数理最適化　レポート課題2

0-1整数計画ソルバーの作成と計算機実験

提出締切　2020年8月7日

情報学群情報科学類　3年
学籍番号：201811319
氏名：永崎遼太

目次

1. 実験の概要

1.1 目標

　汎用的な最適化数理モデルである整数計画問題に対する最適化アルゴリズムについて理解する。

1.2 実験の内容

　以下の3つのことを行う。
・線形計画問題およびその解法（単体法）について理解し、実装する。（cf. テキスト [3] 第2章）
・ナップサック問題に対する分枝限定法を理解し、実装する。（cf. テキスト [3] 第3章）
・0-1整数計画問題ソルバーを作成する。本レポートではこの実験内容についてまとめる。

1.3 テキスト

　以下のテキストを参考として実験を行う。

[3] 「Pythonによる数理最適化入門」 久保幹雄（監修）／並木誠（著） （朝倉書店） 2018年 」

2. 実験内容

2.1 0-1整数計画問題

　本実験では、ピンパッキング問題についての0-1整数計画問題についてソルバーを作成し、それに関する計算機実験を行った。

2.2 実験方法

　0-1整数計画問題ソルバーを作成する。問題例の解が出力されるかどうかの確認、さらに問題サイズを変えたときの計算時間比較も行う。

3. ピンパッキング問題

3.1 ピンパッキング問題の説明

　ピンパッキング問題とは離散数学の組み合わせ論の中のNP困難問題で、与えられた「荷物（重さや個数がついている）」を詰める「箱（便やコンテナなど）」の最小数を見つける問題である。

具体的には、次のような問題である。

問題：

それぞれ重さが w_j であるような品物が n=10個ある。これを容量 C=25 の容器に入れて運びたい。必要な容器の最小数を求めよ。

10個の品物の重量

品物	1	2	3	4	5	6	7	8	9	10
重量 w_j (kg)	8	9	5	6	8	5	5	6	9	9

3.2 ビンパッキング問題の定式化

　ビンパッキング問題を、0-1整数線形最適化問題として解く方法と、列生成法を利用して解く方法の1つの方法についての定式化を考える。

それぞれについて説明をする。

3.2.1 0-1 整数線形最適化問題

　ビンパッキング問題の 0-1 整数線形最適化問題の定式化について説明する。定式化をする上で以下のようなポイントがある。

1. 容器は品物の数だけあれば十分である（最悪でも1つの容器に1つの品物を入れるという方法がある）ので、容器の集合も品物の集合と同じNとする。
2. 品物iがj番目の容器に入っている時1、そうでない時0を表す0-1変数$x_{i,j}\ (i, j \in N)$を用意する。
3. j番目の容器に何か品物が入っているときには1、入っていないときには0を表す0-1変数$z_j\ (j \in N)$を用意する。

これらの変数を使うと、ビンパッキング問題は以下のように定式化される。

$$\text{最小化} \quad \sum_{j \in N} z_j$$

$$\text{条件} \quad \sum_{j \in N} x_{i,j} = 1,\ \forall i \in N$$

$$\sum_{i \in N} w_i x_{i,j} \leq C z_j,\ \forall j \in N$$

$$x_{i,j} \in \{0,1\},\ z_j \in \{0,1\},\ \forall i, j \in N$$

条件の1番目の式は、どの品物戻れ下の容器に必ず入れることを表している。2番目の式は、もしj番目の容器が使われているとしたら、そこに入っている品物の総重量はC以下であることを表している。

3.2.2 列生成法

列生成法(column generation method)とは、入力行列が横に長い場合の線形最適化問題を解くための手法である。

まず、次のような不等式標準型の線形最適化問題を考える。

$$\text{最大化} \quad C_K^T x_K$$

$$\text{条件} \quad A x \leq b,\ x \geq 0$$

これをPとする。また、この双対問題を考える。

$$\text{最小化} \quad b^T y$$

$$\text{条件} \quad A^T y \geq c,\ y \geq 0$$

これをDとする。列（変数）のインデックスを$N = \{1,2,...,n\}$ とし、KをNの部分集合とする。この部分集合に関してPの部分問題$P(K)$を次のように定義する。

最大化　$C_K^T x_K$

条件　$A_i^T y \geq c_i,\ x_K \geq 0$

A_Kは$m * K$でAの部分行列、c_K, x_K はc, xのKで添字つけされた部分ベクトルである。問題$P(K)$の双対問題$D(K)$は次のようにして定義できる。

最小化　$b^T y$

条件　$A_i^T y \geq c_i,\ \forall i \in K,\ y \geq 0$

この双対問題$D(K)$の変数yは、元々の問題Pの双対問題と同じ大きさである。さらに$D(K)$には最適化y^{*K}が存在すると仮定すると、次のような性質が成立する。

