防衛大学校理工学研究報告
第 42 巻　第 1 号
平成 16 年 4 月

# ナップサック関数の不連続点を全列挙するアルゴリズム
## （松井　甲子雄教授，柏木　英一教授に捧ぐ）

藤本 晶子 *　　山田 武夫 **
(平成 16 年 3 月 16 日受付；平成 16 年 4 月 0 日受理)

# Algorithms to List up All the Discontinuity Points of a Knapsack Function
## (Dedicated to Prof. K. MATSUI and Prof. E. KASHIWAGI)

By Masako FUJIMOTO* and Takeo YAMADA**

**概要:** The knapsack function $z(c)$ is the optimal objective value of a knapsack problem as a function of the capacity $c$. It is known that $z(c)$ is a non-decreasing, right-continuous step function. In this paper we present two algorithms to list up all the discontinuity points of $z(c)$ included in a fixed interval $[c_0, c_1]$. We implement these algorithms in C language, and through numerical experiments compare these against the Nemhauser-Ullman algorithm. We find that one of our algorithms is superior to the existing method for the case with large $c_0$ and relatively small $c_1 - c_0$.
**Key words :** knapsack problem, knapsack function, combinational optimization.

## 1. はじめに

**ナップサック問題** [6, 10]

$$\textbf{KP}: \quad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \qquad (1.1)$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c \qquad (1.2)$$

$$x_j \in \{0, 1\}, \quad \forall j \qquad (1.3)$$

はオペレーションズ・リサーチや情報工学において基本的問題の 1 つで，この問題やさまざまな変型 [2, 4, 5, 13, 14] について数多くの研究がなされてきたが，本稿ではナップサック容量 $c$ が固定されておらず，区間 $[c_0, c_1]$ 内のあらゆる値をとり得る場合を考察する．すなわち，KP を $[c_0, c_1]$ 内の全ての $c$ について，パラメトリックに解くことを考える．

以下においては，問題のデータ $p_j, w_j$ $(j = 1, \cdots, n)$ はすべて正整数とする．KP の最適目的関数値を $z(c)$ と表し，これを $c$ の関数と考えたものを**ナップサック関数** [9] というが，これは単調非減少，右連続な階段関数となる．[2, 11] この関数上の点 $(c, z(c))$ が $z(c)$ の**不連続点**であるとは

$$\lim_{\varepsilon \downarrow 0} z(c - \varepsilon) \neq z(c) \qquad (1.4)$$

を意味する．

問題は $[c_0, c_1]$ 間で $z(c)$ のすべての不連続点を列挙することであるが，このような問題は一般化ナップサック共有問題 [2, 3] を解く際などにも重要な部分問題として現れる．

## 2. 従来の方法

$c$ の区間が $[0, c_1]$ の場合，すなわち $c_0 = 0$ のときは，**動的計画法** [1, 8, 12] (dynamic programming : DP) と，それを改良した Nemhauser, Ullman の方法 [11]（以下 **NU 法**と呼ぶ）が知られている．本節では，後に必要となる範囲でこれらについて概説する．

### 2.1 動的計画法

商品 1〜$k$ についてのナップサック問題

* 防衛大学校 第 41 期理工学研究科学生 情報数理専攻
** 防衛大学校 情報工学科 教授

**KP$_k$ :**

$$\text{maximize} \quad \sum_{j=1}^{k} p_j x_j \tag{2.1}$$

$$\text{subject to} \quad \sum_{j=1}^{k} w_j x_j \le c \tag{2.2}$$

$$x_j \in \{0,1\}, \quad \forall j \tag{2.3}$$

を考え，$z_k(c)$ をその最適目的関数値とすると，$k \ge 1$ の場合**最適性の原理** [1] より

$$z_k(c) = \begin{cases} \max\{z_{k-1}(c), z_{k-1}(c-w_k)+p_k\}, & c \ge w_k \text{の場合} \\ z_{k-1}(c), & \text{その他} \end{cases} \tag{2.4}$$

が成立する．$k=0$ の場合は

$$z_0(c) \equiv 0 \tag{2.5}$$

である．そこで，$k=1,2,\cdots$ の順に $[0,c_1]$ 内のすべての $c$ について $z_k(c)$ を逐次計算して行くと，$z_n(c)$ が求めるナップサック関数 $z(c)$ となる．

動的計画法では $k=1,\cdots,n$ について，すべての整数値 $c \in [0,c_1]$ で $z_k(c)$ の値を計算するので，その計算量は $O(nc_1)$ となり，$n$ や $c_1$（または，これら両方）が大きい場合には効率が悪い．

### 2.2　NU 法

この方法は，動的計画法と同様に (2.4) 式を基礎とするが，すべての $c \in [0,c_1]$ で $z_k(c)$ を計算するのではなく，不連続点のみで $z_k(c)$ を計算する．これを説明するために以下の記号を導入する．一般に，右連続な階段関数 $z(c)$ に対してその不連続点全体の集合を $DC(z)$ と記し，その個々の要素を $(z_w^k, z_p^k)$，$k=1,\cdots,n(z)$ と表す．すなわち，

$$DC(z) = \{(z_w^k, z_p^k) \mid k=1,\cdots,n(z)\} \tag{2.6}$$

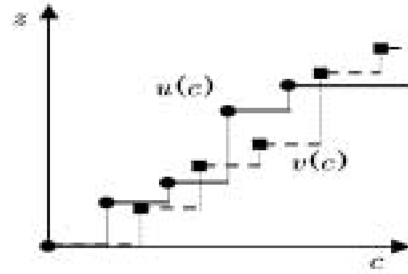
で，一般性を失うことなく，

$$z_w^0 = z_p^0 \equiv 0 \tag{2.7}$$

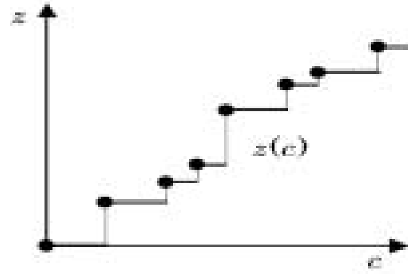
$$z_w^0 < z_w^1 < \cdots < z_w^{n(z)} \tag{2.8}$$

と仮定する．ここで，$n(z) := |DC(z)|$.

