TCS-TR-A-17-81

TCS Technical Report

ZDD-Based Enumeration of Pareto-Optimal Solutions for 0-1 Multi-Objective Knapsack Problems

by

Hirofumi Suzuki and Shin-ichi Minato

Division of Computer Science

Report Series A

June 1, 2017

Hokkaido University
Graduate School of
Information Science and Technology

Email: h-suzuki@ist.hokudai.ac.jp

Phone: +81-011-706-6471

Fax: +81-011-706-6471

ZDD-Based Enumeration of Pareto-Optimal Solutions for 0-1 Multi-Objective Knapsack Problems

Hirofumi Suzuki
Grad. School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060–0814, Japan

Shin-ichi Minato
Grad. School of Info. Sci. and Tech.
Hokkaido University
Sapporo 060–0814, Japan

June 1, 2017

(Abstract) Finding pareto-optimal solutions is a basic approach in multi-objective combinatorial optimization. In this paper, we focus on 0-1 multi-objective knapsack problems, and present an algorithm to enumerate all pareto-optimal solutions of them, inspired by (Bazgan, Hugot, and Vanderpooten, Computers & OR, 36(1):260–279, 2009). Our algorithm is based on dynamic programming techniques using an efficient data structure, called zero-suppressed binary decision diagram (ZDD), which handles set of combinations compactly. In our algorithm, we utilize ZDDs for pruning inessential partial solutions. As an output of the algorithm, we can obtain a useful ZDD indexing all pareto-optimal solutions. The results of our experiments showed that our algorithm is clearly faster than previous method in several three-objective and four-objective instances, which are harder problems to be solved.

1 Introduction

In *multi-objective combinatorial optimization*, efficient solutions have the property called pareto-optimality, that any objective can not be improved whithout changing other objective for the worse. A solution with the pareto-optimality is called *pareto-optimal solution*. Several results and annotated bibliographies of multi-objective combinatorial optimization can be found in [7].

In this paper, we focus on the *0-1 multi-objective knapsack problem* which is the one of multi-objective combinatorial optimization problems. The single-objective version of this problem is well-studied in the literature [6]. Multi-objective cases have many practical applications for example capital budgeting [10], and selecting transportation investment alternatives [11].

To find a reduced set or the complete set of pareto-optimal solutions is a typical approach in multi-objective combinatorial optimization. In the 0-1 multi-objective knapsack problem, metaheuristics are better to find an approximately reduced set in many cases [5] [12]. Only for the bi-objective case, labeling algorithm [4] and ϵ-constraint method [3] can find the complete set. For general cases, the dynamic programming and some pruning based approach has been proposed [1]. Bazgan *et al.* showed that their approach is faster than labeling algorithm and ϵ-constraint method in bi-objective cases. Moreover, they conducted experiments to find the complete set in three-objective cases.

In this paper, we propose a more practical algorithm to find the complete set of pareto-optimal solutions inspired by the Bazgan's approach. The algorithm uses an efficient data structure, named *zero-suppressed binary decision diagram* (ZDD), which can index a huge number of combinations in compact form [8]. There are many practical applications of ZDDs in combinatorial optimization problems including the single-objective knapsack problem [2] [9]. We designed pruning heuristics using ZDDs which indexes all feasible solutions. It is more accurate than conventional methods. In addition, we can get the useful ZDD indexing all the pareto-optimal solutions as the output of our algorithm.

We conducted computational experiments to compare Bazgan's algorithm and our algorithm. The results of computation time showed that our algorithm is clearly faster than Bazgan's algorithm for various types of three-objective and four-objective instances, which are harder problems to be solved. We also investigated the compression efficiency of output ZDDs.

This paper is organized as follows. In section 2, we formally define the 0-1 multi-objective knapsack problem and describe ZDDs. Section 3 presents the framework of our algorithm. Pruning conditions using our algorithm are described in section 4. Section 5 presents the algorithmic issues. Computational experiments and results are reported in section 6. In section 7, we conclude this paper.

2 Preliminaries

2.1 0-1 Multi-objective Knapsack Problem

The 0-1 multi-objective knapsack problem is known as one of multi-objective combinatorial optimization problems. Given n items $I = \{1, \dots, n\}$ and an integer capacity $C > 0$. In m-objective cases, each item $i \in I$ has an integer weight $w_i > 0$, and a m-dimensional non-negative integer-valued vector $\boldsymbol{v}^i = (v^i_1, \dots, v^i_m)$. A solution of the problem is a combination of items $X \subseteq I$. The total weight of a solution X is denoted by $w(X) = \sum_{i \in X} w_i$. If X is a feasible solution, it satisfies $w(X) \leq C$. We define the set of all the feasible solutions $\mathcal{X} = \{X \subseteq I | w(X) \leq C\}$.