1. 任意の$i \notin K$に対し、$c_j - A_j^T y^{*K} \leq 0 \Leftrightarrow y^{*K}$は問題Dの最適解である
2. $c_j - A_j^T y^{*K} > 0$となる$j \notin K$が存在する $\Leftrightarrow y^{*K}$は問題Dの最適解ではない

加えて、線形最適化問題に関して次のような性質が成立しているとしている。

1. 係数行列$A \in \mathbb{R}^{m*n}$の列数は行数に対して非常に大きいとする。
2. $D(K)$の最適解y^{*K}が効率よく求めら得るようなKを簡単に求められる。
3. y^{*K}が求められていれば$c_j - A_j^T y^{*K} > 0$となる$A_j(j \notin K)$は、効率よく計算でき、またこのようなjが存在しないことも効率よく確認できる。

このような状況のもとで、次のような線形最適化問題に対するアルゴリズムが、列生成法である。

1. $D(K)$の最適解y^{*K}が効率よく求められるようなKを初期値としてスタートする。
2. $D(K)$の最適解y^{*K}を求める
3. y^{*K}を元に、$c_j - A_j^T y^{*K} > 0$となる$A_j(j \notin K)$を求める。なければy^{*K}がPの最適解であるということであるので、終了する

4. $K = K + \{j\}$として2へ

この列生成法をビンパッキング問題へ適応する。

$N = \{1, 2, \cdots, 10\}$ とする。空ではなく1つの容器に収容可能な品物集合 J を集めたものを M とする。

$$M = \{J \subseteq N \mid \sum_{j \in J} w_i \leq C\}$$

M のそれぞれの要素 J に対して、0-1変数 x_J を用意する。$x_J = 1$ の時集合 J に入っている品物を1つの容器 J に入れる。$x_J = 0$ の時は集合 J の組み合わせは使わないと考える。さらに、ペア $(i, J)(i \in N, J \in M)$ について、$a_{i,J}$ を次のように決定する。

$$a_{i,J} = \begin{cases} 1 & if\ i \in J \\ 0 & if\ i \notin J \end{cases}$$

これらを使うと、ピンパッキング問題は以下のように定式化される。

$$\begin{aligned} &\text{minimize} \ \textstyle\sum_{J \in M} x_J \\ &\text{subject to} \\ &\quad \textstyle\sum_{J \in M} a_{i,J} x_J \geq 1, \ \forall i \in N \\ &\quad x_J \in \{0, 1\}, \ \forall J \in M \end{aligned}$$

この問題をBPとする。BPは、制約式の個数は n と少ないが、変数の個数が n の増加に対して M の要素の個数が指数関数的に増加していくので、少し大きな n に対しても実時間で解くのは困難であると予想される。そこで、列生成を利用する。ピンパッキング問題は以下のように線形緩和した問題を考える。

$$
\begin{aligned}
&\text{minimize} \ \sum_{J \in M} x_J \\
&\text{subject to} \\
&\quad \sum_{J \in M} a_{i,J} x_J \geq 1, \ \forall i \in N \\
&\quad x_J \geq 0, \ \forall J \in M
\end{aligned}
$$

この問題をLBPとする。

また、この問題の双対問題を定式化すると、次のようになる。

$$
\begin{aligned}
&\text{maximize} \ \sum_{i \in N} y_i \\
&\text{subject to} \\
&\quad \sum_{i \in N} a_{i,J} y_i \leq 1, \ \forall J \in M \\
&\quad y_i \geq 0, \ \forall i \in N
\end{aligned}
$$

この問題をD-LBPとする。

問題BPと問題LBPについて、以下のことが成り立つ。

LBPの最適値(=D-LBPの最適値) ≤ BPの最適値

このことから、LBPに列生成方を適用して得られるのは、BPの最適値ではなく、BPの下界である。さらに、この下界がBPの最適解になるようなより良い近似解を生成する方法を用いる。

M の部分集合 $O \in M$ について O に属する J に関する変数のみを扱う部分問題 LBP(O)は次のようにして表される。

$$
\begin{aligned}
&\text{minimize} \ \sum_{i \in O} x_J \\
&\textit{subject to} \\
&\quad \sum_{J \in O} a_{i,J} x_J \geq 1, \ \forall i \in N \\
&\quad x_J \geq 0, \ \forall J \in O
\end{aligned}
$$

さらに、この問題の双対問題D-LBP(O)は次のようになる。

$$
\begin{aligned}
&\text{maximize} \ \textstyle\sum_{i \in N} y_i \\
&subject\ to \\
&\quad \textstyle\sum_{i \in N} a_{i,J} y_i \leq 1, \ \forall J \in O \\
&\quad y_i \geq 0, \ \forall i \in N
\end{aligned}
$$