2 つの右連続な階段関数 $u(c), v(c)$ の不連続点の集合 $DC(u), DC(v)$ に対し，$z(c) := \max\{u(c), v(c)\}$ の不連続点集合 $DC(z)$ を求める操作を $u(c), v(c)$ の




(a) 階段関数 $u(c), v(c)$
(a) Step functions $u(c)$ and $v(c)$.




(b) 縦方向マージの結果
(b) Vertically merged function.

**図 2.1** 階段関数とそれらの縦方向マージ
**Fig. 2.1** Step functions and their vertical merge.

**縦方向マージ** という（図 2.1 参照）．このためのアルゴリズム v_merge は次のように与えられる．

**アルゴリズム v_merge**

入力：$DC(u), DC(v)$
出力：$DC(z)$
**Step 1.** $DC(z) \leftarrow \{(0,0)\}$, $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$ とする．
**Step 2.** $u_w^i = v_w^j = \infty$ なら $DC(z)$ を出力して終了．
**Step 3.** $u_w^i < v_w^j$ なら Step 4 へ，$u_w^i > v_w^j$ なら Step 5 へ進む．$u_w^i = v_w^j$ の場合は，$u_p^i \ge v_p^j$ なら Step 4 へ，$u_p^i < v_p^j$ なら Step 5 へ進む．
**Step 4.** (i) $z_p^k < u_p^i$ なら $DC(z) \leftarrow DC(z) \cup \{(u_w^i, u_p^i)\}$ , $k \leftarrow k+1$ とする．
(ii) $i \leftarrow i+1$ として Step 2 へ戻る．
**Step 5.** (i) $z_p^k < v_p^j$ なら $DC(z) \leftarrow DC(z) \cup \{(v_w^j, v_p^j)\}$ , $k \leftarrow k+1$ とする．
(ii) $j \leftarrow j+1$ として Step 2 へ戻る．

ところで，

$$z_{k-1}^S(c) := \begin{cases} z_{k-1}(c-w_k)+p_k, & c \ge w_k \text{の場合} \\ 0, & \text{その他} \end{cases} \quad (2.9)$$

とすると，$z_{k-1}^S(c)$ は $z_{k-1}(c)$ を $(w_k, p_k)$ だけ**シフト**（平行移動）したもので，これを用いると (2.4) 式は

$$z_k(c) = \max\{z_{k-1}(c), z_{k-1}^S(c)\} \quad (2.10)$$

と書ける．

NU 法は，$z_0(c) \equiv 0$ から始めて，(2.10) 式によって $z_k(c)$ を逐次 $k=n$ まで求めるもので，次のようにシフトと v_merge を交互に反復する．

**NU 法**

**Step 1.** $z_0(c) \equiv 0$ とする．
**Step 2.** $k = 1, 2, \cdots, n$ について以下を行う．
(i) $z_{k-1}(c)$ をシフトして $z_{k-1}^S(c)$ を得る．
(ii) $z_{k-1}(c)$ と $z_{k-1}^S(c)$ に v_merge を適用して $z_k(c)$ を得る．

### 2.3 既存の方法の問題点

本章であげた 2 つの方法のうち，NU 法は動的計画法に比べかなり大規模な問題を解くことができるが，これらはいずれも $c_0 = 0$ の場合を対象としており，$c_0$ がある程度大きくなると計算の無駄が多くなる．

これに対して，$c_0$ が大きく $[c_0, c_1]$ の幅はさほど大きくないときに，$[c_0, c_1]$ の範囲のみで KP をパラメトリックに解きたいことが生じる [2,3] が，これについての先行研究は見当たらない．

## 3. 下方探索法

ナップサック関数 $z(c)$ において，任意の整数 $c \ge 0$ に対し，横座標が $c$ を超えない範囲で最大の不連続点を $D(c)$ と記す．すなわち，$D(c) = (c^\dagger, z(c^\dagger))$ とすると，

$$c^\dagger = \min\{c' \mid z(c') = z(c)\} \quad (3.1)$$

$D(c)$ を求めるために次の**逆問題**を考える．

**IKP$(z)$ :**

$$\text{minimize} \quad \sum_{j=1}^{n} w_j x_j \quad (3.2)$$

$$\text{subject to} \quad \sum_{j=1}^{n} p_j x_j \ge z \quad (3.3)$$

$$x_j \in \{0,1\}, \quad \forall j \quad (3.4)$$

この問題の最適目的関数値を $c(z)$ とすると，$z(c)$ と $c(z)$ は互いに他の '逆関数' となっている．[2]

$c$ が与えられたとき，問題 KP$(c)$ を解くと，$z = z(c)$ が得られる．次にこの $z$ に対して，IKP$(z)$ を解くと不連続点 $D(c)$ が求められる．このことを利用すると，次のアルゴリズムにより $[c_0, c_1]$ 内の不連続点をすべて列挙することができる．この方法は，$c = c_1$ から逐次 $c$ の値を下げながら不連続点を探索していくので，**下方探索法**（downward search）と呼ぶ．