The value of a solution X is denoted by the sum of the valued vectors $\boldsymbol{v}(X) = \sum_{i \in X} \boldsymbol{v}^i$. Let us define the order relation $<$ on valued vectors as follows:

$$\boldsymbol{a} < \boldsymbol{b} \iff \begin{cases} a_i \leq b_i, \ \forall i \in \{1, \dots, m\}, \text{ and} \\ a_j < b_j, \ \exists j \in \{1, \dots, m\}. \end{cases} \tag{1}$$

We also define a partial order relation Δ on $\mathcal{X}$, called *dominance relation*, using the order relations $<$ as follows:

$$\forall X, X' \in \mathcal{X},\ X \Delta X' \iff \boldsymbol{v}(X) < \boldsymbol{v}(X'). \tag{2}$$

If $X \Delta X'$, we say that X is dominated by X' or X' dominates X. On the dominance relation, a solution $X \in \mathcal{X}$ is called pareto-optimal solution, if X is not dominated by any feasible solution $X' \in \mathcal{X}$. Let $\mathcal{P}$ be the set of all the pareto-optimal solutions. A typical goal of the problem is to find a reduced set $\mathcal{P}' \subseteq \mathcal{P}$ or the complete set $\mathcal{P}$. In this paper, our goal is to find the complete set $\mathcal{P}$.

2.2 Zero-suppressed Binary Decision Diagram

Zero-suppressed binary decision diagram (ZDD) is a data structure to represent a set of combinations in compact form. A ZDD is a directed acyclic graph $\mathcal{Z} = (\mathcal{N}, \mathcal{A})$ with node set $\mathcal{N}$ and arc set $\mathcal{A}$. It has two terminal nodes $\top$ and $\bot$. Each non-terminal node $\alpha \in \mathcal{N}$ has two arcs 0-arc and 1-arc. A node pointed by x-arc of α is denoted by α_x, and called x-child of α. A ZDD has a root node ρ which is not pointed by any arcs. In 0-1 multi-objective knapsack problems, we manipulate the set of items I with ordering as $1 < \ldots < n$. Each non-terminal node $\alpha \in \mathcal{N}$ has a label $\ell(\alpha) \in I$ where $\ell(\alpha) < \ell(\alpha_x)$ $(x \in \{0, 1\})$. We suppose that $\ell(\top) = \ell(\bot) = n + 1$ for convenience.

A path from a node α to the terminal $\top$ represents a combination $X \subseteq I$. Item $i \in I$ belongs to X iff the path has a 1-arc which goes out a node labeled with i. We define the set of combinations represented by a node $\alpha \in \mathcal{N}$ as follows:

$$\mathcal{X}(\alpha) = \{X \subseteq I \mid \text{A path } \alpha \text{ to } \top \text{ corresponds to } X\}. \tag{3}$$

Especially, $\mathcal{X}(\rho)$ is the whole set represented by $\mathcal{Z}$.

ZDDs can be reduced by the following two rules.

1. Share any isomorphic subgraphs.
2. Delete all nodes whose 1-arc points $\bot$.

We can use the above two rules in any order. Then the form of the ZDD is uniquely determined, and we call it *irreducible ZDD* (show Fig. 1). We suppose that ZDD is irreducible unless otherwise noted. Note that any irreducible ZDD for $\mathcal{X}$ has no arc which points $\bot$, and we can construct it in $O(nC)$ time [2].

3 Framework of Dynamic Programming over ZDDs

3.1 Basic Concept

Let $\mathcal{Z} = (\mathcal{N}, \mathcal{A})$ be the ZDD with a root node ρ where $\mathcal{X}(\rho) = \mathcal{X}$. Then we try to extract the set $\mathcal{P}$ from $\mathcal{Z}$ by the dynamic programming (DP). In the DP, we

i	w_i
1	2
2	2
3	3

$C = 5$

$\chi = \{\phi, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}\}$

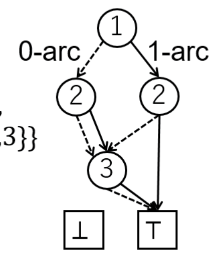

Figure 1: An example of ZDD for a 0-1 multi-objective knapsack problem

manage sets of states corresponding to feasible solutions. A state s consists of a m-dimentional valued vector $s.\boldsymbol{v}$ and a node $s.\alpha \in \mathcal{N}$.

The DP consists of n phases. Let S_i be a set of states that each state $s \in S_i$ satisfies $\ell(s.\alpha) = i + 1$. Then S_i corresponds to a set of solutions consisting of the first i items $\{1, \ldots, i\}$. Initially, let $S_0 = \{s_\phi\}$ where $s_\phi.\boldsymbol{v} = \mathbf{0}$, and $s_\phi.\alpha = \rho$. It represents the empty knapsack. At the phase i we extend all the states $s \in S_{i-1}$ as follows: For all $b \in \{0, 1\}$, we generate a state s_b where $s_b.\boldsymbol{v} = s.\boldsymbol{v} + b \times \boldsymbol{v}^i$, and $s_b.\alpha = s.\alpha_b$. After that s_b is added to the set of states S_j where $j = \ell(s_b.\alpha) - 1$. Note that, it is not necessary to maintain the capacity, because all the transitions on $\mathcal{Z}$ generate a feasible solution.

In this process, many states will not become any pareto-optimal solution. Thus we conduct a pruning of them as much as possible for obtaining S_n which corresponds to $\mathcal{P}$.

3.2 Pruning Condition

Because of the property that $\mathcal{X}(\rho) = \mathcal{X}$, the set of all the combinations of items which can be added to a state s without violation is represented by $\mathcal{X}(s.\alpha)$. Let us define the set of valued vectors $\mathcal{E}(s) = \{s.\boldsymbol{v} + \boldsymbol{v}(X) \mid X \in \mathcal{X}(s.\alpha)\}$. It corresponds to the set of all the feasible solutions extended from s.