双対問題D-LBP(O)が最適解 y^{*O} を持つとすると、この値が問題D-LBPの最適解であるための必要十分条件は以下のとおりである。

$$
y^{*O} \text{が問題D-LBPの最適解} \quad \Leftrightarrow \quad 1 \geq \textstyle\sum_{i \in N} a_{i,J} y_i^{*O}, \ \forall J \in M
$$

ここで、 $a_{i,J}$ は $\sum_{i \in N} a_{i,J} w_j \leq C$ をみたし、$i \in J$ の時1で、$i \notin J$ の時0である数である。

問題D-LBPの最適解であるための必要十分条件は、重さを w_i 、価格を y^{*O} としたときの次のナップザック問題 $KP(y^{*O})$ を解くことによって確かめることができる。

$$
\begin{aligned}
&\text{maximize} \ \textstyle\sum_{i \in N} p_i y_i^{*O} \\
&subject\ to \\
&\quad \textstyle\sum_{i \in N} p_i w_i \leq C \\
&\quad p_i \in \{0,1\} \ for \ i \in N
\end{aligned}
$$

$KP(y^{*O})$ の最適値が1以下ならば問題D-LBPの最適解であるための必要十分条件が成立し、$KP(y^{*O})$ の最適値が1より大きければ $p_i = a_{i,J}$ となるような J を O に加えてD-LBP(O)を解きなおせば良い。これをD-LBPの最適解がもとまるまで繰り返すと、結果としてLBPの最適解、つまりピンパッキング問題の下界がえらえっる。最終的に、この下界が算出されたときの O を使った整数最適化問題 BP(O) をソルバーで解く。得られた最適値と下界とのさが1未満ならば、その解は元の問題BPの最適解である。1より大きい場合は、その解は元の問題の近似解である。

4. ソルバーの実装

ビンパッキング問題を0-1 整数線形最適化問題として解く方法と、列生成法を用いた方法の2つの方法についてのソルバーを実装する。

4.1 0-1 整数線形最適化問題

Jupyter Notebook にて、binpacking_solver.ipynb ファイルを作成した。

実際に作成したコードは次のとおりである。

```
from pulp import *
MEPS=1.0e-8
```

```
def binpacking(capacity,w):
    n=len(w)
    items=range(n)
    bpprob=LpProblem(name='BinPacking2', sense=LpMinimize)
    z=[LpVariable('z'+str(j),lowBound=0,cat='Binary')for j in items]
    x=[[LpVariable('x'+str(i)+'_'+str(j),lowBound=0,cat='Binary') for j in items] for i in items]

    bpprob += lpSum(z[i] for i in items)
    for i in items:
        bpprob += lpSum(x[i][j] for j in items) ==1
    for j in items:
        bpprob += lpSum(x[i][j]*w[i] for i in items) <= capacity*z[j]

    bpprob.solve()
    result=[]
    for j in items:
        if z[j].varValue > MEPS:
            r=[w[i] for i in items if x[i][j].varValue > MEPS]
            result.append(r)
    print(result)
```

```
capacity = 25
items = set(range(17))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

binpackingメソッド内で、3.2.1節の定式化に基づいて、変数、目的関数、制約式を定義している。

実際の問題を3番目のセルで定義している。

4.2 列生成法

Jupyter Notebook にて、binpacking_solver.ipynb ファイルを作成した。列生成法では、分枝限定法を用いたナップザック問題を解く方法を利用ため、ナップザック問題を解くためのクラス、メソッドを定義する必要がある。

4.2.1 ナップザック問題クラスの定義

分枝限定法を実現するために、ナップザック問題クラスを次のように定義する。

```
class KnapsackProblem(object):
    """The difinition of KnapSackProblem """
    def __init__(self,name,capacity,items,costs,weights,zeros=set(),ones=set()):
        self.name=name
        self.capacity=capacity
        self.items=items
        self.costs=costs
        self.weights=weights
        self.zeros=zeros
        self.ones=ones
        self.lb=-100
        self.ub=-100
        ratio={j:costs[j]/weights[j] for j in items}
        self.sitemList=[k for k,
                        v in sorted(ratio.items(),key=lambda x:x[1], reverse=True)]
        self.xlb={j:0 for j in self.items}
        self.xub={j:0 for j in self.items}
        self.bi=None

    def getbounds(self):
        """ Calculate the upper and lower bounds. """
        for j in self.zeros:
            self.xlb[j]=self.xub[j]=0
        for j in self.ones:
            self.xlb[j]=self.xub[j]=1
        if self.capacity < sum(self.weights[j] for j in self.ones):
            self.lb=self.ub=-100
            return 0

        ritems = self.items-self.zeros-self.ones
        sitems=[j for j in self.sitemList if j in ritems]
        cap=self.capacity-sum(self.weights[j] for j in self.ones)

