NAIST-IS-MT1651202

Master's Thesis

Enumerating and Indexing Constrained Graph Partitions Using Decision Diagrams

Yu Nakahata

September 12, 2018

Graduate School of Information Science
Nara Institute of Science and Technology

A Master's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
MASTER of ENGINEERING

Yu Nakahata

Thesis Committee:

Professor Shoji Kasahara	(Supervisor)
Professor Minoru Ito	(Co-supervisor)
Associate Professor Masahiro Sasabe	(Co-supervisor)
Associate Professor Takashi Horiyama	(Saitama University)
Assistant Professor Jun Kawahara	(Co-supervisor)

Enumerating and Indexing Constrained Graph Partitions Using Decision Diagrams*

Yu Nakahata

Abstract

Graph partitioning is important for several applications such as evacuation planning, political redistricting, VLSI design, and so on. Many types of graph partitioning problems are NP-hard, and thus it is difficult to find a good graph partition efficiently. In addition, it is often the case that there are several objective functions and we cannot decide the priority between them clearly. Then it is difficult to define what is the optimal solution. In such a case, it is useful to enumerate some solutions with moderately good objectives. However, there are exponentially many graph partitions in a graph, and thus it is difficult to enumerate such graph partitions efficiently.

In this thesis, we study an approach using a *zero-suppressed binary decision diagram* (ZDD), which is a data structure representing a family of sets in a compressed way. Especially, we focus on two types of graph partitioning problems and propose algorithms to construct ZDDs representing sets of such graph partitions. First, we deal with the *evacuation planning problem*, which asks us to partition a target area, represented by a graph, into several regions so that each region contains exactly one shelter. The problem has many complex constraints on the distances between evacuees and the assigned shelters, the capacities of shelters, and the convexity of components such that the intersections of evacuation routes less occur. There is an existing method to construct a ZDD representing a set of graph partitions satisfying the constraints. However, the algorithm is limited to a grid graph because the definition of convexity of components is specialized

*Master's Thesis, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-MT1651202, September 12, 2018.

for grid graphs. To deal with general graphs, we reformulated the definition of convexity of components as spanning shortest path forests (SSPFs), and propose an algorithm to construct a ZDD representing a set of SSPFs in a given graph. Experimental results using real-world map data show that our algorithm can construct a ZDD for graphs with hundreds of edges in a few minutes.

Second, we deal with *balanced graph partitions*. When we are given a vertex-weighted graph, it is important to find a graph partition such that each weight of its connected component is in a given range for applications such as political redistricting, VLSI design, parallel processing, and so on. There is an existing algorithm to construct a ZDD representing the set of balanced graph partitions. However, the computation is tractable only for graphs with less than a hundred of vertices because the algorithm tries to construct the ZDD in one step, which leads to the increase of the memory consumption exponentially. To construct a ZDD more efficiently, we propose a new algorithm for the problem. The proposed algorithm divides the construction of the ZDD into several steps and suppresses the memory consumption. Experimental results show that our algorithm runs up to tens of times faster than the existing algorithm.

Keywords:

Graph algorithm, Graph partitioning, Decision diagram, Frontier-based search, Enumeration problem

Contents

List of Figures

List of Tables

1 Introduction

Graph partitioning is important for several applications such as evacuation planning [1], political redistricting [2], VLSI design [3], and so on. Many types of graph partitioning problems are NP-hard [4], and thus it is difficult to find a good graph partition efficiently. Therefore, many heuristics and approximation algorithms have been proposed [4, 5]. However, such algorithms do not always yield an optimal solution. In addition, it is often the case that there are several objective functions and we cannot decide the priority between them clearly. Then it is difficult to define what is the optimal solution. In such a case, it is useful to enumerate some solutions with moderately good objectives. However, there are exponentially many graph partitions in a graph, and thus it is difficult to enumerate such graph partitions efficiently.

Recently, an approach using a *zero-suppressed binary decision diagram* (ZDD) [6] has been proposed. A ZDD is a data structure representing a family of sets in a compressed way. Using a ZDD, we can represent a set of graph substructures in a given graph G, including graph partitions, as a family of edge subsets of G. There are some algorithms to efficiently construct a ZDD representing some types of graph substructures such as paths [7], cycles [7], forests [8], and so on. A framework of such algorithms are called *frontier-based search* [7, 9, 10]. A ZDD can sometimes represent a set of graph substructures exponentially smaller than the explicit list of them, and thus an algorithm based on frontier-based search can sometimes construct a ZDD exponentially faster than explicitly enumerating graph substructures. Using a ZDD, we can process many queries for a family of sets. For example, we can count the number of sets in the family, randomly sample a set, extract the sets including or excluding a specified element from a ZDD, extract the set with minimum or maximum weight, and so on. In addition, when we are given two ZDDs, we can construct a ZDD representing the union, in-

tersection, or difference of the two families represented by the input ZDDs. These queries can be answered without decompressing ZDDs. Therefore, constructing a ZDD representing a set of graph substructures is useful and the approach is used in many fields such as electrical distribution networks [8], network reliability evaluation [11, 12], political redistricting [13], and so on.