**下方探索法**

**Step 1.** $c := c_1$
**Step 2.** $c \le c_0$ なら終了．そうでなければ KP$(c)$ を解き，$z := z(c)$ を求める．
**Step 3.** IKP$(z)$ を解き，最適関数値 $c^\dagger := c(z)$ を得る．$D(c) = (c^\dagger, z)$ は不連続点なので，これを出力する．
**Step 4.** $c := c^\dagger - 1$ として Step 2 へ戻る．

図 3.1 は上のアルゴリズムの挙動を示したもので，図中の $\sharp i$ は Step $i$ を意味する．

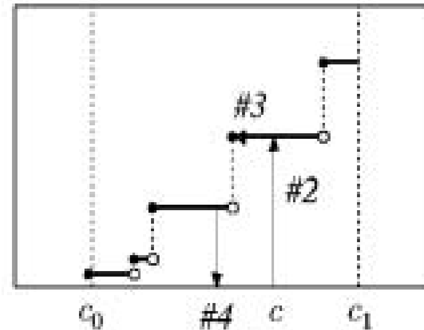



**図 3.1** 下方探索法の挙動
**Fig. 3.1** Downward search.

**例 3.1**

表 3.1 で $c_0 = 0, c_1 = 42$ としたときの下方探索法の動作を表 3.2 に示す．この例では全体で 22 回ナップサック問題を解き，図 3.2 のように 11 個の不連続点を得ている．

**表 3.1** 商品データ
**Table 3.1** Problem data.

| 商品 $j$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 利得 $p_j$ | 5 | 9 | 3 | 11 | 7 |
| 重量 $w_j$ | 2 | 12 | 13 | 8 | 6 |

**表 3.2** 下方探索法の挙動
**Table 3.2** Behavior of the downward search.

| ステップ | $c$ | $z$ | $c^\dagger$ |
|---|---|---|---|
| 0 | 42 | 35 | 41 |
| 1 | 40 | 32 | 28 |
| 2 | 27 | 27 | 26 |
| 3 | 25 | 25 | 22 |
| 4 | 21 | 23 | 16 |
| 5 | 15 | 18 | 14 |
| 6 | 13 | 16 | 10 |
| 7 | 9 | 12 | 8 |
| 8 | 7 | 7 | 6 |
| 9 | 5 | 5 | 2 |
| 10 | 1 | 0 | 0 |

ところで，上の方法では 1 つの不連続点を得るのに KP と IKP の 2 つのナップサック問題を解く必要がある．しかし，$n$ が大きい場合ナップサック問題を厳密に解くにはある程度時間がかかるので，この回数をさらに削減することが望ましい．このために次の事実を利用する．今，ナップサック問題 KP を解くために Horowitz-Sahni 法 7)（**HS 法**）を用いるものとし，その結果得られる点を $H(c) = (H_w(c), H_p(c))$ と記すと，$H(c)$ は 90% 以上の確率で不連続点 $D(c)$ と一致することが実験的に確かめられる．このことを利用すると，次のアルゴリズムが得られる．

**下方探索法（改良版）**

**Step 1.** $c := c_1$ とし，$H(c)$ を計算する．
**Step 2.** $c \le c_0$ なら終了．
**Step 3.** $c' \leftarrow H_w(c) - 1$ とし，$H(c')$ を計算する．
**Step 4.** もし $H_p(c') < H_p(c)$ ならば，$H(c)$ は不連続点であることが確認されたので，これを出力する．
**Step 5.** $c \leftarrow c', H(c) \leftarrow H(c')$ として Step 2 へ戻る．

この場合，$c' < c$ なので $H_p(c') \le H_p(c)$ であるが，さらに $H_p(c') < H_p(c)$ か，$H_p(c') = H_p(c)$ かによって次の 2 つの場合が生じる．

$H_p(c') < H_p(c)$ のときは，$H(c)$ は不連続点で，Step 4 で出力される．これに対して，$H_p(c') = H_p(c)$ の場合は $H(c)$ は不連続点でない．この場合は $c$ に対応する不連続点 $D(c)$ を求めるのに，2 回以上 HS 法を適用することになる．実際には，前者が後者よりもはるかに多く生起するので，1 つの不連続点について KP を 1 回解けば良いことが多い．