        for j in sitems:
            if self.weights[j] <= cap:
                cap-=self.weights[j]
                self.xlb[j]=self.xub[j]=1
            else:
                self.xub[j]=cap/self.weights[j]
                self.bi=j
                break
            self.lb=sum(self.costs[j]*self.xlb[j] for j in self.items)
            self.ub=sum(self.costs[j]*self.xub[j] for j in self.items)

    def __str__(self):
        """ KnapSachProblemの情報を印字 """
        return ('Nama = '+self.name+',capacity='+str(self.capacity)+',\n'
                'items = '+str(self.items)+',\n'+
                'costs = '+str(self.costs)+',\n'+
                'weights = '+str(self.weights)+',\n'+
                'zeros = '+str(self.zeros)+', ones = '+str(self.ones)+',\n'+
                'lb = '+str(self.lb)+', ub = '+str(self.ub)+',\n'+
                'sitemList = '+str(self.sitemList)+',\n'+
                'xlb = '+str(self.xlb)+',\n'+'xub = '+str(self.xub)+',\n'+
                'bi = '+str(self.bi)+'\n')
```

コンストラクタと上界、下界を計算するメソッドgetbound、問題の情報を出力するための__str__メソッドの定義からなる。

4.2.2 ナップザック問題に対する分枝限定法

　ナップザック問題クラスを使った分枝限定法を実装したKnapsackProblemSolveメソッドを定義する。

```
def KnapsackProblemSolve(capacity,items,costs,weights):
    from collections import deque
    queue=deque()
    root=KnapsackProblem('KP',capacity=capacity,
                         items=items,costs=costs,weights=weights,
                         zeros=set(),ones=set())
    root.getbounds()
    best=root
    queue.append(root)

    while queue !=deque([]):
        p=queue.popleft()
        p.getbounds()
        if p.ub> best.lb: #bestを更新する可能性がある
            if p.lb>best.lb: #bestを更新する
                best=p
            if p.ub>p.lb: #子問題を作って分枝する
                k=p.bi
                p1=KnapsackProblem(p.name+'+'+str(k),
                                   capacity=p.capacity,items=p.items,
                                   costs=p.costs,weights=p.weights,
                                   zeros=p.zeros,ones=p.ones.union({k}))
                queue.append(p1)
                p2=KnapsackProblem(p.name+'-'+str(k),
                                   capacity=p.capacity,items=p.items,
                                   costs=p.costs,weights=p.weights,
                                   zeros=p.zeros.union({k}),ones=p.ones)
                queue.append(p2)
    return 'Optimal', best.lb, best.xlb
```

問題を管理するためのキューとしてcollections.dequeを使っている。

4.2.3 ビンパッキング問題への列生成法の適用

ビンパッキング問題の緩和問題を列生成法で解き、近似解を得るまでを以下のようにして実装する。

```
def binpacking(capacity,w):
    m=len(w)
    items=set(range(m))
    A=np.identity(m)

    solved=False
    columns=0
    dual=LpProblem(name='D(K)', sense=LpMaximize)
    y=[LpVariable('y'+str(i),lowBound=0) for i in items]

    dual += lpSum(y[i] for i in items) # 目的関数の設定
    for j in range(len(A.T)): # 制約条件の付加
        dual += lpDot(A.T[j],y) <=1, 'ineq'+str(j)

    while not(solved):
        #dual
        dual.solve()

        costs={i: y[i].varValue for i in items}
        weights={i: w[i] for i in items}
        (state, val, sol) = KnapsackProblemSolve(capacity, items, costs,weights)

        if val >= 1.0+MEPS:
            a=np.array([int(sol[i]) for i in items])
            dual += lpDot(a,y) <= 1 ,'ineq'+str(m+columns)
            A=np.hstack((A,a.reshape((-1,1))))
            columns+=1
        else:
            solved=True

    print('Generated columns: ', columns)

    m,n =A.shape
    primal =LpProblem(name='P(K)',sense=LpMinimize)
    x =[LpVariable('x'+str(j),lowBound=0,cat='Binary') for j in range(n)]

    primal += lpSum(x[j] for j in range(n)) # 目的関数の設定
    for i in range(m): # 制約条件の付加
        primal += lpDot(A[i],x) >=1,'ineq'+str(i)

    primal.solve()
    if value(primal.objective)-value(dual.objective) < 1.0-MEPS:
        print('Optimal solution found: ')
    else:
        print('Approximated solution found: ')
    K =[j for j in range(n) if x[j].varValue > MEPS]
    result =[]
    itms =set(range(m))
    for j in K:
        J = {i for i in range(m) if A[i,j] > MEPS and i in items}
        r = [w[i] for i in J]
        items -= J
        result.append(r)
    print(result)
```

そして、4.1節と同様にして、実際の問題を以下で定義している。

```
capacity = 25
items = set(range(10))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