In this thesis, we focus on two types of graph partitioning problems and propose algorithms to construct ZDDs representing sets of such graph partitions. First, we deal with the *evacuation planning problem*, which asks us to partition a target area, represented by a graph, into several regions so that each region contains exactly one shelter. 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. Since the problem has many objective functions, it is useful to enumerate graph partitions with good objectives. Takizawa et al. [1] proposed an algorithm using a ZDD. They first split a target area into square cells. They adopted the convexity proposed by Chen et al. [14] and enumerate graph partitioning satisfying the convexity, distance, and capacity constraints. However, since they split the area into square cells, their method is only appropriate for areas with a regular structure such as Kyoto, a city in Japan. The definition of convexity of Chen et al. is limited to grid graphs and it is difficult to extend the definition to general graphs. In order to deal with *general graphs*, we reformulate the convexity for general graphs from the definition for grid graphs. We formulate the convexity in a general graph as a *spanning shortest path forest* (SSPF). Using SSPFs, we can reduce intersections of evacuation routes when evacuees go to the assigned shelters in shortest paths. We propose an algorithm to construct a ZDD representing a set of SSPFs in a given graph based on frontier-based search. We also propose an algorithm to deal with the distance and capacity constraint. Experimental results using real-world map data show that our algorithm can construct a ZDD for graphs with hundreds of edges in a few minutes.

Second, we deal with *balanced graph partitions*. When we are given a vertex-weighted graph, it is important to find a graph partition such that each weight of its connected component is in a given range. There are applications such as

political redistricting, VLSI design, parallel processing, and so on. There is an approach based on integer linear programming (ILP) [2]. Their algorithm finds a graph partition with the smallest disparity, where the disparity is the maximum ratio of the weights of two connected components in a partition. However, in practice, there are many other constraints such as geographical, sociological, and political requirements. It is difficult to write such constraints in ILP and solve it. Kawahara et al. [13] proposed an algorithm to construct a ZDD representing the set of graph partitions such that the weights of their connected components are within a given range. However, the computation is tractable only for graphs with less than a hundred of vertices. We can design an algorithm for the upper-bound constraint, that is, weights of connected components are at most a given value, with a small modification of the algorithm for the shelter-capacity constraint which we propose in Chapter 4. However, it is difficult to extend the algorithm to the lower-bound constraint, that is, weights of connected components are at least a given value. In Chapter 5, we propose an efficient algorithm to construct a ZDD for the lower-bound constraint to enumerate balanced graph partitions. Experimental results show that our algorithm runs up to tens of times faster than the existing algorithm.

The rest of this thesis is organized as follows. Chapter 2 gives related work. In Chapter 3, we give preliminaries commonly used in the thesis. Chapters 4 and 5 describe the studies on the evacuation planning problem and the balanced graph partitioning, respectively. Chapter 4 is based on "Enumerating All Spanning Shortest Path Forests with Distance and Capacity Constraints" [15], which is to be appeared in IEICE Transactions on Fundamentals of Electronics. Chapter 5 is based on "Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams" [16], which is a proceedings of the 17th International Symposium on Experimental Algorithms (SEA 2018). We give the conclusion in Chapter 6.

2 Related work

An approach using a zero-suppressed binary decision diagram (ZDD) [6] is studied in various fields. Sekine et al. [10] proposed an algorithm that constructs the ZDD representing all the spanning trees of a given graph. Knuth [7] shows how to create a ZDD for s-t paths. Kawahara et al. [9] generalized their algorithms, which can treat various kinds of subgraphs. Inoue et al. [8] presented an algorithm that constructs a ZDD for rooted spanning forests and utilized it to minimize the loss of electricity in an electrical distribution network. Kawahara et al. [13] designed an algorithm to enumerate and index all the partitions of a given graph into the specified number of components. There are other studies related to reliability evaluation [11, 12], enumerating puzzle problems [17], solving a variant of the longest path problem [18] and exact calculation of impact diffusion in Web [19].

Graph partitioning has been studied in various fields such as evacuation planning [1], political redistricting [2], VLSI design [3], and so on. There are several models for balanced graph partitions. One model is (k, ν)-balanced graph partition, which is a graph partition such that it has exactly k components and each component has at most $\lfloor \nu n/k \rfloor$ vertices, where, for a real number a, $\lfloor a \rfloor$ is the largest integer which is not more than a. Andreev and Račke [4] show that the problem is NP-hard and no polynomial time approximation algorithm can guarantee a finite approximation ratio unless P = NP when $\nu = 1$. They also show that there exists $\mathcal{O}(\log^2 n)$ approximation algorithm for any constant $\nu > 1$. There are algorithms based on local search [20, 21], tabu search [22, 23], and flow based heuristics [5, 24].

3 Preliminaries

In this Chapter, we give preliminaries commonly used in the thesis. We explain a zero-suppressed binary decision diagram (ZDD) and frontier-based search.