The pruning process uses d relations $\Delta_1, \ldots, \Delta_d$ which is similar to the dominance relation. Each relation Δ_k $(k = 1, \ldots, d)$ must have the following property:

$$\forall s, s' \in S_i \ (i = 1, \ldots, n), \ s\Delta_k s' \Rightarrow \forall \boldsymbol{e} \in \mathcal{E}(s), \ \exists \boldsymbol{e}' \in \mathcal{E}(s'), \ \boldsymbol{e} < \boldsymbol{e}'. \qquad (4)$$

$s\Delta_k s'$ means that all the feasible solutions extended from s is dominated by another solution extended from s'. We call such relations as *future dominance relation.* According to the d future dominance relations, we can conduct a pruning as follows: For a state $s \in S_i$, if there is a state $s' \in S_i$ and an integer $k \in \{1, \ldots, d\}$ where $s\Delta_k s'$, we delete s from S_i.

When at least one relation Δ_k satisfies

$$\forall s, s' \in S_n, \ s\Delta_k s' \iff s.\boldsymbol{v} < s'.\boldsymbol{v}, \qquad (5)$$

S_n becomes the set of valued vectors of $\mathcal{P}$. In the next section, we define two future dominance relations used in our algorithm.

4 Future Dominance Relations Using ZDDs

4.1 Relation Based on Implicit Capacities

The first future dominance relation is based on the following observation. For two states $s, s' \in S_i$ $(i = 1, \ldots, n)$, we suppose that $s.\boldsymbol{v} < s'.\boldsymbol{v}$ and $\mathcal{X}(s.\alpha) \subseteq \mathcal{X}(s'.\alpha)$. For all $X \in \mathcal{X}(s.\alpha)$, $\boldsymbol{e} = s.\boldsymbol{v} + \boldsymbol{v}(X) \in \mathcal{E}(s)$. Also, $\boldsymbol{e}' = s'.\boldsymbol{v} + \boldsymbol{v}(X) \in \mathcal{E}(s')$, because $X \in \mathcal{X}(s'.\alpha)$. Then $\boldsymbol{e} < \boldsymbol{e}'$. This situation matches the condition (4). Thus we define the future dominance relation Δ_c on S_i $(i = 1, \ldots, n)$ by:

$$\forall s, s' \in S_i,\ s \Delta_c s' \iff s.\boldsymbol{v} < s'.\boldsymbol{v},\ \text{and}\ \mathcal{X}(s.\alpha) \subseteq \mathcal{X}(s'.\alpha). \tag{6}$$

Δ_c also satisfies (5), because $\mathcal{X}(s.\alpha) = \mathcal{X}(s'.\alpha) = \mathcal{X}(\top) = \phi$ for $i = n$.

We can check $\mathcal{X}(s.\alpha) \subseteq \mathcal{X}(s'.\alpha)$ easily as follows: For a node α, let us define the maximum weight on $\mathcal{X}(\alpha)$ as $c(\alpha) = \max\{w(X) \mid X \in \mathcal{X}(\alpha)\}$. Then $c(s.\alpha)$ denotes the remaining capacity of a state s implicitly. Therefore $c(s.\alpha) \leq c(s'.\alpha)$ and $\mathcal{X}(s.\alpha) \subseteq \mathcal{X}(s'.\alpha)$ are equivalent. For calculating $c(\alpha)$, we can use the recursion formula defined by:

$$c(\top) = 0,\ c(\alpha) = \max\{c(\alpha_0), c(\alpha_1) + w_{\ell(\alpha)}\}. \tag{7}$$

Indeed, $c(\top)$ must be 0 obviously. When $c(\alpha_0)$ equals the maximum weight in all the combinations which leaves the item $\ell(\alpha)$, and $c(\alpha_1)$ is the opposite, $c(\alpha)$ must be the maximum of $c(\alpha_0)$ and $c(\alpha_1) + w_{\ell(\alpha)}$.

4.2 Relation Based on Upper Bound

The second future dominance relation is based on the comparison between an upper bound of a state and a greedy extension of another state.

Let us define a process for generating a greedy extension of a state s as follows: We start at the node $\beta = s.\alpha$. While $\beta \neq \top$, we trace 1-arc and update $\beta = \beta_1$. According to this process, we trace 1-arcs as much as possible, i.e. we try to take items while the capacity is enough. Let $\boldsymbol{g}(\alpha)$ be the valued vector obtained by the above process starting at a node α. It can be represented by the recursion formula defined by:

$$\boldsymbol{g}(\top) = \boldsymbol{0},\ \boldsymbol{g}(\alpha) = \boldsymbol{g}(\alpha_1) + \boldsymbol{v}^{\ell(\alpha)}. \tag{8}$$

The valued vector of a greedy extension of a state s is $s.\boldsymbol{v} + \boldsymbol{g}(s.\alpha) \in \mathcal{E}(s)$.

For a node α, let us define an upper bound on $\mathcal{X}(\alpha)$ as follows: a valued vector $\boldsymbol{u}$ is an upper bound on $\mathcal{X}(\alpha)$ iff $\max\{v_i(X) \mid X \in \mathcal{X}(\alpha)\} \leq u_i$ $(i = 1, \ldots, m)$.