5. 実行結果

　ビンパッキング問題を0-1 整数線形最適化問題として解く4.1節で実装したソルバーと、列生成法を用いた4.2節で実装したソルバーの2つのソルバーについて、問題例の解が出力されるかどうかの確認、 さらに問題サイズを変えたときの計算時間についてまとめる。

5.1 0-1 整数線形最適化問題

　問題サイズを変え、次の例題1～7を解いた。

例題1

　　容量：　25

　　荷物：　重さが5～9のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(10))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9]
[[5, 6, 5, 6], [5, 9, 9], [8, 9, 8]]
CPU times: user 5.27 ms, sys: 5.7 ms, total: 11 ms
Wall time: 28.6 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9]

詰め方：
[[5, 6, 5, 6], [5, 9, 9], [8, 9, 8]]

計算時間：
CPU times: user 5.27 ms, sys: 5.7 ms, total: 11 ms
Wall time: 28.6 ms

例題2
荷物の個数が同じで、容量、荷物の重さを大きくする。

容量：　50
荷物：　重さが10～29のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 50
items = set(range(10))
np.random.seed(1)
w={i:np.random.randint(10,30) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[15, 21, 22, 18, 19, 21, 15, 25, 10, 26]
[[15, 18, 15], [22, 26], [21, 25], [19, 21, 10]]
CPU times: user 5.13 ms, sys: 4.96 ms, total: 10.1 ms
Wall time: 36 ms
```

荷物：
[15, 21, 22, 18, 19, 21, 15, 25, 10, 26]

詰め方：
[[15, 18, 15], [22, 26], [21, 25], [19, 21, 10]]

計算時間：
CPU times: user 5.13 ms, sys: 4.96 ms, total: 10.1 ms
Wall time: 36 ms

例題3
荷物の個数が同じで、容量、荷物の重さを大きくする。

容量：　100
荷物：　重さが30～59のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 100
items = set(range(10))
np.random.seed(2)
w={i:np.random.randint(30,60) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[38, 45, 43, 38, 52, 41, 48, 41, 38, 37]
[[38, 37], [43, 41], [45, 41], [38, 48], [52, 38]]
CPU times: user 5.31 ms, sys: 5.33 ms, total: 10.6 ms
Wall time: 28.7 ms
```

荷物：
[38, 45, 43, 38, 52, 41, 48, 41, 38, 37]

詰め方：
[[38, 37], [43, 41], [45, 41], [38, 48], [52, 38]]

計算時間：
CPU times: user 5.21 ms, sys: 5.22 ms, total: 10.4 ms
Wall time: 27.4 ms

例題4

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである15個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた

出力：

```
capacity = 25
items = set(range(15))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9]
[[5, 5, 6, 9], [9, 6, 7], [8, 5, 9], [6, 9], [8, 9, 7]]
CPU times: user 9.24 ms, sys: 5.72 ms, total: 15 ms
Wall time: 42 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9]

詰め方：
[[5, 5, 6, 9], [9, 6, 7], [8, 5, 9], [6, 9], [8, 9, 7]]

計算時間：
CPU times: user 9.24 ms, sys: 5.72 ms, total: 15 ms
Wall time: 42 ms

例題5

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである30個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(30))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9]
[[5, 9, 6, 5], [6, 6, 6, 6], [9, 7, 9], [6, 7, 6, 6], [5, 6, 7, 5], [5, 9, 9], [8, 8, 9], [9, 7, 8], [9, 9]]
CPU times: user 28.9 ms, sys: 6.25 ms, total: 35.2 ms
Wall time: 71.4 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9]

詰め方：
[[5, 9, 6, 5], [6, 6, 6, 6], [9, 7, 9], [6, 7, 6, 6], [5, 6, 7, 5], [5, 9, 9], [8, 8, 9], [9, 7, 8], [9, 9]]

計算時間：
CPU times: user 28.9 ms, sys: 6.25 ms, total: 35.2 ms
Wall time: 71.4 ms

例題6

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである50個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(50))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6,
5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8]
[[5, 8, 5, 7], [6, 9, 9], [8, 6, 5, 6], [5, 9, 5, 6], [5, 7, 7, 6], [9, 6, 5, 5], [9, 8, 8], [9, 9, 6], [6, 6
, 5, 8], [9, 9, 7], [8, 9, 8], [6, 5, 6, 8], [9, 7, 9], [6, 7, 6, 6]]
CPU times: user 67.5 ms, sys: 7.3 ms, total: 74.8 ms
Wall time: 13.1 s
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6, 5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8]

詰め方：
[[5, 8, 5, 7], [6, 9, 9], [8, 6, 5, 6], [5, 9, 5, 6], [5, 7, 7, 6], [9, 6, 5, 5], [9, 8, 8], [9, 9, 6], [6, 6, 5, 8], [9, 9, 7], [8, 9, 8], [6, 5, 6, 8], [9, 7, 9], [6, 7, 6, 6]]

計算時間：
CPU times: user 67.5 ms, sys: 7.3 ms, total: 74.8 ms
Wall time: 13.1 s

例題7

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである60個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(60))
np.random.seed(7)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking(capacity,w)
```

```
[9, 6, 8, 8, 9, 6, 5, 6, 7, 7, 5, 9, 5, 9, 5, 8, 7, 8, 9, 9, 5, 5, 5, 8, 5, 7, 5, 6, 9, 6, 8, 8, 5, 8, 5, 6,
5, 5, 9, 9, 9, 7, 6, 8, 5, 9, 5, 9, 8, 6, 8, 8, 8, 6, 8, 8, 9, 8, 6, 8]
[[7, 5, 5, 8], [9, 7, 9], [8, 9, 8], [6, 5, 5, 9], [5, 6, 8, 6], [8, 6, 5, 6], [6, 6, 5, 8], [5, 6, 9, 5], [8
, 9, 8], [9, 7, 9], [8, 9, 8], [6, 5, 9, 5], [9, 8, 8], [5, 7, 5, 8], [9, 8, 8], [5, 7, 5, 8], [9, 6, 8]]
CPU times: user 92.9 ms, sys: 8.56 ms, total: 102 ms
Wall time: 23.1 s
```