3.1 Zero-suppressed binary decision diagram

A *zero-suppressed binary decision diagram* (ZDD) [6] 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*.* 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 destination of the x-arc of a non-terminal node α the *x-child* of α. We denote the label of α by $l(\alpha)$ and in this paper, assume that $l(\alpha) \in \mathbb{Z}^+ \cup \{\infty\}$ for any $\alpha \in N_Z$, where $\mathcal{Z}^+$ is the set of positive integers. For convenience, we let $l(\top) = l(\bot) = \infty$. For each directed 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 $|Z|$.

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 $p = (n_1, a_1, n_2, a_2, \ldots, n_k, a_k, \top) \in \mathcal{P}_Z$ 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. As an example, we illustrate the ZDD representing the family $\{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$ in Fig. 3.1. In the figure, a dashed arc (⇢) and

*To avoid confusion, we use the words "vertex" and "edge" for input graphs and "nodes" and "arcs" for decision diagrams.

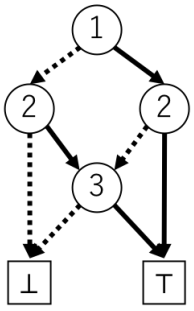

Figure 3.1: Example of a ZDD

a solid arc ($\rightarrow$) are a 0-arc and a 1-arc, respectively. On the ZDD in Fig. 3.1, 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. We denote a ZDD representing a family $\mathcal{F}$ by $Z_{\mathcal{F}}$.

3.2 Frontier-based search

Frontier-based search [7, 9, 10] 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 frontier-based search. 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 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 $n_i.\texttt{conf}$, 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 $\bot$ 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}.

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

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

As for $n_i.\texttt{conf}$, 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 [7] (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 $F_i = (\bigcup_{j=1}^{i-1}\{\{u,v\} \mid e_j = \{u,v\}\}) \cap (\bigcup_{k=i}^{m}\{\{u,v\} \mid e_k = \{u,v\}\})$. For convenience, we define $F_0 = F_m = \emptyset$. States of vertices in F_{i-1} are stored into $n_i.\texttt{conf}$. 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.

The efficiency of an algorithm based on frontier-based search 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.

4 Evacuation planning for general graphs

4.1 Summary

This chapter studies a variant of the graph partitioning problem, called the evacuation planning problem, which asks us to partition a target area, represented by a graph, into several regions so that each region contains exactly one shelter. 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. This chapter formulates the convexity of connected components as a *spanning shortest path forest* for general graphs, and proposes 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 *zero-suppressed binary decision diagrams* (ZDDs) as a compressed expression. 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 of edges in a few minutes.

4.2 Introduction

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. [1] 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. [14]. 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. [1]). 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 parti-

tions satisfying the structural and distance constraints. As we show in Section 4.5, 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* [7, 9, 10], 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 [13], 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 [13] 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 4.3, we give some preliminaries and formulate our problem. We propose our algorithm in Sections 4.4 and 4.5. Section 4.6 gives experimental results.

4.3 Preliminaries

4.3.1 Notation

Let an input graph be a vertex and edge weighted graph $G = (V, E, popu, w)$. Assume that G is simple, connected and undirected. Here, $V = \{1, 2, \ldots, n\}$ 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, where $\mathbb{Z}^+$ is a set of positive integers. For a vertex v, $popu(v)$ indicates the population of v. The function $w : E \to \mathbb{R}^+$ is an edge weight function, where $\mathbb{R}^+$ is a set of positive real numbers. 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 $S = \{1, 2, \ldots, r\} \subseteq V$ 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 $U \subseteq V, G[U]$ is a vertex induced subgraph in G by U and, for $E' \subseteq E, G[E']$ is an edge induced subgraph in G by E'. We regard $U \subseteq V$ and $E' \subseteq E$ in the same light as $G[U]$ and $G[E']$, respectively. For a vertex v and a subgraph $E' \subseteq E$, let $C_{E'}(v)$ be the set of the vertices adjacent to v, containing v. 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.

4.3.2 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.

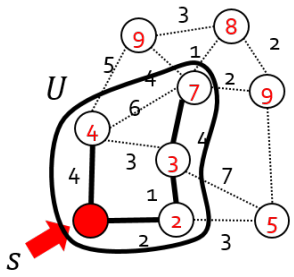

Figure 4.1: Example of a shortest path tree

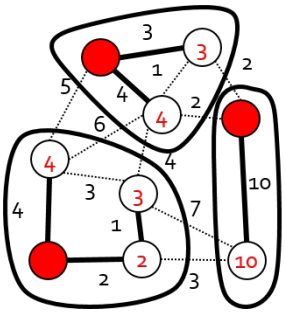

Figure 4.2: Example of a spanning shortest path forest

Definition 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* $u \in U$*.*

Figure 4.1 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 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*.*

Figure 4.2 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,

$$\forall s \in S, \sum_{v \in C_F(s)} popu(v) \leq K \cdot cap(s), \tag{4.1}$$

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*:

$$\forall v \in V, d(v, s_v) \leq \min\{D, R \cdot d^*(v)\}, \tag{4.2}$$

     where s_v is the nearest shelter to v in F.
  2. The *shelter-capacity constraint*:

$$\forall s \in S, \sum_{v \in C_F(s)} popu(v) \leq K \cdot cap(s). \tag{4.3}$$

4.4 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 4.5, 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.

