depicted by solid and dashed rectangles, respectively. There are only two $\mathcal{R}^l$'s that are compatible with $(\mathcal{C}, \mathcal{R})$: $\mathcal{R}^l_1 = \{\{C_1\}, \{C_2\}, \{C_3\}\}$ and $\mathcal{R}^l_2 = \{\{C_1\}, \{C_2, C_3\}\}$. $\mathsf{calcSubState}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l_1)$ returns $(\mathcal{C}^r_1, \mathcal{R}^r_1)$, where $\mathcal{C}^r_1 = \{C_1, C_2, C_3\}$ and $\mathcal{R}^r_1 = \{\{C_1\}, \{C_2, C_3\}\}$. $\mathsf{calcSubState}(\mathcal{C}, \mathcal{R}, \mathcal{R}^l_2)$ returns $(\mathcal{C}^r_2, \mathcal{R}^r_2)$, where $\mathcal{C}^r_2 = \{C_1, C_4\}$, $\mathcal{R}^r_2 = \{\{C_1\}, \{C_4\}\}$, and $C_4 = C_2 \cup C_3 = \{u_3, u_4, u_5\}$.

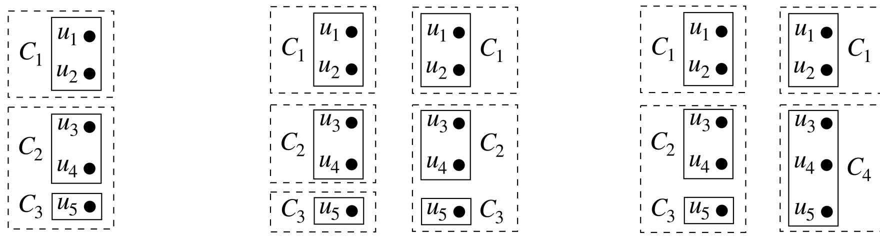
Finally, the following theorem states the bound of constructed ZSDD size.

Theorem 6.3. *If α is a ZSDD representing the set of all spanning trees constructed by our top-down algorithm, the size of α is $\mathcal{O}(|E|W^{3W})$, where W is the width of the vtree.*

As discussed in Section 6.4.2, there exists a vtree whose width equals the branch-width of the graph. Given such a vtree, the size of a constructed ZSDD is $\mathcal{O}(|E|\mathrm{bw}(G)^{3\mathrm{bw}(G)})$.

6.5 Experiments

We conduct experiments to evaluate the performance of the proposed top-down construction algorithms for ZSDDs in the same way as an existing paper [78]. The vtrees for ZSDDs are obtained by a practical algorithm to find a branch decomposition with a small width [80]. To implement the top-down algorithm for ZDDs, we use the top-down algorithm for ZSDDs with a limitation that vtrees must be right-linear. Here, a vtree is *right-linear*



(a) A state. (b) Prime-sub pairs.

Figure 6.5: Label of the connectivity constraint and corresponding prime-sub pairs.

if, for every internal vnode, its left child is a leaf. Since there is a one-to-one correspondence between ZDDs with ZSDDs using right-linear vtrees, by inputting right-linear vtrees, we can simulate ZDD construction. We use two element orders for ZDDs. The first one uses the order obtained by a breadth-first traversal of input graphs, as is used in graphillion [63], a library that implements a top-down construction algorithm for ZDDs. The other one uses the order induced from the vtrees used in the proposed method. Here we say an order is induced if a left-right traversal of a vtree gives the visiting order of variables [81]. We use the benchmark graphs of [78]: TSPLIB and RomeGraph. We constructed ZSDDs representing two types of subgraphs: 1) maximum degree at most two and 2) spanning trees. All code was written in C++ and compiled by g++-5.4.0 with -O3 option. All experiments were conducted on a machine with Intel Xeon W-2133 3.60 GHz CPU and 256 GB RAM.

Tables 6.1 and 6.2 show the results. In the tables, TD means the proposed method. Z(b) and Z(v) indicate top-down methods for ZDDs that employ breadth-first ordering and vtree traversing ordering, respectively. The empty fields indicate failure to complete within 600 seconds. We omit the instances for which all the methods finished within a second and at most one method finished within 600 seconds. In almost all cases, TD ran fastest and the sizes of ZSDDs are smaller than those of ZDDs. For example, for spanning trees (Table 6.2), the time of TD is up to 7898 times faster than Z(b), and 188 times faster than Z(v). The size of TD is up to 476 times smaller than Z(b) and 73 times smaller than Z(v). These results show the efficiency of our method. Using constructed ZDDs and ZSDDs, we can also enumerate subgraphs explicitly in polynomial time per subgraph [8, 49].