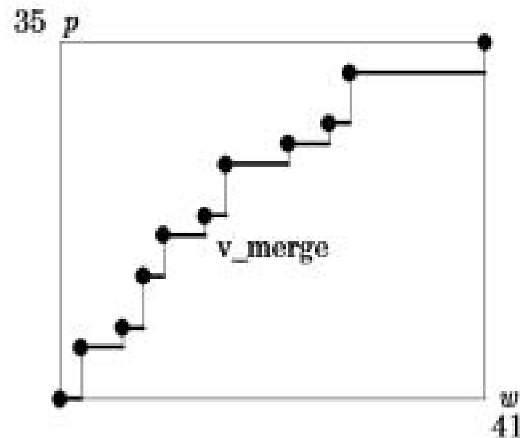



**図 3.2** 下方探索法の実行例
**Fig. 3.2** Result of the downward search.

**例 3.2**

下方探索法（改良版）を例 3.1 と同じ問題に適用すると，その動作は表 3.3 のようになる．

**表 3.3** 改良アルゴリズムの動作
**Table 3.3** Behavior of the improved algorithm.

| ステップ | $c$ | $H(c)$ ( $w$, $p$ ) | $c'$ | $H(c')$ ( $w$, $p$ ) |
|---|---|---|---|---|
| 0 | 42 | (41, 35)* | 40 | (28, 32) |
| 1 | 40 | (28, 32)* | 27 | (26, 27) |
| 2 | 27 | (26, 27)* | 25 | (23, 25) |
| 3 | 25 | (23, 25) | 22 | (22, 25) |
| 4 | 22 | (22, 25)* | 21 | (16, 23) |
| 5 | 21 | (16, 23)* | 15 | (14, 18) |
| 6 | 15 | (14, 18)* | 13 | (10, 16) |
| 7 | 13 | (10, 16)* | 9 | ( 8, 12) |
| 8 | 9 | ( 8, 12)* | 7 | ( 6, 7) |
| 9 | 7 | ( 6, 7)* | 5 | ( 2, 5) |
| 10 | 5 | ( 2, 5)* | 1 | ( 0, 0) |

上の表で，下線は HS 法によりナップサック問題を解いたことを意味し，⋆ は不連続点として出力された点を表す．全体で，ナップサック問題を 11 回解いて前と同様の 11 個の不連続点を得ている．例 3.1 に比べ，KP を解く回数が約半分ですんでいることがわかる．

## 4.　複合法

本節では，NU 法と下方探索法を組み合わせた**複合法**（hybrid method）と呼ぶ方法を考える．今，$[c_0, c_1]$ で商品 1〜$k-1$ についてのナップサック関数 $z_{k-1}(c)$ が図 4.1 のとおりであったとする．

$z_{k-1}(c)$ を $(w_k, p_k)$ だけシフトした関数 $z^S_{k-1}(c)$ は図 4.2 の破線（■ ··· ■部分）のようになるが，区間 $[c_0, c_0+w_k]$ 間でこの関数は未知である．そこで，この区間を対象に商品 1〜$k-1$ に関するナップサック関数を下方探索法で求めると，$[c_0, c_1]$ 全体で関数 $z^S_{k-1}(c)$ が復元される（復元された部分を▲ ··· ▲で示す）．複合法は次のようにまとめられる．

**複合法**

**Step 1.** $z_0(c) \equiv 0$ とする．
**Step 2.** $k = 1, 2, \cdots, n$ について以下を行う．
(i) 関数 $z_{k-1}(c)$ を $(w_k, p_k)$ だけシフトする．
(ii) $[c_0, c_0+w_k]$ で商品 1〜$k-1$ について下方探索法を適用して $z^S_{k-1}(c)$ を復元する．
(iii) $z_{k-1}(c)$ と $z^S_{k-1}(c)$ を縦方向マージして $z_k(c)$ を得る．

**例 4.1**

例 3.1 と同じ問題で $c_0 = 9, c_1 = 42$ として複合法を適用すると，ナップサック問題を 10 回解いて図 4.3 のように 7 個の不連続点を得た．

## 5. 数値実験

前章のアルゴリズムを ANSI C 言語で実装し，IBM RS/6000 SP44 Model 270 (CPU：POWER3-II SMP 2 way, 375MHz) 上で，$n, c_0, c_1$ を様々に変えて一連の数値実験を行った．また，$w_j$ と $p_j$ は以下の相関タイプ[10] によりランダムに発生させた．

- 無相関 (UNCOR)

  $w_j$ ：$[1, 10000]$ の一様乱数
  $p_j$ ：$[1, 10000]$ の一様乱数で，$w_j$ と独立

- 弱相関 (WEAK)

  $w_j$ ：$[1, 10000]$ の一様乱数
  $p_j$ ：$[w_j, w_j + 200]$ の一様乱数

### 5.1 無相関の場合

5.1〜5.3 節では UNCOR の場合の結果をまとめる．表 5.1 は，$c_0 = 2000n$, $c_1 = c_0 + 10000$ の場合（以下，これを**実験 0** と呼ぶ）で，各行は 10 回のランダムな試行の平均値であり，♯DC, ♯DC(Total), ♯KP はそれぞれ，$[c_0, c_1]$ と $[0, c_1]$ 内の $z(c)$ の不連続点数，および KP を解いた回数を示す．また CPU は計算に要した CPU 時間を秒で示している．