4.4.1 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. (4.4) and (4.5) is satisfied:

$$d_G(s, u) + w(e) = d_G(s, v), \tag{4.4}$$

$$d_G(s, v) + w(e) = d_G(s, u). \tag{4.5}$$

Conversely, if either Eqs. (4.4) or (4.5) 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. (4.4) and (4.5) are never satisfied simultaneously. In T, we orient e in the direction $u \to v$ if Eq. (4.4) is satisfied, which implies u is a parent in T, and $v \to u$ if Eq. (4.5) 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 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 3.2, 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.

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 3.2. 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 4.3, 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,

$$\texttt{indeg}[s][x] = \left| \left\{ e \in C(x) \;\middle|\; \begin{array}{l} e = \{y, x\}, \\ d_G(s, y) + w(e) = d_G(s, x) \end{array} \right\} \right|. \tag{4.6}$$

Third, when there is an edge e in a connected component C in G' containing no shelter such that neither Eqs. (4.4) nor (4.5) 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 (4.4) 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 (4.4) is satisfied for e_i and s.

(d2) Equation (4.5) is satisfied for e_i and s.

(d3) Neither Eqs. (4.4) nor (4.5) 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 $\texttt{valid}[s][C(v)]$ in N', otherwise 1 for $\texttt{indeg}[s][v]$ 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`.

4.4.2 More memory-efficient algorithm

In Subsection 4.4.1, 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 Subsection 4.4.1, 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. (4.4) and (4.5): $u \to v$ 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 Subsection 4.4.1 without storing `indeg` into ZDD nodes. This reduces memory consumption. Pseudocode is presented in Algorithms 1–5.

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 1. *The width of a ZDD constructed by Algorithms 1–5 is* $\mathcal{O}(B_f 2^{rf})$.

In the algorithm in Subsection 4.4.1, 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 Subsection 4.4.1. 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`.

4.5 Shelter-Capacity constraint

In this section, we propose how to deal with the shelter-capacity constraint efficiently. Kawahara et al. [13] 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 4.4, 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 Subsection 4.4.1. However, here we use the new value -1. $\texttt{cmp}[v] = -1$ 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 Subsection 4.4.1, 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*. Because x is adopted, the total population of adopted vertices is updated as $\texttt{sm\_popu} \leftarrow \texttt{sm\_popu} + \mathit{popu}(x)$ 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: $\mathit{cap\_max} = \max\{\mathit{cap}(v) \mid v \in S\}$ and $\mathit{popu\_max} = \max\{\mathit{popu}(v) \mid v \in V\}$. If $\texttt{sm\_popu} > \mathit{cap\_max} + \mathit{popu\_max}$ 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 $\texttt{sm\_popu} > \mathit{cap}(s)$ in N', then C satisfies 1 and 2. So we should make $\mathbf{1}$ 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 $\mathbf{0}$. In the case $i = m$, which indicates G' has no edges, we should also make $\mathbf{0}$.

In order to extend the algorithm to cases such that $K > 1$, we only have to set $\mathit{cap}(s) \leftarrow K \cdot \mathit{cap}(s)$ for all $s \in S$ before running the algorithm. Pseudocode is presented in Algorithm 6.

*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 6), and thus we can simply set $\texttt{cmp}[x] \leftarrow x$ here without loss of correctness.

Let us consider the width of a ZDD constructed by Algorithm 6. Algorithm 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`[†]. Therefore we obtain the following lemma.

Lemma 2. *The width of a ZDD constructed by Algorithm 6 is* $\mathcal{O}(B_f P)$.

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

$$Z_3 = Z_1 \searrow Z_2 = \{\alpha \in Z_1 \mid \forall \beta \in Z_2, \alpha \not\supseteq \beta\}. \tag{4.7}$$

This operation is known as *nonsupset* [7]. The operation can be realized by using *restrict* operation. Z_1.Restrict(Z_2) is defined as

$$Z_1.\mathrm{Restrict}(Z_2) = \{\alpha \in Z_1 \mid \exists \beta \in Z_2, \alpha \supseteq \beta\}. \tag{4.8}$$

From Eqs. (4.7) and (4.8), we obtain the following equation:

$$Z_3 = Z_1 \setminus Z_1.\mathrm{Restrict}(Z_2). \tag{4.9}$$

Both setminus and restrict operations are supported in ordinary ZDD libraries, so we can utilize them.