荷物：
[9, 6, 8, 8, 9, 6, 5, 6, 7, 7, 5, 9, 5, 9, 5, 8, 7, 8, 9, 9, 5, 5, 5, 8, 5, 7, 5, 6, 9, 6, 8, 8, 5, 8, 5, 6, 5, 5, 9, 9, 9, 7, 6, 8, 5, 9, 5, 9, 8, 6, 8, 8, 8, 6, 8, 8, 9, 8, 6, 8]

詰め方：
[[7, 5, 5, 8], [9, 7, 9], [8, 9, 8], [6, 5, 5, 9], [5, 6, 8, 6], [8, 6, 5, 6], [6, 6, 5, 8], [5, 6, 9, 5], [8, 9, 8], [9, 7, 9], [8, 9, 8], [6, 5, 9, 5], [9, 8, 8], [5, 7, 5, 8], [9, 8, 8], [5, 7, 5, 8], [9, 6, 8]]

計算時間：
CPU times: user 92.9 ms, sys: 8.56 ms, total: 102 ms
Wall time: 23.1 s

5.2 列生成法

問題サイズを変え、次の例題1～8を解いた。

例題1

容量：　25

荷物：　重さが5～9のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(10))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9]
Generated columns:  22
Optimal solution found:
[[8, 9, 5], [9, 6, 8], [9, 5, 5, 6]]
CPU times: user 43.1 ms, sys: 69.5 ms, total: 113 ms
Wall time: 199 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9]

詰め方：
[[8, 9, 5], [9, 6, 8], [9, 5, 5, 6]]

計算時間：
CPU times: user 43.1 ms, sys: 69.5 ms, total: 113 ms
Wall time: 199 ms

例題2
荷物の個数が同じで、容量、荷物の重さを大きくする。

容量：　50
荷物：　重さが10～29のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 50
items = set(range(10))
np.random.seed(1)
w={i:np.random.randint(10,30) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[15, 21, 22, 18, 19, 21, 15, 25, 10, 26]
Generated columns:  18
Optimal solution found:
[[22, 21], [10, 15, 25], [19], [15, 18], [21, 26]]
CPU times: user 37.2 ms, sys: 67.5 ms, total: 105 ms
Wall time: 169 ms
```

荷物：
[15, 21, 22, 18, 19, 21, 15, 25, 10, 26]

詰め方：
[[22, 21], [10, 15, 25], [19], [15, 18], [21, 26]]

計算時間：
CPU times: user 37.2 ms, sys: 67.5 ms, total: 105 ms
Wall time: 169 ms

例題3
荷物の個数が同じで、容量、荷物の重さを大きくする。

容量：　100
荷物：　重さが30～59のいずれかである10個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 100
items = set(range(10))
np.random.seed(2)
w={i:np.random.randint(30,60) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[38, 45, 43, 38, 52, 41, 48, 41, 38, 37]
Generated columns:  12
Approximated solution found:
[[37, 38], [41, 41], [38], [45, 48], [43], [38, 52]]
CPU times: user 26.6 ms, sys: 46.5 ms, total: 73.1 ms
Wall time: 124 ms
```

荷物：
[38, 45, 43, 38, 52, 41, 48, 41, 38, 37]

詰め方：
[[37, 38], [41, 41], [38], [45, 48], [43], [38, 52]]

計算時間：
CPU times: user 26.6 ms, sys: 46.5 ms, total: 73.1 ms
Wall time: 124 ms

例題：4

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである15個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(15))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9]
Generated columns:  40
Optimal solution found:
[[6, 7, 5, 6], [6, 9, 9], [5, 7, 8, 5], [8, 9], [9, 9]]
CPU times: user 97.4 ms, sys: 137 ms, total: 234 ms
Wall time: 373 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9]