Let $\boldsymbol{u}(\alpha)$ be the valued vector which is the tight upper bound on $\mathcal{X}(\alpha)$. We can calculate $\boldsymbol{u}(\alpha)$ by the recursion formula defined by:

$$\boldsymbol{u}(\top) = \mathbf{0},\ u_i(\alpha) = \max\{u_i(\alpha_0), u_i(\alpha_1) + v_i^{\ell(\alpha)}\}\ (i = 1, \ldots, m). \tag{9}$$

Indeed, $\mathbf{0}$ is the tight upper bound on $\mathcal{X}(\top) = \phi$ obviously. When $\boldsymbol{u}(\alpha_b)$ is the tight upper bound on $\mathcal{X}(\alpha_b)$ $(b \in \{0, 1\})$, $u_i(\alpha)$ should be the maximum of $u_i(\alpha_0)$ and $u_i(\alpha_1) + v_i^{\ell(\alpha)}$ for each $i = 1, \ldots, m$. The valued vector of an upper bound calculated from a state s is $s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha)$.

Using greedy extensions and upper bounds, we can construct a future dominance relation as follows: For two states $s, s' \in S_i$ $(i = 1, \ldots, n)$, if $s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha) < s'.\boldsymbol{v} + \boldsymbol{g}(s'.\alpha)$, then $\boldsymbol{e} < s'.\boldsymbol{v} + \boldsymbol{g}(s'.\alpha)$ for all $\boldsymbol{e} \in \mathcal{E}(s)$, because $e_i \leq s.v_i + u_i(s.\alpha)$ for each $i = 1, \ldots, m$. This situation matches the condition (4). Thus we define the future dominance relation Δ_u on S_i $(i = 1, \ldots, n)$ by:

$$\forall s, s' \in S_i,\ s_i \Delta_u s'_i \iff s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha) < s'.\boldsymbol{v} + \boldsymbol{g}(s'.\alpha). \tag{10}$$

5 Algorithm

5.1 Preprocess

For evaluating the future dominance relations Δ_c and Δ_u, we can calculate $c(\alpha)$, $\boldsymbol{g}(\alpha)$, and $\boldsymbol{u}(\alpha)$ for all nodes $\alpha \in \mathcal{N}$ beforehand. We conduct the bottom-up DP with recursion formula (7), (8), and (9). Initially, we set $c(\top) \leftarrow 0$, $\boldsymbol{g}(\top) \leftarrow \mathbf{0}$, and $\boldsymbol{u}(\top) \leftarrow \mathbf{0}$. After that we process the nodes in reverse topological order (i.e. $\top$ to ρ). The pseudo code of preprocess is represented in Algorithm 1.

Algorithm 1 Preprocess for calculating $c(\alpha)$, $\boldsymbol{g}(\alpha)$, and $\boldsymbol{u}(\alpha)$

```
1: $c(\top) \leftarrow 0$, $\boldsymbol{g}(\top) \leftarrow \mathbf{0}$, $\boldsymbol{u}(\top) \leftarrow \mathbf{0}$
2: for $\alpha \in \mathcal{N}$ in the reverse topological order do
3:     $j \leftarrow \ell(\alpha)$
4:     $c(\alpha) \leftarrow \max\{c(\alpha_0), c(\alpha_1) + w_j\}$
5:     $\boldsymbol{g}(\alpha) \leftarrow \boldsymbol{g}(\alpha_1) + \boldsymbol{v}^j$
6:     $u_i(\alpha) \leftarrow \max\{u_i(\alpha_0), u_i(\alpha_1) + v_i^j\}$ for $i = 1, \ldots, m$
7: end for
```

5.2 Generating ZDD for $\mathcal{P}$

For generating a ZDD for $\mathcal{P}$, we should store the copy of all the states and the transitions in the DP. We start at a ZDD with only a root node. After that, every time a transition from a state is occured in the DP, we create a new arc and a new child node which corresponds to the transition. We show the details of this process in the following.

Let $\mathcal{N}'$ be a set of new generated nodes, and $\mathcal{A}'$ be a set of new generated arcs. Let $\mathcal{N}'(s)$ be a node which corresponds to a state s. For the initial state s_ϕ, we create a root node ρ' and set $\mathcal{N}'(s_\phi) \leftarrow \rho'$. When we create a state s_b ($b \in \{0,1\}$) by the transition from a state s, if s_b is not already created in the DP, we also create a new node β where $\ell(\beta) = \ell(s_b.\alpha)$, and set $\mathcal{N}'(s_b) \leftarrow \beta$. At that time, β becomes a b-child of $\mathcal{N}'(s)$. Then we create a b-arc which points $\mathcal{N}'(s)$ to β, and add it to $\mathcal{A}'$.

In the pruning phase, if a state s is pruned, we update $\mathcal{N}'(s) \leftarrow \bot$. As a special case, when we conduct a transition from a state s where $s.\alpha_0 = s.\alpha_1$, we generate only s_1 and prune s_0 immediately, because $c(s_0.\alpha) = c(s_1.\alpha)$ and $s_0.\boldsymbol{v} < s_1.\boldsymbol{v} = s_0.\boldsymbol{v} + \boldsymbol{v}^{\ell(s.\alpha)}$ (i.e. $s_0 \Delta_c s_1$). At the end of the phase n we update $\mathcal{N}'(s) = \top$ for all $s \in S_n$. This process can store all the states and the transition of the DP as a ZDD form. Then we get a ZDD $\mathcal{Z}' = (\mathcal{N}', \mathcal{A}')$ where $\mathcal{X}(\rho') = \mathcal{P}$. However, $\mathcal{Z}'$ is not irreducible in allmost all cases, thus we should reduce it.