Table 6.1: Results of constructing ZSDDs and ZDDs representing the set of all subgraphs whose maximum degrees are at most 2.

			Time (ms)			Size		
instance	$\|V\|$	$\|E\|$	TD	Z(b)	Z(v)	TD	Z(b)	Z(v)
att48	48	130	381	6801	2291	194786	1065745	507169
berlin52	52	145	1021	-	36354	807660	-	5229861
eil51	51	142	1012	247736	46524	774280	27277682	5974875
grafo10106	100	119	5	2617	16	2658	15461	7529
grafo10124	100	139	9237	-	40842	3060950	-	3283397
grafo10153	100	136	3784	-	4658	832943	-	561283
grafo10183	100	132	132	-	157837	80127	-	4088915
grafo10184	100	140	4981	-	119366	1006210	-	2002968
grafo10204	100	148	156529	-	303366	15712819	-	19847326
grafo10223	100	135	863	-	5956	330554	-	826121

6.6 Conclusion

We have proposed a novel framework of algorithms for top-down ZSDD construction. We have shown the solid subroutines for three fundamental constraints: the number of edges, degree of vertices, and connectivity of vertices. We have shown the sizes of constructed ZSDDs can be bounded by the branch-width of the input graph. Experiments confirmed the efficiency of our method. Using Apply operations, we can combine several constraints. For example, we can extract connected subgraphs from ZSDD α by constructing ZSDD β representing the set of all connected subgraphs and computing $\alpha \cap \beta$. We believe that our framework can be used to solve various real-world problems.

Table 6.2: Results of constructing ZSDDs and ZDDs representing the set of all spanning trees.

			Time (ms)			Size		
instance	$\|V\|$	$\|E\|$	TD	Z(b)	Z(v)	TD	Z(b)	Z(v)
att48	48	130	3494	103871	3005	279613	5098205	387715
berlin52	52	145	11826	-	62706	937746	-	3194017
eil51	51	142	25828	-	94272	838254	-	7178190
ulysses22	22	56	39	3391	65	3036	520035	16762
grafo10106	100	119	28	221161	53	1756	836212	4057
grafo10183	100	132	2866	-	538878	224373	-	16414697
grafo10223	100	135	48563	-	128097	1009299	-	7313087
grafo10248	100	126	301	195249	672	16524	1617024	47605

Chapter 7

Conclusions and Future Directions

In this thesis, we have proposed implicit enumeration algorithms of subgraphs. Below we conclude this thesis by summarizing each contribution and suggesting future work for the contribution. We also show future directions of this research area.

Chapter 3: Evacuation Planning for General Graphs. We have dealt with the evacuation planning problem. We reformulated the convexity of components as spanning shortest path forests (SSPFs) to deal with general graphs and have proposed an algorithm to construct a ZDD representing a set of SSPFs. We have also proposed algorithms to deal with the distance and capacity constraints efficiently. As shown in experimental results using real-world map data, the proposed algorithm can construct ZDDs in a few minutes for input graphs with hundreds of edges. As future work, it is important to consider new constraints such as the reliability of roads.

Chapter 4: Balanced Graph Partition. We have proposed an algorithm to construct a ZDD representing all the graph partitions such that all the weights of its connected components are at least a given value. As shown in the experimental results, the proposed algorithm has succeeded in constructing a ZDD representing a set of more than 10^{12} graph partitions in ten seconds, which is 30 times faster than the existing state-of-the-art algorithm. Future work is devising a more memory efficient algorithm that enables us to deal with larger graphs, that is, graphs with hundreds of vertices. It is also important to seek for efficient algorithms to deal with other constraints on weights such that the ratio of the maximum and the minimum of weights is at most a specified value.

Chapter 5: Planar Subgraph Enumeration. Given graphs G and H, we have shown a method to implicitly enumerate topological-minor-embeddings of H in G using decision diagrams. We also have shown a useful application of our method to enumerating subgraphs characterized by forbidden topological minors, including planar, outerplanar, series-parallel, and cactus subgraphs. Computational experiments show that our method can find all planar subgraphs up to 122,544 times faster than a naive backtracking-based method and could solve more problems than the backtracking-based method. We have applied our method also for outerplanar, series-parallel, and cactus subgraphs. Future work is extending our method from topological minors to general minors.