詰め方：
[[6, 7, 5, 6], [6, 9, 9], [5, 7, 8, 5], [8, 9], [9, 9]]

計算時間：
CPU times: user 97.4 ms, sys: 137 ms, total: 234 ms
Wall time: 373 ms

例題5

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである30個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(30))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9]
Generated columns:  68
Approximated solution found:
[[6, 5, 6, 6], [5, 6], [9, 9, 6], [9, 7, 9], [7, 9, 9], [9, 9, 7], [6, 7, 6, 5], [8, 9], [5, 8], [6, 6, 8, 5]
]
CPU times: user 247 ms, sys: 227 ms, total: 475 ms
Wall time: 892 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9]

詰め方：
[[6, 5, 6, 6], [5, 6], [9, 9, 6], [9, 7, 9], [7, 9, 9], [9, 9, 7], [6, 7, 6, 5], [8, 9], [5, 8], [6, 6, 8, 5]]

計算時間：
CPU times: user 247 ms, sys: 227 ms, total: 475 ms
Wall time: 892 ms

例題6

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである50個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(50))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6,
5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8]
Generated columns:  69
Approximated solution found:
[[6, 6, 6, 6], [6, 6, 6, 6], [6, 5, 7], [5, 7, 7, 5], [8, 8, 8], [8, 8], [8, 8], [9, 9], [9, 9], [9], [9, 6,
9], [5, 9, 5], [5, 5, 7], [9, 6, 5], [6, 5], [9, 5, 8], [9, 7, 6], [9, 6, 7]]
CPU times: user 364 ms, sys: 235 ms, total: 599 ms
Wall time: 944 ms
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6, 5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8]

詰め方：
[[6, 6, 6, 6], [6, 6, 6, 6], [6, 5, 7], [5, 7, 7, 5], [8, 8, 8], [8, 8], [8, 8], [9, 9], [9, 9], [9], [9, 6, 9], [5, 9, 5], [5, 5, 7], [9, 6, 5], [6, 5], [9, 5, 8], [9, 7, 6], [9, 6, 7]]

計算時間：
CPU times: user 364 ms, sys: 235 ms, total: 599 ms
Wall time: 944 ms

例題7

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである60個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(60))
np.random.seed(2)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[5, 5, 8, 7, 8, 5, 7, 6, 8, 7, 9, 9, 9, 8, 9, 7, 8, 8, 7, 6, 7, 9, 8, 5, 9, 8, 6, 7, 5, 9, 9, 7, 9, 7, 6, 5,
7, 7, 6, 5, 6, 5, 7, 6, 6, 6, 9, 7, 8, 5, 8, 5, 7, 7, 5, 9, 7, 5, 7, 9]
Generated columns:  96
Approximated solution found:
[[5, 5, 5, 5, 5], [5, 5, 5, 5], [6, 6, 6, 6], [6, 6, 6, 6], [7, 7, 7], [7, 7], [7, 7], [7, 7, 7], [7, 7], [7,
7], [8, 8, 8], [8, 8], [8, 8], [8, 8, 8], [5, 9], [9], [5, 9, 6], [7, 9, 9], [7, 9], [9, 9, 7], [9, 9], [9, 9
, 5]]
CPU times: user 596 ms, sys: 336 ms, total: 932 ms
Wall time: 1.93 s
```

荷物：
[5, 5, 8, 7, 8, 5, 7, 6, 8, 7, 9, 9, 9, 8, 9, 7, 8, 8, 7, 6, 7, 9, 8, 5, 9, 8, 6, 7, 5, 9, 9, 7, 9, 7, 6, 5, 7, 7, 6, 5, 6, 5, 7, 6, 6, 6, 9, 7, 8, 5, 8, 5, 7, 7, 5, 9, 7, 5, 7, 9]

詰め方：
[[5, 5, 5, 5, 5], [5, 5, 5, 5], [6, 6, 6, 6], [6, 6, 6, 6], [7, 7, 7], [7, 7], [7, 7], [7, 7, 7], [7, 7], [7, 7], [8, 8, 8], [8, 8], [8, 8], [8, 8, 8], [5, 9], [9], [5, 9, 6], [7, 9, 9], [7, 9], [9, 9, 7], [9, 9], [9, 9, 5]]

計算時間：
CPU times: user 596 ms, sys: 336 ms, total: 932 ms
Wall time: 1.93 s

例題8

容量、荷物の重さの範囲をそのままに荷物の個数を多くする。

容量：　25
荷物：　重さが5〜9のいずれかである100個

このビンパッキング問題を解くと、出力は次のようになり、最適解が求められた。

出力：

```
capacity = 25
items = set(range(100))
np.random.seed(1)
w={i:np.random.randint(5,10) for i in items}
w2= [w[i] for i in items]
print(w2)