The pseudo code of the above process is represented in Algorithm 2. For the pruning phase, we design the function `pruneDominatedStates` in the following.

Algorithm 2 Enumeration of all the pareto-optimal solutions

```
1: Create ZDD $\mathcal{Z} = (\mathcal{N}, \mathcal{A})$ for $\mathcal{X}$ with a root node $\rho$, and use Algorithm 1
2: Set $\mathcal{N}' \leftarrow \phi$ and $\mathcal{A}' \leftarrow \phi$
3: $s_\phi.\boldsymbol{v} \leftarrow \mathbf{0}$, $s_\phi.\alpha \leftarrow \rho$, $S_0 \leftarrow \{s_\phi\}$, $\mathcal{N}'(s_\phi) \leftarrow \rho'$
4: for $i = 1, \ldots, n$ do
5:   for $s \in S_{i-1}$ do
6:     for $b \in \{0,1\}$ do
7:       if $b = 0$ and $s.\alpha_0 = s.\alpha_1$ then
8:         $\beta \leftarrow \mathcal{N}'(s)$, $\beta_b \leftarrow \bot$
9:       else
10:        Create $s_b$, and set $s_b.\boldsymbol{v} \leftarrow s.\boldsymbol{v} + b \times \boldsymbol{v}^i$, $s_b.\alpha \leftarrow s.\alpha_b$
11:        $j \leftarrow \ell(s_b.\alpha) - 1$
12:        if $s_b \in S_j$ then
13:          Create a $b$-arc which ponits $\mathcal{N}'(s)$ to $\mathcal{N}'(s_b)$ and add it to $\mathcal{A}'$
14:        else
15:          Create a new node $\beta$ where $\ell(\beta) = \ell(s_b.\alpha)$
16:          $\mathcal{N}'(s_b) \leftarrow \beta$, $S_j \leftarrow S_j \cup \{s_b\}$
17:          Create a $b$-arc which points $\mathcal{N}'(s)$ to $\beta$, and add it to $\mathcal{A}'$
18:        end if
19:      end if
20:    end for
21:  end for
22:  pruneDominatedStates($S_i$, $\mathcal{N}'$)
23: end for
24: $\mathcal{N}'(s) \leftarrow \top$ for all $s \in S_n$
25: Create $\mathcal{Z}' = (\mathcal{N}', \mathcal{A}')$ and apply the reduction rules to $\mathcal{Z}'$
26: return $\mathcal{Z}'$
```

5.3 Pruning Dominated States

Here we discuss about how to apply the future dominance relations Δ_c and Δ_u. It is important that we make the pruning process faster as much as possible. Indeed, pruning is the process to take computation time most. For a simple way, we will compare all pairs of states in $O(m|S_i|^2)$ time at each phase $i = 1, \dots, n$. We present some techniques for accelerating the simple process in the following.

Let $\geq_{lex}$ be the reverse lexicographical relation defined on valued vectors as follows (suppose that $\boldsymbol{a}$ and $\boldsymbol{b}$ are a m-dimensional valued vector respectively):

$$\boldsymbol{a} \geq_{lex} \boldsymbol{b} \iff \begin{cases} \exists i \in \{1, \dots, m\},\ a_j = b_j\ (j < i) \text{ and } a_i > b_i, \text{ or} \\ \forall i \in \{1, \dots, m\},\ a_i = b_i. \end{cases} \tag{11}$$

Let $\geq_c$ be the order relation defined on S_i $(i = 1, \dots, n)$ by:

$$\forall s, s' \in S_i,\ s \geq_c s' \iff \begin{cases} c(s.\alpha) > c(s'.\alpha), \text{ or} \\ c(s.\alpha) = c(s'.\alpha) \text{ and } s.\boldsymbol{v} \geq_{lex} s'.\boldsymbol{v}. \end{cases} \tag{12}$$

If $s \geq_c s'$, then s is not dominated by s' on Δ_c, because $c(s.\alpha) > c(s'.\alpha)$ or $s'.\boldsymbol{v} < s.\boldsymbol{v}$ is satisfied (note that $s.\boldsymbol{v} \geq_{lex} s'.\boldsymbol{v} \Rightarrow s'.\boldsymbol{v} < s.\boldsymbol{v}$). Thus, at the pruning phase for S_i, we sort all the states in S_i by the decreasing order on $\geq_c$. After that we examine whether $s \in S_i$ is dominated on Δ_c or not in the sorted order. At that time, we focus on dominant valued vectors as follows.

Let D be the list of dominant valued vectors with respect to $s.\boldsymbol{v}$ where $s \in S_i$. Initially, we set $D = \phi$. After that we update D, and perform a pruning by Δ_c as follows: For each state $s \in S_i$, we compare $s.\boldsymbol{v}$ and each valued vector $\boldsymbol{d} \in D$. If $s.\boldsymbol{v} < \boldsymbol{d}$, then s is dominated by another state whose valued vector is $\boldsymbol{d}$, and we delete s from S_i. Otherwise, when $\boldsymbol{d} \neq s.\boldsymbol{v}$, we add $s.\boldsymbol{v}$ to D and delete $\boldsymbol{d} \in D$ where $\boldsymbol{d} < s.\boldsymbol{v}$. This process is performed in $O(m|D|)$ time. Note that $|D| \leq |S_i|$.