As for Eq. (4.8), the smaller number of nodes of Z_2 leads to faster calculation. However, as we show in Section 4.6, 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 6, it makes the time complexity of the algorithm worse. Therefore we use an operation between ZDDs instead. Here we use *permit* operation defined as

$$A.\mathrm{Permit}(B) = \{\alpha \in A \mid \exists \beta \in B, \alpha \subseteq \beta\}. \tag{4.10}$$

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

Permit operation is similar to restrict operation. The difference between them is "$\supseteq$" and "$\subseteq$". Using permit operation, our procedure is written as follows:

$$Z_3 = Z_1 \setminus Z_1.\mathrm{Restrict}(Z_2'), \tag{4.11}$$

where

$$Z_2' = Z_2.\mathrm{Permit}(Z_1). \tag{4.12}$$

4.6 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 [25] to implement algorithms based on frontier-based search. To perform operations between ZDDs, we adopted the `SAPPOROBDD` library.

4.6.1 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`[‡], 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[§]. 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. 4.3. We assumed that $popu(v) = 1$

[‡] `https://www.openstreetmap.org`

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

Figure 4.3: The map data of the target area (© OpenStreetMap contributors)

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 Tab. 4.1.

4.6.2 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.

4.6.3 Results

We show the results in Tab. 4.2. D, R and K are the parameters described in Subsection 4.3.2, 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 Sects. 4.4 and 4.5. 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.

Algorithm 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 2: UPDATESTATE(N', i)

```
  // update information of node N' when we adopt e_i
1 if CHECKINDEG(N', i) returns false then
2     return false
3 if UPDATEVALID(N', i) returns false then
4     return false
  // update cmp
5 C_min = min{cmp[u], cmp[v]}
6 C_max = max{cmp[u], cmp[v]}
7 for x ∈ F_{i-1} ∪ e_i such that cmp[x] = C_max do
8     cmp[x] ← c_min
9 return true
```

Algorithm 3: CHECKINDEG(N', i)

```
  // check if we can adopt e_i in N' with respect to the
     constraint of in-degrees
1 if cmp[u] > r and cmp[v] ≤ r then
2     swap u and v.
3 if cmp[u] ≤ r and cmp[v] > r then
      // cmp[u] contains a shelter and cmp[v] contains no shelter.
4     s ← cmp[u]
5     if d(s,u) + w(e_i) ≠ d(s,v) then
6         return false
7     else if IsNearestInCmp(N', v, s) returns false then
8         return false            // the in-degree of v is not zero.
9 else
                  // Neither cmp[u] nor cmp[v] contains any shelter.
10    for s ∈ S do
11        if d(s,v) + w(e_i) = d(s,u) then
12            swap v and u.
13        if d(s,u) + w(e_i) = d(s,v) then
14            if IsNearestInCmp(N', v, s) returns false then
15                valid[s][cmp[v]] ← false
16        else
17            valid[s][cmp[u]] ← false
18            valid[s][cmp[v]] ← false
19 return true
```

Algorithm 4: UPDATEVALID(N', i)

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

Algorithm 5: ISNEARESTINCMP(N', x, s)

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

Algorithm 6: MakeNewNode2($N, i, take$)

Let $e_i = \{u, v\}$.
Copy N to N'.
if $take = 1$ **then**
    for $x \in e_i$ *such that* $\texttt{cmp}[x] = -1$ **do**
        $\texttt{cmp}[x] \leftarrow x$
        $\texttt{sm\_popu} \leftarrow \texttt{sm\_popu} + popu(x)$
    if $\texttt{sm\_popu} > cap\_max + popu\_max$ **then**
        return 0
    if $\texttt{cmp}[u] = \texttt{cmp}[v] \neq -1$ **then**
        return 0
    `// update cmp`
    $C_{\min} = \min\{\texttt{cmp}[u], \texttt{cmp}[v]\}$
    $C_{\max} = \max\{\texttt{cmp}[u], \texttt{cmp}[v]\}$
    for $x \in F_{i-1} \cup e_i$ *such that* $\texttt{cmp}[x] = c_{\max}$ **do**
        $\texttt{cmp}[x] \leftarrow c_{\min}$
if C *is the only connected component on the frontier* **and** C *contains a shelter* s **and** $\texttt{sm\_popu} > cap(s)$ **then**
    return 1
if *there exists a connected component leaves the frontier* **or** $i = m$ **then**
    return 0
return N'

Table 4.1: Capacities of shelters

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

Table 4.2: Experimental results for real-world map data

						Z_1		Z_2		Z_3		
Graph Name	D	R	K	n	m	# node	Time	# node	Time	# node	Time	# solution
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

5 Balanced graph partition

5.1 Summary

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. However, there are a vast number of graph partitions in a graph, and thus it is difficult to enumerate desired graph partitions efficiently. In this chapter, an algorithm to enumerate all the graph partitions such that all the weights of the connected components are at least a specified value is proposed. To deal with a large search space, we use zero-suppressed binary decision diagrams (ZDDs) to represent sets of graph partitions and we design a new algorithm based on frontier-based search, which is a framework to directly construct a ZDD. 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.

5.2 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 compo-

nents (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. [13] proposed an algorithm to construct a ZDD representing the set of balanced graph partitions by frontier-based search [7, 9, 10], 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. [15] 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 [26]. 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., [15]), rooted spanning forests used for power distribution networks (e.g., [8]), and simply connected components representing regions (e.g., [13]), 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) [27], 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 5.3, we give preliminaries. We describe an overview of our algorithm in Subsection 5.4.1, and the detail in the rest of Section 5.4. Section 5.5 gives experimental results.