Chapter 6: Frontier-based search for ZSDDs. We have proposed a novel framework of algorithms for top-down ZSDD construction. We have shown the solid subroutines for three fundamental constraints: the number of edges, degree of vertices, and connectivity of vertices. We have shown the sizes of constructed ZSDDs can be bounded by the branch-width of the input graph. Experiments confirmed the efficiency of our method. Using Apply operations, we can combine several constraints. For example, we can extract connected subgraphs from ZSDD α by constructing ZSDD β representing the set of all connected subgraphs and computing $\alpha \cap \beta$. We believe that our framework can be used to solve various real-world problems.

Open problems. We show the conclusion and future work of this thesis. In this thesis, we have proposed implicit enumeration algorithms to solve the problems more efficiently and generalize the types of subgraphs that can be enumerated. As for efficiency, although we have proposed a more efficient algorithm than the existing one for a specific problem (the balanced graph partitioning in Chapter 4), in general, the sizes of input graphs that can be dealt with by DDs are limited to small. We suggest a direction towards larger graphs in the next section.

As for generality, ZDDs can enumerate a wide range of subgraphs having forbidden graph characterization. Three types of patterns are mainly used for the characterization: subgraphs, induced subgraphs, and minors. For subgraphs, the inclusion relationship can be written as family algebra, and thus can be dealt with by ZDDs. For induced subgraphs, the inclusion relationship is more complicated, but there is an algorithm for them [72]. We have proposed an algorithm for (topological) minors in Chapter 5. We also can enumerate subgraphs without forbidden graph characterization such as spanning shortest path forests (Chapter 3). As for ZSDDs, although we have extended the types of subgraphs from matchings and paths to subgraphs with the degree and connectivity constraints (Chapter 6), there are no known methods to deal with induced subgraphs and minors.

Future directions

We show future directions of this research area.

Multiple DDs for one input graph. In implicit enumeration, the output is a single DD representing a set of subgraphs of a given graph. However, when the size of the output DD is too large, we cannot obtain any result due to memory shortage. As a result, we can deal only with graphs of relatively small sizes, say, graphs with a hundred edges. To deal with larger graphs, we consider representing the output by multiple DDs. First, we partition the input graph into some components. Next, we construct a DD for each component. The results are obtained by combining the results from each DD. In this approach, the size of each DD can be smaller than when the output is a single DD. In addition, multiple DDs can be stored in a compact way using shared-BDD [34] and variable shifting technique [82]. Technical difficulties are how to partition a graph and how to combine the results from multiple DDs.

DDs for dense input graphs. It is important to devise a new DD whose size can be small for dense graphs. As we have seen in Section 5.4, we can deal with sparse king graphs $X_{3,b}$ with thousands of edges while we were only able to dense complete graphs K_n for $n \leq 10$. It is known that, given a graph, the size of a DD representing a set of subgraphs can be bounded by a graph parameter of the input graph. The size of a ZDD and a ZSDD are bounded by the path-width an the branch-width of the input graph, respectively. These parameters are small when the graph is sparse. In contrast, the clique-width [83] is a parameter that can be small not only for sparse graphs but also for dense graphs. If we devise a new DD whose size can be bounded by the clique-width, we can deal with dense graphs efficiently.

DDs specialized for graphs. DDs are data structures for general set families. By identifying an edge set with the edge-induced subgraph, we can use DDs to represent a set of subgraphs. Although this interpretation is useful, there is a possibility that we can design DDs specialized for representing sets of subgraphs, not general set families. If we design such a DD, the size will become smaller than a general DD and we can use queries that are specific to graphs. For example, in Chapter 4, we used TDDs as intermediate data structures to design connected component operation in ZDDs. As another example, to extract subgraphs such that specified two vertices s and t are connected from a ZDD, we need another ZDD representing the set of s-t paths and use restrict operation, which does not have a polynomial-time guarantee. By designing DDs specialized for graphs, we may be able to support such queries in polynomial time. It will be useful for graph-related applications.

Bibliography

[1] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 148:1–148:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[2] Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1884–1896. SIAM, 2013.