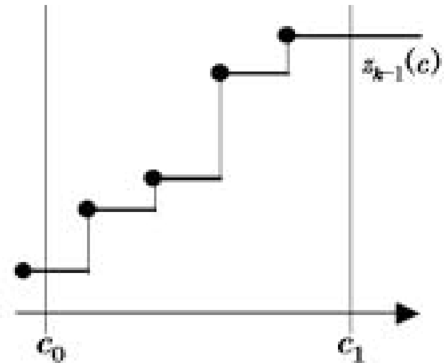



**図 4.1** 関数 $z_{k-1}(c)$
**Fig. 4.1** Function $z_{k-1}(c)$ .

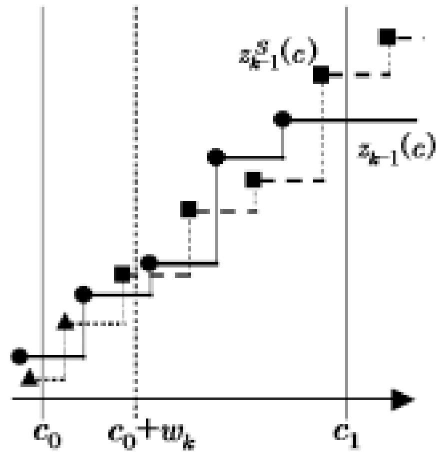



**図 4.2** 複合法
**Fig. 4.2** Hybrid method.

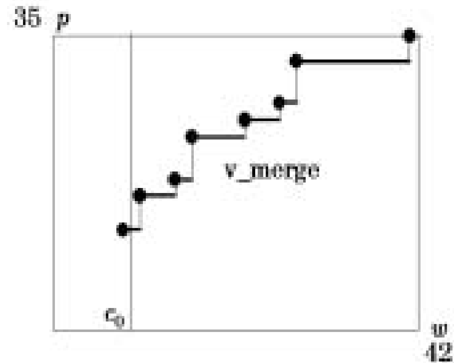



**図 4.3** 複合法の実行例
**Fig. 4.3** Result of the hybrid method.

**表 5.1** 無相関の場合
**Table 5.1** The uncorrelated case (UNCOR).

| $n$ | ♯DC | NU 法 | | 下方探索法 | | 複合法 | |
|---|---|---|---|---|---|---|---|
| | | ♯DC(Total) | CPU | ♯KP | CPU | ♯KP | CPU |
| 800 | 283.6 | 55152.6 | 2.60 | 292.0 | 1.28 | 1843.7 | 6.01 |
| 1000 | 332.9 | 85292.4 | 5.08 | 346.6 | 2.27 | 2364.9 | 12.20 |
| 1200 | 402.5 | 116971.6 | 8.71 | 412.9 | 3.62 | 2977.8 | 20.40 |
| 1400 | 401.8 | 154480.1 | 14.06 | 413.5 | 4.63 | 3313.5 | 30.48 |
| 1600 | 489.6 | 198854.7 | 22.08 | 500.8 | 7.24 | 3982.7 | 46.53 |
| 1800 | 552.7 | 246995.1 | 31.98 | 565.8 | 10.29 | 4569.3 | 68.62 |
| 2000 | 586.6 | 298255.0 | 44.19 | 600.6 | 12.93 | 5135.2 | 89.73 |
| 4000 | 1000.1 | 1048121.6 | 338.98 | 1058.9 | 83.40 | 11086.5 | 794.04 |
| 6000 | 1446.6 | 2174917.4 | 1049.52 | 1547.6 | 286.31 | 18154.0 | 3282.50 |
| 8000 | 1949.4 | 3596284.4 | 2330.95 | 2109.2 | 724.23 | 24871.1 | 7388.12 |
| 10000 | 2208.5 | 5264102.4 | 4368.92 | 2466.6 | 1320.84 | 31843.7 | 12248.63 |

図 5.1 , 5.2 は, 表 5.1 の計算時間および不連続点数を問題サイズ $n$ の関数として示したものである．これらから，本章の例では

1. 計算速度は下方探索法，NU 法，複合法の順である（図 5.1 ），

2. 不連続点数は $n$ にほぼ比例する（図 5.2 ），

ことがわかる．複合法は不連続点を全列挙するアルゴリズムとしては他の方法に劣るが，$n$ 個の商品についてのナップサック関数が求められているときに，商品 $n+1, n+2, \cdots$ が 1 つずつ追加されるような場合に，ナップサック関数を逐次更新していく，‘オンライン・アルゴリズム’ としては有用と考えられる．

表 5.1 の下方探索法について，不連続点数と KP を解いた回数の比率を図 5.3 に示す．これから，KP を解く回数は不連続点数の 1.1 倍程度であることがわかる．

### 5.2　区間幅の影響

表 5.2 に，区間幅を 50000 とした場合の結果をまとめる．すなわち，$c_0 = 2000n, c_1 = c_0 + 50000$（**実験 1**）で，各行は表 5.1 と同様，10 回の試行の平均値である．— は 20000 秒以内の CPU 時間で計算が終了しなかったケースを含むものを示している．