5.3 Preliminaries

5.3.1 Notation

Let $\mathbb{Z}^+$ be the set of positive integers. For $k \in \mathbb{Z}^+$, we define $[k] = \{1, 2, \ldots, k\}$. 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 $N(E', v) = \{u \mid \{u, v\} \in E'\}$.

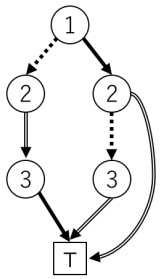

Figure 5.1: Example of a TDD

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 $\mathrm{abs}(S^\pm) = \{e \mid (+e \in S^\pm) \vee (-e \in S^\pm)\}$.

5.3.2 Ternary decision diagram

A *ternary decision diagram* (TDD) [27] 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*.

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. 5.1 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, $\bot$ and the arcs pointing at it are omitted for simplicity. The TDD in the figure has three directed paths from the root node to $\top$: $1 \to 2 \Rightarrow \top$, $1 \to 2 \dashrightarrow 3 \Rightarrow \top$, and $1 \dashrightarrow 2 \Rightarrow 3 \to \top$, which correspond to $\{+1, -2\}, \{+1, -3\}$, and $\{-2, +3\}$, respectively.

5.4 Algorithms

5.4.1 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. [13] 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 [26]. 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, 26].

Nakahata et al. [15] 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 $Z_{\mathcal{A}}$.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 $Z_{\mathcal{A}}$.restrict($Z_{\mathcal{S}}$) from $Z_{\mathcal{A}}$ by the set difference operation $Z_{\mathcal{A}} \setminus (Z_{\mathcal{A}}.\text{restrict}(Z_{\mathcal{S}}))$ [6], 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 following observation.

Observation 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 1, we associate a signed subgraph $S^{\pm}$ with a connected subgraph $(\mathrm{dom}(S), S)$:

$$S^{\pm} = S^{+} \cup S^{-}, \tag{5.1}$$

$$S^{+} = \{+e \mid e \in S\}, \tag{5.2}$$

$$S^{-} = \{-e \mid (e = \{u, v\} \in E \setminus S) \wedge (\{u, v\} \cap \mathrm{dom}(S) \neq \emptyset)\}. \tag{5.3}$$

$S^{\pm}$ is a signed subgraph such that $\mathrm{abs}(S^{+})$ and $\mathrm{abs}(S^{-})$ are sets of edges satisfying Conditions 1 and 2 in Observation 1, respectively. Note that $\mathrm{abs}(S^{-})$ is

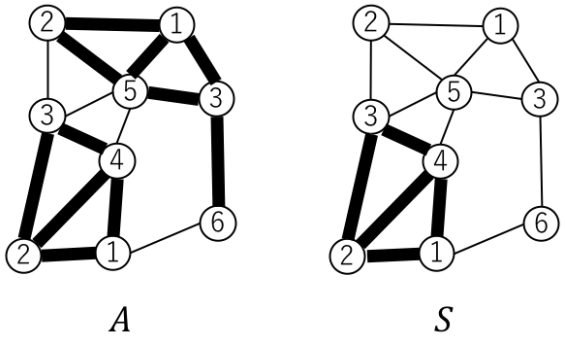

Figure 5.2: Graph partition and its connected component

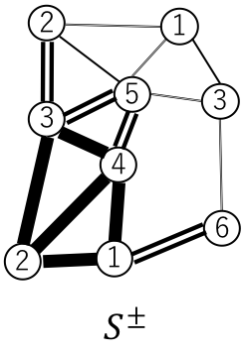

Figure 5.3: Signed subgraph with minimal cutset

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*. Figure 5.2 shows an example of a graph 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. Figure 5.3 shows $S^{\pm}$ associated with S in Fig. 5.2. 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. 5.2 indeed contains all the edges in $\mathrm{abs}(S^+)$ and does not

contain any edges in 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 abs(S^+) and no edge in 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 $Z_{\mathcal{A}} \setminus Z_{\mathcal{S}^\uparrow}$ [6].

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

5.4.2 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 Subsection 5.4.4.

We can construct $Z_{\mathcal{S}}$ using frontier-based search.We design an algorithm in a similar way to the algorithm 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` [13], 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 3. *The width of $Z_{\mathcal{S}}$ is $\mathcal{O}(B_f L)$, where $f = \max\{|F_i| \mid i \in [m]\}$.*

5.4.3 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 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 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 2.

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 $\texttt{colors}$, 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 7 and 8, respectively.

We give the following lemma on the width of a ZDD constructed by Algorithms 7 and 8.

Lemma 4. *The width W_T of a TDD constructed by Algorithms 7 and 8 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 [25] with Algorithms 7 and 8. 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 7. 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 $\alpha = \bot$, we return $\bot$ because we cannot ensure (a) anymore. Otherwise, we go on to Line 3 of Algorithm 7. 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 $Z_{\mathcal{S}}$.