Moreover, to finish the comparison as soon as possible, we sort the list D by the decreasing order on $\geq_{lex}$. We suppose that $D = \{\boldsymbol{d}^1, \dots, \boldsymbol{d}^{|D|}\}$, and $\boldsymbol{d}^j \geq_{lex} \boldsymbol{d}^k$ $(j < k)$ in the following. We compare $s.\boldsymbol{v}$ and $\boldsymbol{d}^j$ in order of $j = 1, \dots, |D|$. When $s.\boldsymbol{v} \geq_{lex} \boldsymbol{d}^j$, we can stop the comparison, because $\boldsymbol{d}^j$ is the last element with possibility to dominate s. After that, when $\boldsymbol{d}^j \neq s.\boldsymbol{v}$, we insert $s.\boldsymbol{v}$ to the j-th position of D, and delete $\boldsymbol{d}^k$ $(j \leq k)$ where $\boldsymbol{d}^k < s.\boldsymbol{v}$. Note that D is still sorted by the decreasing order on $\geq_{lex}$ after the insertion and the deletion.

The algorithm to apply Δ_c is represented in the first 6 lines of Algorithm 3. The function `updateDominator` represented in Algorithm 4 is used to update D and examine whether a state is dominated or not.

For applying Δ_u, we can use the similar techniques of applying Δ_c. Let G be the list of valued vectors with respect to $s.\boldsymbol{v} + \boldsymbol{g}(s.\alpha)$ where $s \in S_i$. We can generate G by applying `updateDominator`$(G,\ s.\boldsymbol{v} + \boldsymbol{g}(s.\alpha))$ for all $s \in S_i$. Then G is sorted by the decreasing order on $\geq_{lex}$ and have no unnecessary valued vector. We suppose that $G = \{\boldsymbol{g}^1, \ldots, \boldsymbol{g}^{|G|}\}$, and $\boldsymbol{g}^j \geq_{lex} \boldsymbol{g}^k$ $(j < k)$ in the following. Then we perform a pruning by Δ_u as follows: For a state $s \in S_i$, let $\boldsymbol{u} = s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha)$. We compare $\boldsymbol{u}$ and $\boldsymbol{g}^j$ in order of $j = 1, \ldots, |G|$. If $\boldsymbol{u} \geq_{lex} \boldsymbol{g}^j$, we stop the comparison. Otherwise, when $\boldsymbol{u} < \boldsymbol{g}^j$, we delete s from S_i. The algorithm to apply Δ_u is represented in the remaining lines of Algorithm 3.

Especially, at n-th pruning phase, S_n corresponds to $\mathcal{P}$ after applying Δ_c according to the property (5). Thus we do not apply Δ_u at n-th pruning phase.

Algorithm 3 `pruneDominatedStates`(S_i, $\mathcal{N}'$)

```
1: Create the empty list $D$
2: for $s \in S_i$ in the decreasing order on $\geq_c$ do
3:   if updateDominator($D$, $s.\boldsymbol{v}$) = "dominated" then
4:     $\mathcal{N}'(s) = \perp$, $S_i \leftarrow S_i \setminus \{s\}$
5:   end if
6: end for
7: if $i = n$ then
8:   return
9: end if
10: Create the empty list $G$
11: updateDominator($G$, $s.\boldsymbol{v} + \boldsymbol{g}(s.\alpha)$) for all $s \in S_i$
12: for $s \in S_i$ do
13:   $j \leftarrow 1$ /* $G = \{\boldsymbol{g}^1, \ldots, \boldsymbol{g}^{|G|}\}$ where $\boldsymbol{g}^j \geq_{lex} \boldsymbol{g}^k$ $(j < k)$ */
14:   while $j \leq |G|$ and $\boldsymbol{g}^j \geq_{lex} s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha)$ do
15:     if $s.\boldsymbol{v} + \boldsymbol{u}(s.\alpha) < \boldsymbol{g}^j$ then
16:       $\mathcal{N}'(s) = \perp$, $S_i \leftarrow S_i \setminus \{s\}$
17:       break
18:     end if
19:     $j \leftarrow j + 1$
20:   end while
21: end for
```

5.4 Reordering Heuristics

The order of items is an important issue for single-objective knapsack problems. It is well-known that the decreasing order with respect to v_1^i/w_i, i.e. value per unit weight, is better to obtain a good solution. For the multi-objective version, Bazgan *et al.* proposed the order $\mathcal{O}^{max}$, which refers to the ranking of items with respect to each objective. For the details, please refer to [1].

Here we propose a new order which is a natural expansion of single-objective cases as follows: Let $\boldsymbol{p}^i$ be the *potential vector* defined by $\{v_1^i/w_i, \ldots, v_m^i/w_i\}$ for each $i \in I$. We define the $\mathcal{O}^{pot}$ as the decreasing order of the potential vectors on $\geq_{lex}$. Moreover, we define the $\widetilde{\mathcal{O}^{pot}}$ as the reverse order of $\mathcal{O}^{pot}$.