[3] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3):678–692, 1997.

[4] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 100-A(9):1773–1784, 2017.

[5] Donald E. Knuth. *The art of computer programming, Vol. 4A, Combinatorial algorithms, Part 1.* Addison-Wesley Professional, 1st edition, 2011.

[6] Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In John Staples, Peter Eades,

Naoki Katoh, and Alistair Moffat, editors, *Algorithms and Computation, 6th International Symposium, ISAAC '95, Cairns, Australia, December 4-6, 1995, Proceedings*, volume 1004 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 1995.

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

[8] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, pages 272–277. ACM Press, 1993.

[9] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998.

[10] Ronald C. Read and Robert E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[11] Rui A. Ferreira, Roberto Grossi, Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. Amortized $\tilde{O}(|V|)$-delay algorithm for listing chordless cycles in undirected graphs. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 418–429. Springer, 2014.

[12] Takeaki Uno and Hiroko Satoh. An efficient algorithm for enumerating chordless cycles and chordless paths. In Saso Dzeroski, Pance Panov, Dragi Kocev, and Ljupco Todorovski, editors, *Discovery Science - 17th International Conference, DS 2014, Bled, Slovenia, October 8-10, 2014. Proceedings*, volume 8777 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2014.

[13] Takeo Yamada, Seiji Kataoka, and Kohtaro Watanabe. Listing all the minimum spanning trees in an undirected graph. *Int. J. Comput. Math.*, 87(14):3175–3185, 2010.

[14] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In Hon Wai Leong, Hiroshi

Imai, and Sanjay Jain, editors, *Algorithms and Computation, 8th International Symposium, ISAAC '97, Singapore, December 17-19, 1997, Proceedings*, volume 1350 of *Lecture Notes in Computer Science*, pages 92–101. Springer, 1997.

[15] Takeaki Uno. A fast algorithm for enumerating bipartite perfect matchings. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation, 12th International Symposium, ISAAC 2001, Christchurch, New Zealand, December 19-21, 2001, Proceedings*, volume 2223 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2001.

[16] Manu Basavaraju, Pinar Heggernes, Pim van 't Hof, Reza Saei, and Yngve Villanger. Maximal induced matchings in triangle-free graphs. In Dieter Kratsch and Ioan Todinca, editors, *Graph-Theoretic Concepts in Computer Science - 40th International Workshop, WG 2014, Nouan-le-Fuzelier, France, June 25-27, 2014. Revised Selected Papers*, volume 8747 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2014.

[17] Kazuhiro Kurita, Kunihiro Wasa, Takeaki Uno, and Hiroki Arimura. Efficient enumeration of induced matchings in a graph without cycles with length four. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(9):1383–1391, 2018.

[18] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2010.

[19] Christopher J. Henry and Sheela Ramanna. Maximal clique enumeration in finding near neighbourhoods. *Trans. on Rough Sets XVI*, page 103–124, 2013.

[20] Carla D. Savage. A survey of combinatorial Gray codes. *SIAM Rev.*, 39(4):605–629, 1997.

[21] David Avis and Komei Fukuda. Reverse search for enumeration. *Discret. Appl. Math.*, 65(1-3):21–46, 1996.

[22] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[23] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[24] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvement of boolean comparison method based on binary decision diagrams. In *1988 IEEE International Conference on Computer-Aided Design, ICCAD 1988, Santa Clara, CA, USA, November 7-10, 1988. Digest of Technical Papers*, pages 2–5. IEEE Computer Society, 1988.

[25] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *1988 IEEE International Conference on Computer-Aided Design, ICCAD 1988, Santa Clara, CA, USA, November 7-10, 1988. Digest of Technical Papers*, pages 6–9. IEEE Computer Society, 1988.

[26] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1990, Santa Clara, CA, USA, November 11-15, 1990. Digest of Technical Papers*, pages 126–129. IEEE Computer Society, 1990.

[27] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In Richard C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 46–51. IEEE Computer Society Press, 1990.

[28] Yusuke Matsunaga and Masahiro Fujita. Multi-level logic optimization using binary decision diagrams. In *1989 IEEE International Conference on Computer-Aided Design, ICCAD 1989, Santa Clara, CA, USA, November 5-9, 1989. Digest of Technical Papers*, pages 556–559. IEEE Computer Society, 1989.