図 5.4 は表 5.1（実験 0）と表 5.2（実験 1）の比較，すなわち，区間幅が 10000 と 50000 の場合の計算時間の比較である．これから，

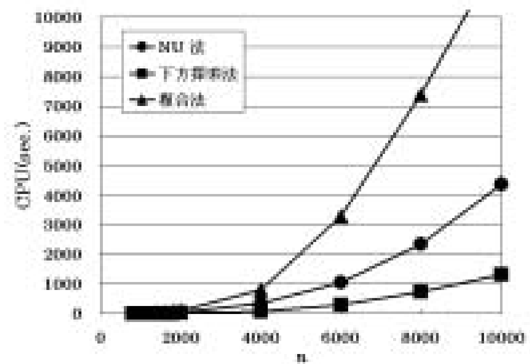



**図 5.1** 計算時間 (UNCOR)
**Fig. 5.1** CPU time in seconds (UNCOR).

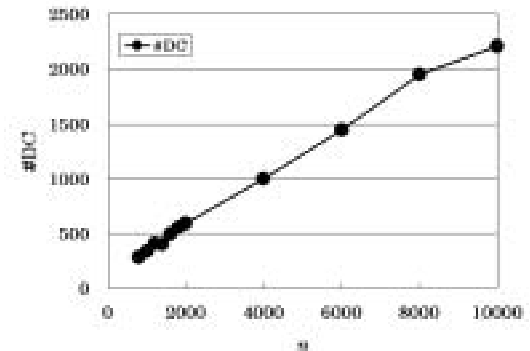



**図 5.2** 不連続点数 (UNCOR)
**Fig. 5.2** Number of discontinuity points (UNCOR).

**表 5.2** 区間幅 50000 の場合 (UNCOR)
**Table 5.2** The UNCOR case with $c_1 - c_0 = 50000$.

| $n$ | ♯DC | NU 法 | | 下方探索法 | | 複合法 | |
|---|---|---|---|---|---|---|---|
| | | ♯DC(Total) | CPU | ♯KP | CPU | ♯KP | CPU |
| 800 | 1437.9 | 56306.9 | 2.65 | 1466.0 | 6.49 | 2463.3 | 8.25 |
| 1000 | 1734.3 | 86693.8 | 5.14 | 1775.6 | 11.76 | 3177.3 | 16.67 |
| 1200 | 1950.0 | 118519.1 | 8.75 | 1993.7 | 18.06 | 4048.5 | 28.26 |
| 1400 | 2056.7 | 156135.0 | 14.07 | 2099.2 | 23.66 | 4523.1 | 42.94 |
| 1600 | 2314.3 | 200679.4 | 22.26 | 2369.1 | 34.75 | 5476.8 | 65.72 |
| 1800 | 2617.1 | 249059.5 | 32.05 | 2680.2 | 48.10 | 6353.6 | 97.23 |
| 2000 | 2918.7 | 300587.1 | 44.36 | 2996.3 | 65.81 | 7163.3 | 130.07 |
| 4000 | 4945.1 | 1052066.6 | 340.20 | 5190.7 | 400.08 | 16002.5 | 1144.30 |
| 6000 | 7314.1 | 2180784.9 | 1050.55 | 7841.8 | 1477.06 | 18154.0 | 6172.29 |
| 8000 | 9812.9 | 3604147.9 | 2353.84 | 10681.2 | 3597.13 | 24871.1 | 8599.87 |
| 10000 | 11072.3 | 5272966.2 | 4626.43 | 12345.6 | 6644.28 | — | — |

1. 下方探索法では，計算時間は区間幅に比例している, すなわち，区間幅が 5 倍となると計算時間もほぼ 5 倍となっている,
2. NU 法では，区間幅の影響はほとんど受けない,
3. 区間幅が大きい場合は，下方探索法より NU 法が速い,

ことがわかる．

### 5.3 区間の位置の影響

表 5.3 は，$c_0 = 4000n$, $c_1 = c_0 + 10000$ とした場合（**実験 2**）の結果である．表中で — としているのはメモリー不足で計算できなかったケースを含むものを示している．

図 5.5 ～ 5.6 は表 5.1（実験 0）と表 5.3（実験 2），すなわち，区間の位置が $c_0 = 2000n$ と $c_0 = 4000n$ の場合の不連続点数と計算時間の比較である．これらから，以下が観察される．

1. 区間の位置によって，不連続点数はかなり差がある．
2. 下方探索法では，計算時間は区間の位置の影響をあまり受けない,
3. NU 法も計算時間は区間の影響をあまり受けないが，探索範囲 $[0, c_1]$ が広くなるとメモリー不足で計算ができなくなるケースが生じる．

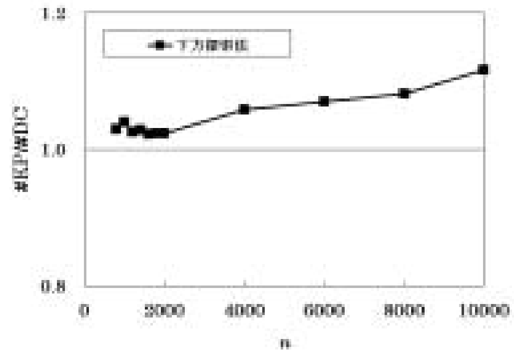



**図 5.3** 不連続点数と KP を解く回数の比較
**Fig. 5.3** Number of discontinuity points v.s. number of KPs solved.

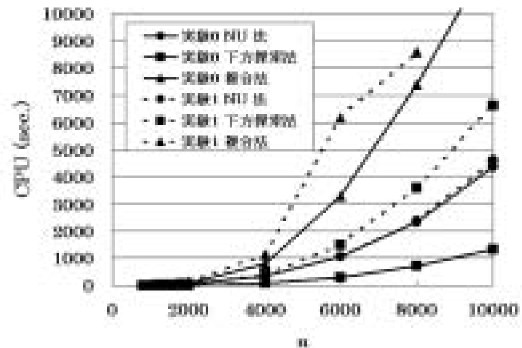



**図 5.4** 探索幅の影響
**Fig. 5.4** Comparison of the cases with $c_1 - c_0 = 10000/50000$.

**表 5.3** 区間の位置 $c_0 = 4000n$ の場合 (UNCOR)
**Table 5.3** The UNCOR case with $c_0 = 4000n$.

| $n$ | ♯DC | NU 法 | | 下方探索法 | | 複合法 | |
|---|---|---|---|---|---|---|---|
| | | ♯DC(Total) | CPU | ♯KP | CPU | ♯KP | CPU |
| 800 | 82.0 | 84194.5 | 3.60 | 85.4 | 0.48 | 520.3 | 2.83 |
| 1000 | 102.3 | 128970.6 | 6.90 | 104.7 | 0.92 | 728.8 | 6.28 |
| 1200 | 115.9 | 177514.3 | 11.93 | 119.8 | 1.50 | 907.4 | 10.94 |
| 1400 | 118.0 | 235776.9 | 19.46 | 122.0 | 2.08 | 990.8 | 16.28 |
| 1600 | 142.1 | 303026.5 | 30.05 | 145.5 | 3.14 | 1158.7 | 24.14 |
| 1800 | 155.6 | 378624.8 | 43.78 | 165.1 | 4.48 | 1414.1 | 36.44 |
| 2000 | 192.6 | 456764.2 | 59.85 | 200.0 | 6.55 | 1634.4 | 52.11 |
| 4000 | 338.0 | 1609665.7 | 440.73 | 363.3 | 55.96 | 3758.3 | 603.29 |
| 6000 | 466.2 | 3340094.9 | 1476.66 | 507.2 | 203.44 | 4396.9 | 3034.04 |
| 8000 | 553.7 | — | — | 622.6 | 452.23 | 5795.9 | 5040.60 |
| 10000 | 681.4 | — | — | 783.3 | 898.90 | 7676.2 | 10447.10 |