Algorithm 4 `updateDominator`(DOM, $\boldsymbol{v}$)

```
1: $j \leftarrow 1$ /* $DOM = \{\boldsymbol{d}^1, \dots, \boldsymbol{d}^{|DOM|}\}$ where $\boldsymbol{d}^j \geq_{lex} \boldsymbol{d}^k$ $(j < k)$ */
2: while $j <= |DOM|$ and $\boldsymbol{d}^j \geq_{lex} \boldsymbol{v}$ do
3:   if $\boldsymbol{v} < \boldsymbol{d}^j$ then
4:     return "dominated"
5:   else if $\boldsymbol{v} = \boldsymbol{d}^j$ then
6:     return "non dominated"
7:   end if
8:   $j \leftarrow j + 1$
9: end while
10: Insert $\boldsymbol{v}$ at the $j$-th position in $DOM$
11: Erase $\boldsymbol{d}^k$ from $DOM$ where $\boldsymbol{d}^k < \boldsymbol{v}$ $(j < k)$
12: return "non dominated"
```

6 Computational experiments and results

All code was implemented in C++ (g++5.4.0 with the -O3 option). We used 64-bit Ubuntu 16.04 LTS with an Intel Core i7-3930K 3.2 GHz CPU and 64 GB RAM. All instances satisfy $10 \leq v^i_j, w_i \leq 100$ $(i = 1, \dots, n,\ j = 1, \dots, m)$ and $C = 10n$. The following types of instances were considered.

- Type 1: All values and weights are decided uniformly at random.
- Type 2: All valued vectors satisfy $50m - 10 \leq \sum_{j=1}^{m} v^i_j \leq 50m + 10$ $(i = 1, \dots, n)$. This causes negative correlation between the objectives. All weights are decided uniformly at random.
- Type 3: All values are decided uniformly at random. All weights satisfy $\frac{1}{m}\sum_{j=1}^{m} v^i_j - 10 \leq w_i \leq \frac{1}{m}\sum_{j=1}^{m} v^i_j + 10$. This causes positive correlation between values and the weights.

6.1 Algorithmic Efficiency

We conducted comparison between our algorithm and Bazgan's algorithm. Bazgan's algorithm is also based on a DP with pruning. It does not use any supporting data structures, and does not generate any index of $\mathcal{P}$.

6.1.1 Order of Items

First we compared the three orders presented in section 5.4 ($\mathcal{O}^{max}$, $\mathcal{O}^{pot}$, and $\mathcal{O}^{\widetilde{pot}}$) and random order on 100 instances of each type. Table 1 clearly shows that $\mathcal{O}^{pot}$ accelerates our algorithm. However, Bazgan's algorithm have nothing worthy of special mention. Thus, in the following, we use $\mathcal{O}^{pot}$ for reordering of items.

Table 1: Average computation time (sec) on each reordering

			Bazgan's				Proposal			
Type	n	m	random	$\mathcal{O}^{max}$	$\mathcal{O}^{pot}$	$\widetilde{\mathcal{O}^{pot}}$	random	$\mathcal{O}^{max}$	$\mathcal{O}^{pot}$	$\widetilde{\mathcal{O}^{pot}}$
	200	2	5.31	**3.05**	4.32	9.79	10.78	5.81	**4.35**	12.96
1	100	3	**18.42**	19.64	19.17	59.57	15.98	11.85	**7.59**	28.92
	70	4	**19.20**	29.03	21.89	81.88	9.84	7.34	**5.59**	30.77
	150	2	7.16	6.15	**6.02**	8.42	11.86	10.37	**6.63**	9.99
2	60	3	**10.81**	16.06	12.94	20.47	5.84	7.78	**3.69**	10.12
	50	4	**70.17**	105.25	**70.17**	126.93	17.12	25.08	**9.71**	40.93
	120	2	18.37	**9.24**	11.60	9.89	35.31	19.55	18.82	**18.22**
3	50	3	52.91	40.81	41.47	**36.29**	31.30	23.38	**20.95**	22.83
	35	4	33.39	29.91	29.26	**26.27**	7.50	6.43	**5.96**	6.71

6.1.2 Computation Times

Next we investigated the number of generated states to evaluate an effect of the pruning. According to the table 2, our algorithm generates less states than Bazgan's algorithm for each case. There are up to 4 times differences. These results show that our pruning methods using ZDDs are quite effective. This impact is connected directly with computation times.

Table 2: Average number of states generated by each algorithm

		$m = 2$			$m = 3$			$m = 4$	
Type	n	Bazgan's	Proposal	n	Bazgan's	Proposal	n	Bazgan's	Proposal
1	200	5632649	2689570	100	3101540	1184175	70	1531573	465897
2	150	5468682	3052779	60	1097170	349233	50	1120602	282585
3	120	7519692	5432447	50	2035120	831832	35	669268	165721

Fig. 2 shows the computation times of our algorithm and Bazgan's algorithm for each instance. In many bi-objective instances, the overhead of constructing ZDDs in our algorithm has appeared greater than the effect of the pruning, because all the bi-objective cases are relatively easy to be solved. On the other hand, our algorithm is clearly faster than Bazgan's algorithm in the three-objective and four-objective cases. There are 2 to 5 times differences in almost all instances. These results show that our algorithm is efficient for harder problems to be solved.