5.4.4 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 Subsection 5.4.2. From Observation 1 and Eqs. (5.1)–(5.3), $\mathcal{S}^\uparrow$ can be written as

$$\mathcal{S}^\uparrow = \{E' \subseteq E \mid \exists S^\pm \in \mathcal{S}^\pm, (\forall + e \in S^\pm, e \in E') \wedge (\forall - e \in S^\pm, e \notin E')\}. \quad (5.4)$$

Using $T_{\mathcal{S}^\pm}$, we can construct $Z_{\mathcal{S}}$ by the algorithm of Suzuki et al. [28].

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.

5.5 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 [13]. 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 [25] and the `SAPPORO_BDD` library.* 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 [13], 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 Tab. 5.1. 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",

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

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 $P = \sum_{v \in V} p(v)$ [13]. 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 Tab. 5.2. 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 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 Tab. 5.3. 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 Tab. 5.2. 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.

Algorithm 7: 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.
1 Let e_i = {u, v}.
2 Copy n_i to n'_i.
3 foreach x ∈ {u, v} do
      // violates Condition 1 in Observation 2
4     if 0 ∈ n'_i.colors[x] and s = + then return ⊥
5     if + ∈ n'_i.colors[x] and s = 0 then return ⊥
6     if n'_i.colors[x] = {−} and s = 0 then
          // Reserve the vertices in the frontier which are
             connected to x by the processed edges.
7         n'_i ← Reserve(n'_i, N(E^{<i}, x) ∩ (F_{i−1} ∪ F_i))
8         if n'_i = ⊥ then return ⊥
9     if 0 ∈ n'_i.colors[x] and s = − then
10        n'_i ← Reserve(n'_i, e_i \ {x})
11        if n'_i = ⊥ then return ⊥
12    if n'_i.reserved[x] = 1 and s = 0 then
13        return ⊥
14    if n'_i.reserved[x] = 1 and s = + then
15        n'_i.reserved[x] ← 0          // The reservation is archived.
16    n'_i.colors[x] ← n'_i.colors[x] ∪ {s}
17 foreach x ∈ {u, v} do
18    if x ∉ F_i then
          // x is leaving the frontier.
19        if n'_i.reserved[x] = 1 and + ∉ n'_i.colors[x] then
              // Although x is reserved, no positive edges are
                 incident to x.
20            return ⊥
21        if n'_i.colors[x] = {−} then
              // Reserve the vertices in the frontier which are
                 connected to x by the processed edges.
22            n'_i ← Reserve(n'_i, N(E^{≤i}, x) ∩ (F_{i−1} ∪ F_i))
23            if n'_i = ⊥ then return ⊥
          // Delete the information about the vertices leaving the
             frontier.
24        n'_i.colors[x] ← {}
25        n'_i.reserved[x] ← 0
26 if i = m then
27    return ⊤                  // All the constraints are satisfied.
28 return n'_i
```

Algorithm 8: 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'.
```
1 Copy n' to n''.
2 **for** $x \in X$ **do**
```
   // We cannot reserve x if there is a zero edge incident to
      x.
```
3 **if** $0 \in n''.\texttt{colors}[x]$ **then return** $\perp$
```
   // Reserve x if there are no positive edges incident to x.
```
4 **if** $+ \notin n''.\texttt{colors}[x]$ **then** $n''.\texttt{reserved}[x] \leftarrow 1$
5 **return** n''

Table 5.1: Summary of input graphs and input graph partitions

				Induced partition		Forest		Rooted forest	
Name	n	m	k	$\lvert Z_{\mathcal{A}}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert Z_{\mathcal{A}}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert Z_{\mathcal{A}}\rvert$	$\lvert\mathcal{A}\rvert$
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 5.2: 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 5.3: 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

6 Conclusion

In this thesis, we have proposed algorithms to construct ZDDs representing two types of graph partitions. First, 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 a ZDD in a few minutes for input graphs with hundreds of edges. As future work, it is important to consider new constraints such as reliability of roads.

Second, we have proposed an algorithm for balanced graph partitions. We can design an algorithm for the upper-bound constraint, that is, weights of connected components are at most a given value, with a small modification of the algorithm for the shelter-capacity constraint which we have proposed in Chapter 4. However, it is difficult to extend the algorithm to the lower-bound constraint, that is, weights of connected components are at least a given value. Therefore, we proposed an efficient algorithm to construct a ZDD for the lower-bound constraint to enumerate balanced graph partitions. Experimental results show that our algorithm runs tens of times faster than the existing 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.

Acknowledgements

I am deeply grateful to Professor Shoji Kasahara for constructive comments and warm encouragement. I would like to thank Professor Minoru Ito for valuable suggestions on this research. I also thank Associate Professor Masahiro Sasabe for his insightful comments in laboratory meetings, which are essential to improve my study. I would like to express my gratitude to Associate Professor Takashi Horiyama from Saitama University. His comments are important to improve my algorithms. My deepest appreciation goes to Assistant Professor Jun Kawahara. He has spent much time discussing with me and reading my drafts. Without his patient instruction, my thesis would not be completed. I also thank our laboratory secretary Ms. Yoko Hashimoto for her kind support. I am thankful to students in Large-Scale Systems Management Laboratory for supporting me. Last, but not least, I would like to thank my family for supporting me spiritually and financially.