### 5.4　相関の影響

表 5.4 は弱相関の場合の結果（**実験 3**）である．複合法はどのケースも 20000 秒以上の CPU 時間を要したので，比較の対象から除外した．表中で — としているのはメモリー不足で計算できなかったケースを含むものを示している．

図 5.7 ～ 5.8 は表 5.1 （実験 0）と表 5.4 （実験 3），すなわち，無相関と弱相関の場合の不連続点数と計算時間の比較である．これらから，

1. 相関がある場合，不連続点数はかなり大きくなり，その結果計算時間も大きくなる，

ことが認められる．

この理由は以下のように説明される．図 5.9 ～ 5.10 は $n = 8$ の場合の UNCOR, WEAK のナップサック関数のグラフである．この 2 つを比べると，UNCOR の場合，ナップサック関数の上側包絡線は緩やかに上に湾曲した凹関数となり，各不連続点の間隔が大きいのに対し，WEAK の場合は対応する関数が直線的で，各不連続点の間隔は小さい．したがって，相関があると不連続点数は大きくなり，その結果計算もそれに比例して困難になると考えられる．

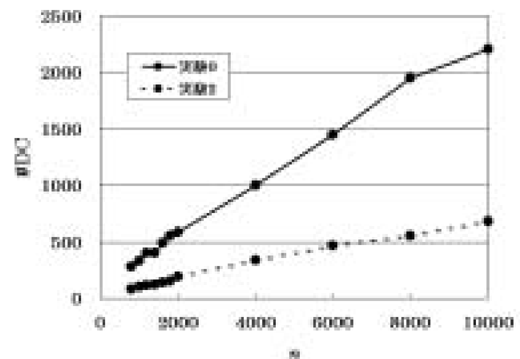



**図 5.5** 探索区間の位置と不連続点数
**Fig. 5.5** Interval positions vs. the number of DCs.

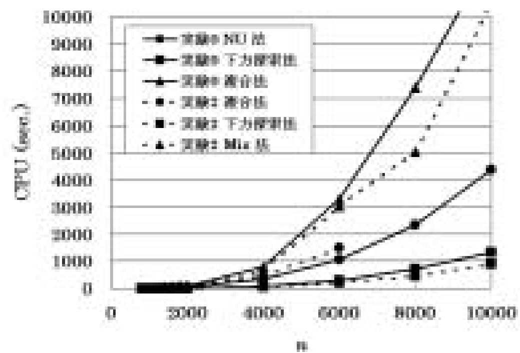



**図 5.6** 探索区間の位置の影響
**Fig. 5.6** Interval positions vs. CPU time.

### 6.　まとめ

ナップサック関数 $z(c)$ の不連続点で，区間 $[c_0, c_1]$ に含まれるものを全列挙するアルゴリズムとして下方探索法と複合法を提案し，C 言語で実装した．既存の NM 法と併せてそれらの効率を数値実験により評価したところ，$c_0$ が大で区間幅 $c_1 - c_0$ があまり大きくない場合には下方探索法が他の方法より効率的であるこ

**表 5.4** 弱相関の場合 (WEAK)
**Table 5.4** The weakly correlated case (WEAK).

| $n$ | ♯DC | NU 法 | | 下方探索法 | |
|---|---|---|---|---|---|
| | | ♯DC(Total) | CPU | ♯KP | CPU |
| 800 | 5524.1 | 763794.7 | 52.95 | 6564.2 | 296.89 |
| 1000 | 6054.9 | 1065322.3 | 93.83 | 7216.3 | 394.88 |
| 1200 | 6495.2 | 1383971.6 | 147.27 | 7738.9 | 489.21 |
| 1400 | 6840.4 | 1718903.6 | 214.86 | 8130.2 | 603.81 |
| 1600 | 7000.9 | 2074621.4 | 297.09 | 8342.3 | 682.88 |
| 1800 | 7213.5 | 2431322.5 | 366.90 | 8573.2 | 812.39 |
| 2000 | 7523.9 | 2799056.9 | 475.08 | 8865.3 | 931.73 |
| 4000 | 8869.3 | — | — | 9699.1 | 1625.81 |
| 6000 | 9245.4 | — | — | 9866.8 | 2779.90 |
| 8000 | 9527.7 | — | — | 9909.7 | 4275.69 |
| 10000 | 9649.6 | — | — | 9942.4 | 6103.97 |