6.2 Efficiency of Indexing

At last we show the number of the pareto-optimal solutions and the size of output ZDDs for each case. Table 3 suggests that pareto-optimal solutions suddenly

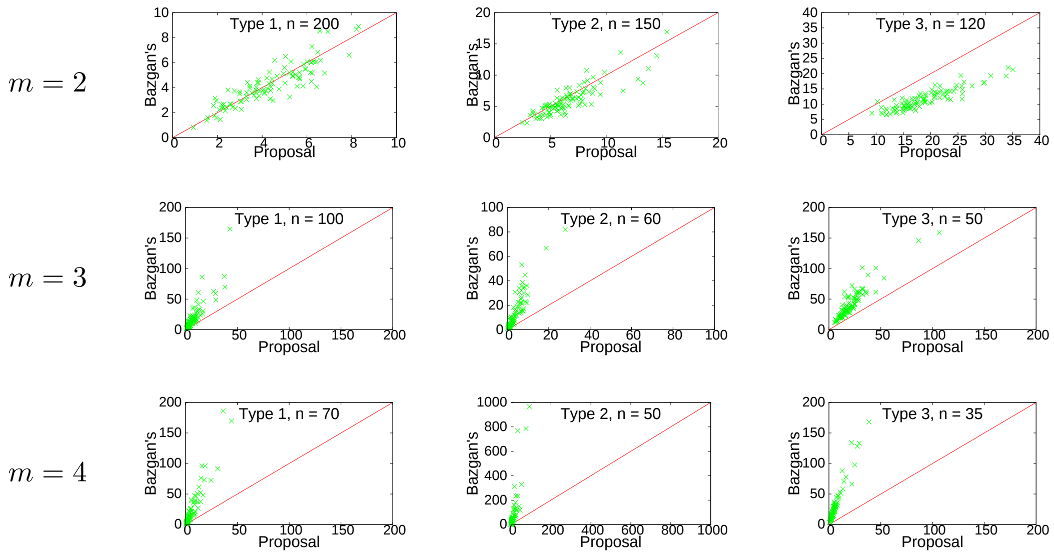

Figure 2: Computation times (sec) for each instance

increase as objectives increase. Each pareto-optimal solution has approximately $\frac{n}{5}$ items under the influence of how to generate instances. Thus, the size of the output ZDDs is small enough for the total number of items in all the pareto-optimal solutions.

Table 3: Average number of pareto-optimal solutions and output ZDD nodes

		$m = 2$			$m = 3$			$m = 4$	
Type	n	#pareto	#nodes	n	#pareto	#node	n	#pareto	#node
1	200	182	1993	100	978	2933	70	1674	2904
2	150	310	2485	60	1494	2386	50	4201	3607
3	120	511	3272	50	3026	4377	35	5227	4319

7 Conclusion

In this study, we have proposed an algorithm to enumerate pareto-optimal solutions of 0-1 multi-objective knapsack problems. The proposed algorithm is based on the DP with supporting ZDD, and construct a ZDD indexing all pareto-optimal solutions. The results showed that the proposed algorithm is faster than a conventional DP in several harder three-objective and four-objective instances. Moreover, the efficiency of indexing pareto-optimal solutions is better for the total size of all the pareto-optimal solutions.

All the main ideas in this study will be applied to multi-objective combinatorial optimization problems with linear objective functions. Thus, an important future work is to construct a generalized method for enumerating pareto-optimal solutions. Also, it is important to make use of ZDDs indexing pareto-optimal solutions in real world problems.

Acknowledgments

Our work is partly supported by JSPS KAKENHI Scientific Research(S) - Number 15H05711.

References

[1] Cristina Bazgan, Hadrien Hugot, and Daniel Vanderpooten. Solving efficiently the 0-1 multi-objective knapsack problem. *Computers & OR*, 36(1):260–279, 2009.

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

[3] V. Chankong and Y. Y. Haimes. *Multiobjective decision making.* Elsevier Science Publishing, New York (USA), 1983.

[4] Captivo M. Eugénia, Clìmaco Jo ao, Figueira José, Martins Ernesto, and Santos José Luis. Solving bicriteria 0-1 knapsack problems using a labeling algorithm. *Comput. Oper. Res.*, 30(12), October 2003.

[5] Xavier Gandibleux and Arnaud Fréville. Tabu search based procedure for solving the 0-1 multiobjective knapsack problem: The two objectives case. *J. Heuristics*, 6(3):361–383, 2000.

[6] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems.* Springer, 2004.

[7] Ehrgott Matthias and Gandibleux Xavier. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum*, 22(4):425–460, 2000.

[8] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993.

[9] Shin-ichi Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[10] Meir J. Rosenblatt and Zilla Sinuany-Stern. Generating the discrete efficient frontier to the capital budgeting problem. *Operations Research*, 37(3):384–394, 1989.

[11] Junn-Yuan Teng and Gwo-Hshiung Tzeng. A multiobjective programming approach for selecting non-independent transportation investment alternatives. *Transportation Research Part B: Methodological*, 30(4):291 – 307, 1996.

[12] Dalessandro Soares Vianna and José Elias Claudio Arroyo. A GRASP algorithm for the multi-objective knapsack problem. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile*, pages 69–75, 2004.