[29] Rolf Drechsler, Nicole Drechsler, and Wolfgang Günther. Fast exact minimization of BDDs. In Basant R. Chawla, Randal E. Bryant, and

Jan M. Rabaey, editors, *Proceedings of the 35th Conference on Design Automation, Moscone center, San Francico, California, USA, June 15-19, 1998*, pages 200–205. ACM Press, 1998.

[30] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In Tony Ambler, Jochen A. G. Jess, and Hugo De Man, editors, *Proceedings of the conference on European design automation, EURO-DAC'91, Amsterdam, The Netherlands, 1991*, pages 50–54. EEE Computer Society, 1991.

[31] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In Michael R. Lightner and Jochen A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 42–47. IEEE Computer Society / ACM, 1993.

[32] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In Kam-Wing Ng, Prabhakar Raghavan, N. V. Balasubramanian, and Francis Y. L. Chin, editors, *Algorithms and Computation, 4th International Symposium, ISAAC '93, Hong Kong, December 15-17, 1993, Proceedings*, volume 762 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 1993.

[33] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In Richard C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 40–45. IEEE Computer Society Press, 1990.

[34] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In Richard C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 52–57. IEEE Computer Society Press, 1990.

[35] Elsa Loekito and James Bailey. Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams. In

Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 307–316. ACM, 2006.

[36] Shin-ichi Minato and Hiroki Arimura. Frequent pattern mining and knowledge indexing based on zero-suppressed BDDs. In Saso Dzeroski and Jan Struyf, editors, *Knowledge Discovery in Inductive Databases, 5th International Workshop, KDID 2006, Berlin, Germany, September 18, 2006, Revised Selected and Invited Papers*, volume 4747 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2006.

[37] Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, volume 5012 of *Lecture Notes in Computer Science*, pages 234–246. Springer, 2008.

[38] Yuko Sakurai, Suguru Ueda, Atsushi Iwasaki, Shin-ichi Minato, and Makoto Yokoo. A compact representation scheme of coalitional games based on multi-terminal zero-suppressed binary decision diagrams. In David Kinny, Jane Yung-jen Hsu, Guido Governatori, and Aditya K. Ghose, editors, *Agents in Principle, Agents in Practice - 14th International Conference, PRIMA 2011, Wollongong, Australia, November 16-18, 2011. Proceedings*, volume 7047 of *Lecture Notes in Computer Science*, pages 4–18. Springer, 2011.

[39] Olivier Coudert. Solving graph optimization problems with ZBDDs. In *European Design and Test Conference, ED&TC '97, Paris, France, 17-20 March 1997*, pages 224–228. IEEE Computer Society, 1997.

[40] David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.

[41] Shinsaku Sakaue, Masakazu Ishihata, and Shin-ichi Minato. Efficient bandit combinatorial optimization algorithm with zero-suppressed binary decision diagrams. In Amos J. Storkey and Fernando Pérez-Cruz,

editors, *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, volume 84 of *Proceedings of Machine Learning Research*, pages 585–594. PMLR, 2018.

[42] Shinsaku Sakaue. *Online, Submodular, and Polynomial Optimization with Discrete Structures*. PhD thesis, Kyoto University, March 2020.

[43] Elsa Loekito, James Bailey, and Jian Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowl. Inf. Syst.*, 24(2):235–268, 2010.

[44] Shin-ichi Minato. πDD: A new decision diagram for efficient problem solving in permutation space. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2011.

[45] Yuma Inoue and Shin-ichi Minato. An efficient method for indexing all topological orders of a directed graph. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, volume 8889 of *Lecture Notes in Computer Science*, pages 103–114. Springer, 2014.

[46] Takanori Maehara and Yuma Inoue. Group decision diagram (GDD): A compact representation for permutations. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 2986–2994. AAAI Press, 2019.

[47] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1990, Santa Clara, CA, USA, November 11-15, 1990. Digest of Technical Papers*, pages 92–95. IEEE Computer Society, 1990.

[48] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826. IJCAI/AAAI, 2011.

[49] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1058–1066. AAAI Press, 2016.

[50] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.

[51] Hiroshi Imai, Kyoko Sekine, and Keiko Imai. Computational investigations of all-terminal network reliability via BDDs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A:714–721, 1999.