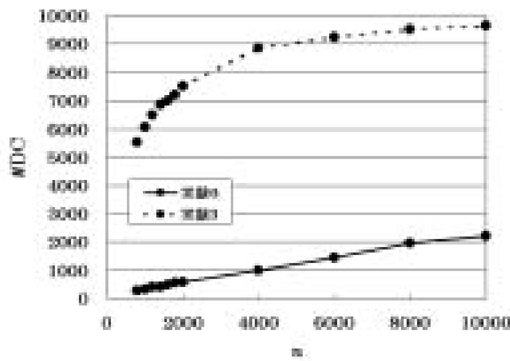



**図 5.7** 相関と不連続点数
**Fig. 5.7** Correlation vs. #DCs.

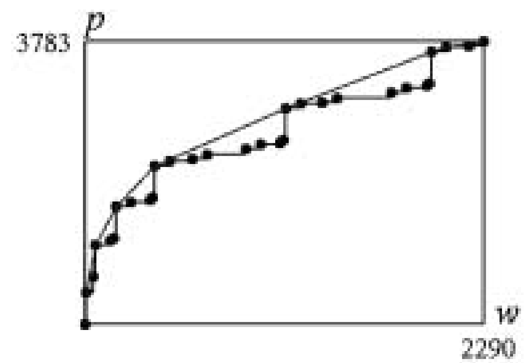



**図 5.9** ナップサック関数：無相関の場合
**Fig. 5.9** Knapsack function (UNCOR).

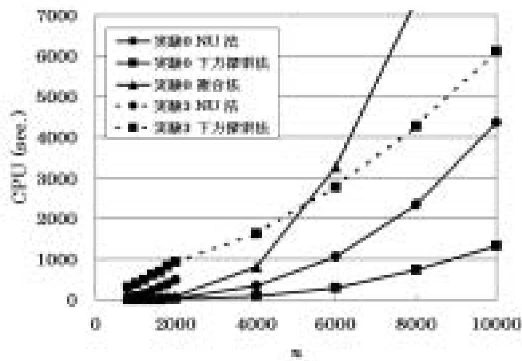



**図 5.8** 相関の影響
**Fig. 5.8** Correlation vs. CPU time.

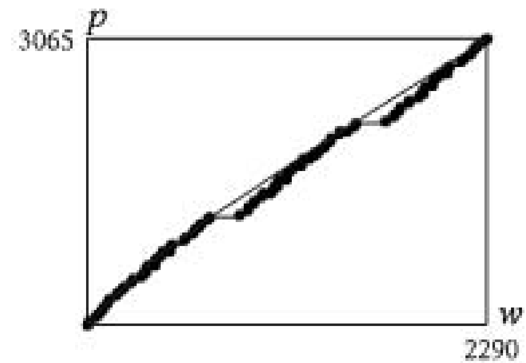



**図 5.10** ナップサック関数：弱相関の場合
**Fig. 5.10** Knapsack function (WEAK).

とがわかった．複合法は，商品が一つずつ追加させるような場合に，ナップサック関数を逐次更新するオンライン・アルゴリズムとして使用するのが適当と考えられる

**参考文献**


1) Bellman, R., *Dynamic Programming*, Princeton Univ. Press, 1957.
2) 藤本晶子, 一般化ナップサック共有問題の解法に関する研究, 修士論文, 防衛大学校, 2004.
3) 藤本晶子, 山田武夫, 片岡靖詞, 渡辺宏太郎, 一般化ナップサック共有問題の厳密解法, 防衛大学校理工学研究報告, 第 41 巻第 1 号, pp. 123-135, 2003.
4) Fujimoto, M. and Yamada, T., “An exct algorithm for the knapsack sharing problem with common items”, mimeo., Dept. of Computer Science, The National Defense Academy, 2003, submied to *European Journal of Operational Research.*
5) 二川真由美, ナップサック共有問題に関する研究, 修士論文, 防衛大学校, 1996.
6) 林芳男, 0-1 ナップザック問題の数理とアルゴリズム, 近畿大学商経学会, 2000.
7) Horowitz, E. and Sahni, S., “Computing partitions with applications to the knapsack problem”, *Journal of ACM*, **21**, 277-292 (1974).
8) 茨木俊秀, アルゴリズムとデータ構造, 昭晃堂, 1989.
9) 今野浩, 鈴木久敏, 数理計画法と組合せ最適化, 日科技連, 1982.
10) Martello, S. and Toth, P., *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, 1990.
11) Nemhauser, G. L. and Ullmann, Z., “Discreate dynamic programming and capital allocation”, *Management Science*, **15**, 494-505 (1969).
12) Sedgewick, R., *Algorithms*, Addison-Wesley Publishing Company, 1988. 野下浩平他 訳, 近代科学社, 1990.
13) Yamada, T. and Futakawa, M., “Heuristic and reduction algorithms for the knapsack sharing problem”, *Computers & Operations Research*, **24**, 961-967 (1996).
14) Yamada, T., Futakawa, M. and Kataoka, S., “Some exact algorithms for the knapsack sharing problem”, *European Journal of Operational Research*, **106**, 177-183 (1998).

# Algorithms to List up All the Discontinuity Points of a Knapsack Function

(Dedicated to Prof. K. MATSUI and Prof. E. KASHIWAGI)

By Masako FUJIMOTO* and Takeo YAMADA**

(Received March 16, 2004; accepted for publication April 0, 2004)

**Abstract**

The knapsack function $z(c)$ is the optimal objective value of a knapsack problem as a function of the capacity $c$. It is known that $z(c)$ is a non-decreasing, right-continuous step function. In this paper we present two algorithms to list up all the discontinuity points of $z(c)$ included in a fixed interval $[c_0, c_1]$. We implement these algorithms in C language, and through numerical experiments compare these against the Nemhauser-Ullman algorithm. We find that one of our algorithms is superior to the existing method for the case with large $c_0$ and relatively small $c_1 - c_0$.
**Key words :** knapsack problem, knapsack function, combinational optimization.

* Graduate student, Department of Computer Science, The National Defense Academy
** Professor, Department of Computer Science, The National Defense Academy