References

[1] 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.

[2] Nemoto Toshio and Hotta Keisuke. On the limits of the reduction in population disparity between single-member election districts in Japan (in Japanese). *Japanese Journal of Electorial Studies*, 20(226):136–147, 2005.

[3] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[4] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[5] Peter Sanders and Christian Schulz. High quality graph partitioning. *Graph Partitioning and Graph Clustering*, 588(1), 2012.

[6] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th ACM/IEEE design automation conference*, pages 272–277, 1993.

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

[8] 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 Transactions on Smart Grid*, 5(1):102–111, 2014.

[9] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(9):1773–1784, 2017.

[10] Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proc. of the 6th International Symposium on Algorithms and Computation (ISAAC)*, pages 224–233, 1995.

[11] 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.

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

[13] Jun Kawahara, Takashi Horiyama, Keisuke Hotta, and Shin-ichi Minato. Generating all patterns of graph partitions within a disparity bound. In *Proc. of the 11th International Conference and Workshops on Algorithms and Computation (WALCOM)*, pages 119–131, 2017.

[14] Danny Z Chen, Jinhee Chun, Naoki Katoh, and Takeshi Tokuyama. Efficient algorithms for approximating a multi-dimensional voxel terrain by a unimodal terrain. In *COCOON*, volume 3106, pages 238–248. Springer, 2004.

[15] 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*, 2018 (to appear).

[16] Yu Nakahata, Jun Kawahara, and Shoji Kasahara. Enumerating graph partitions without too small connected components using zero-suppressed binary and ternary decision diagrams. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[17] 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.

[18] 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*, pages 294–305. Springer International Publishing, Cham, 2017.

[19] Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. Exact computation of influence spread by binary decision diagrams. In *Proc. of the 26th International World Wide Conference (WWW)*, pages 947–956, 2017.

[20] Brian W Kernighan and Lin Shen. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[21] Vitaly Osipov and Peter Sanders. n-level graph partitioning. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 278–289, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. An efficient memetic algorithm for the graph partitioning problem. *Annals of Operations Research*, 191(1):1–22, Nov 2011.

[23] Una Benlic and Jin-Kao Hao. A multilevel memetic approach for improving graph k-partitions. *IEEE Transactions on Evolutionary Computation*, 15(5):624–642, 2011.

[24] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms – ESA 2011*, pages 469–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

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

[26] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[27] 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.

[28] Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Enumerating all subgraphs without forbidden induced subgraphs via multivalued decision diagrams. *CoRR*, 2018.

Publication List

Journal

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, The Institute of Electronics, Information and Communication Engineers, 2018.（査読有，採録決定済）

International Conference

1. Yu Nakahata, Jun Kawahara and Shoji Kasahara, "Enumerating Graph Partitions Without Too Small Connected Components Using Zero-suppressed Binary and Ternary Decision Diagrams," The 17th International Symposium on Experimental Algorithms (SEA 2018), Leibniz International Proceedings in Informatics (LIPIcs), 103(21):1–13, L'Aquila, Italy, June 2018.（口頭，査読有）

2. Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shoji Kasahara, "Enumerating All Rooted Shortest Path Forests Using Zero-suppressed Binary Decision Diagrams," The 20th Korea-Japan Joint Workshop on Algorithms and Computation (WAAC 2017), Seoul, Korea, August 2017.（口頭，査読無）

Domestic Conference

1. 中畑 裕，鈴木 浩史，石畠 正和，堀山 貴史，"フロンティア法による DAG の非巡回縮約の列挙"，2018 年度人工知能学会全国大会（第 32 回），鹿児島，6 月，2018 年.（口頭，査読無）

2. 中畑 裕，川原 純，笠原 正治，"グラフ分割集合を表す ZDD に対する連結成分重み下限制約"，情報処理学会第 166 回アルゴリズム研究会，沖縄，1 月，2018 年.（口頭，査読無）

3. 中畑 裕，鈴木 浩史，石畠 正和, 堀山 貴史，“フロンティア法による DAG の DAG への分解”，基盤 (S) 離散構造処理系プロジェクト「2017 年度秋のワークショップ」，北海道，11 月，2017 年.（ポスター，査読無）

4. 中畑 裕，川原 純，笠原 正治，“部分最短経路木分割の列挙”，基盤 (S) 離散構造処理系プロジェクト「2017 年度初夏のワークショップ」，北海道，7 月，2017 年.（ポスター，査読無）

5. 中畑 裕，川原 純，笠原 正治，“グラフの連結成分の大きさを考慮した連結成分分割の高速な列挙”，情報処理学会第 162 回アルゴリズム研究会，大分，3 月，2017 年.（口頭，査読無）