[52] Gary Hardy, Corinne Lucet, and Nikolaos Limnios. K-terminal network reliability measures with binary decision diagrams. *IEEE Trans. Reliab.*, 56(3):506–515, 2007.

[53] Takeru Inoue. Reliability analysis for disjoint paths. *IEEE Trans. Reliab.*, 68(3):985–998, 2019.

[54] Hirofumi Suzuki, Masakazu Ishihata, and Shin-ichi Minato. Exact computation of strongly connected reliability by binary decision diagrams. In Donghyun Kim, R. N. Uma, and Alexander Zelikovsky, editors, *Combinatorial Optimization and Applications - 12th International Conference, COCOA 2018, Atlanta, GA, USA, December 15-17, 2018, Proceedings*, volume 11346 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2018.

[55] Jun Kawahara, Koki Sonoda, Takeru Inoue, and Shoji Kasahara. Efficient construction of binary decision diagrams for network reliability with imperfect vertices. *Reliab. Eng. Syst. Saf.*, 188:142–154, 2019.

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

[57] Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution loss minimization with guaranteed error bound. *IEEE Trans. Smart Grid*, 5(1):102–111, 2014.

[58] Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. Exact computation of influence spread by binary decision diagrams. In Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors, *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 947–956. ACM, 2017.

[59] Atsushi Takizawa, Yasufumi Takechi, Akio Ohta, Naoki Katoh, Takeru Inoue, Takashi Horiyama, Jun Kawahara, and Shin-ichi Minato. Enumeration of region partitioning for evacuation planning based on ZDD. In *Proc. of the 11th International Symposium on Operations Research and its Applications in Engineering, Technology and Management 2013 (ISORA 2013)*, 2013.

[60] Jun Kawahara, Takashi Horiyama, Keisuke Hotta, and Shin-ichi Minato. Generating all patterns of graph partitions within a disparity bound. In Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings*, volume 10167 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 2017.

[61] Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Solving the longest oneway-ticket problem and enumerating letter graphs by augmenting the two representative approaches with ZDDs. In *Computational Intelligence in Information Systems*, pages 294–305, 2017.

[62] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.

[63] Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. Graphillion: software library for very large sets of labeled graphs. *Int. J. Softw. Tools Technol. Transf.*, 18(1):57–66, 2016.

[64] Yuma Inoue and Shin-ichi Minato. Acceleration of ZDD construction for subgraph enumeration via path-width optimization. *TCS-TR-A-16-80. Hokkaido University*, 2016.

[65] Simone Bova. SDDs are exponentially more succinct than OBDDs. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 929–935. AAAI Press, 2016.

[66] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[67] Ryo Yoshinaka, Jun Kawahara, Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato. Counterexamples to the long-standing conjecture on the complexity of BDD binary operations. *Inf. Process. Lett.*, 112(16):636–640, 2012.

[68] Danny Z. Chen, Jinhee Chun, Naoki Katoh, and Takeshi Tokuyama. Efficient algorithms for approximating a multi-dimensional voxel terrain by a unimodal terrain. In Kyung-Yong Chwa and J. Ian Munro, editors, *Computing and Combinatorics, 10th Annual International Conference, COCOON 2004, Jeju Island, Korea, August 17-20, 2004, Proceedings*, volume 3106 of *Lecture Notes in Computer Science*, pages 238–248. Springer, 2004.

[69] Hiroaki Iwashita and Shin-ichi Minato. Efficient top-down ZDD construction techniques using recursive specifications. *TCS Technical Reports*, TCS-TR-A-13-69, 2013.

[70] Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shoji Kasahara. Enumerating all spanning shortest path forests with distance and capacity constraints. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(9):1363–1374, 2018.

[71] Koichi Yasuoka. A new method to represent sets of products: ternary decision diagrams. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 78(12):1722–1728, 1995.

[72] Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Colorful frontier-based search: Implicit enumeration of chordal and in-

terval subgraphs. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis of Experimental Algorithms - Special Event, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2019.

[73] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[74] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, 1999.

[75] Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.

[76] Takashi Horiyama, Jun Kawahara, Shin-ichi Minato, and Yu Nakahata. Decomposing a graph into unigraphs. *CoRR*, abs/1904.09438, 2019.

[77] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[78] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1213–1221. AAAI Press, 2017.

[79] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

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

[81] Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012.

[82] Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 1995.

[83] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discret. Appl. Math.*, 101(1-3):77–114, 2000.