%time binpacking2(capacity,w)
```

```
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6,
5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8, 6, 7, 5, 9, 6, 7, 7, 6, 5, 6, 8, 9, 8, 6, 8, 5, 5, 7, 7, 6, 8, 9, 7
, 5, 5, 6, 6, 8, 5, 5, 9, 7, 9, 8, 8, 5, 8, 9, 8, 9, 9, 9, 6, 5, 9, 7, 5, 7, 9, 6]
Generated columns:  158
Approximated solution found:
[[5, 5, 5, 5, 5], [5, 5, 5, 5], [6, 6, 6, 6], [6, 6, 6, 6], [6, 6, 6], [7, 7, 7], [7, 6], [7, 7], [8, 7, 8],
[8, 8, 8], [6, 8, 8], [8, 8], [8, 8, 8], [8, 8], [9, 9, 5], [9, 9], [9, 9, 6], [7, 6, 6], [9, 5, 5], [7, 8, 7
], [7, 9, 9], [8, 9], [7, 7, 6, 5], [5, 9, 5, 6], [6, 9, 5, 5], [6, 8, 5], [9, 9], [6, 6, 7], [9, 6, 9], [9,
7, 9], [6, 9], [5, 5, 9], [9, 9], [5, 9]]
CPU times: user 1.51 s, sys: 586 ms, total: 2.1 s
Wall time: 5.22 s
```

荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9, 6, 7, 9, 7, 9, 8, 9, 7, 9, 7, 9, 6, 6, 5, 6, 6, 6, 6, 5, 9, 6, 5, 5, 8, 7, 6, 5, 8, 6, 6, 8, 9, 5, 6, 8, 9, 7, 9, 5, 8, 6, 7, 5, 9, 6, 7, 7, 6, 5, 6, 8, 9, 8, 6, 8, 5, 5, 7, 7, 6, 8, 9, 7, 5, 5, 6, 6, 8, 5, 5, 9, 7, 9, 8, 8, 5, 8, 9, 8, 9, 9, 9, 6, 5, 9, 7, 5, 7, 9, 6]

詰め方：
[[5, 5, 5, 5, 5], [5, 5, 5, 5], [6, 6, 6, 6], [6, 6, 6, 6], [6, 6, 6], [7, 7, 7], [7, 6], [7, 7], [8, 7, 8], [8, 8, 8], [6, 8, 8], [8, 8], [8, 8, 8], [8, 8], [9, 9, 5], [9, 9], [9, 9, 6], [7, 6, 6], [9, 5, 5], [7, 8, 7], [7, 9, 9], [8, 9], [7, 7, 6, 5], [5, 9, 5, 6], [6, 9, 5, 5], [6, 8, 5], [9, 9], [6, 6, 7], [9, 6, 9], [9, 7, 9], [6, 9], [5, 5, 9], [9, 9], [5, 9]]

計算時間：
CPU times: user 1.51 s, sys: 586 ms, total: 2.1 s
Wall time: 5.22 s

6. 考察

　5の実行結果において、5.1節の0-1整数線形最適化問題として解くソルバー、5.2節の列生成法を用いたソルバーでの例題1～7では、ランダムに荷物の重さを生成する際のシード値を同じに設定することで、問題設定を同じにしている。例えば、例題1では次のようなビンパッキング問題を解いた。

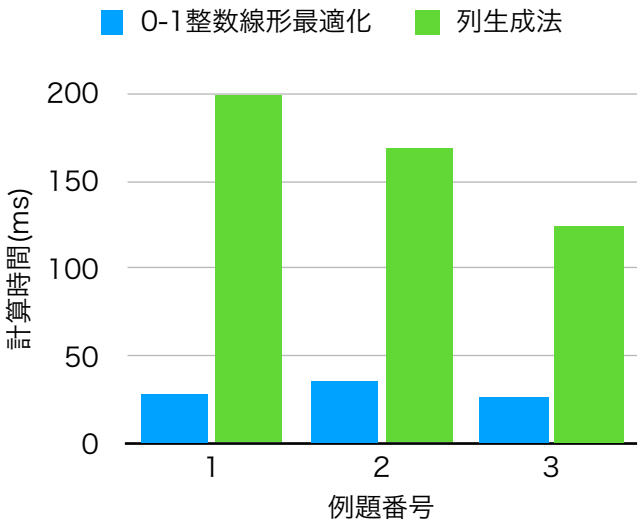
荷物：
[8, 9, 5, 6, 8, 5, 5, 6, 9, 9]

それぞれのソルバーでは次のような、結果になった。

詰め方（0-1整数線形最適化問題）：
[[5, 6, 5, 6], [5, 9, 9], [8, 9, 8]]

詰め方（列生成法）：
[[8, 9, 5], [9, 6, 8], [9, 5, 5, 6]]

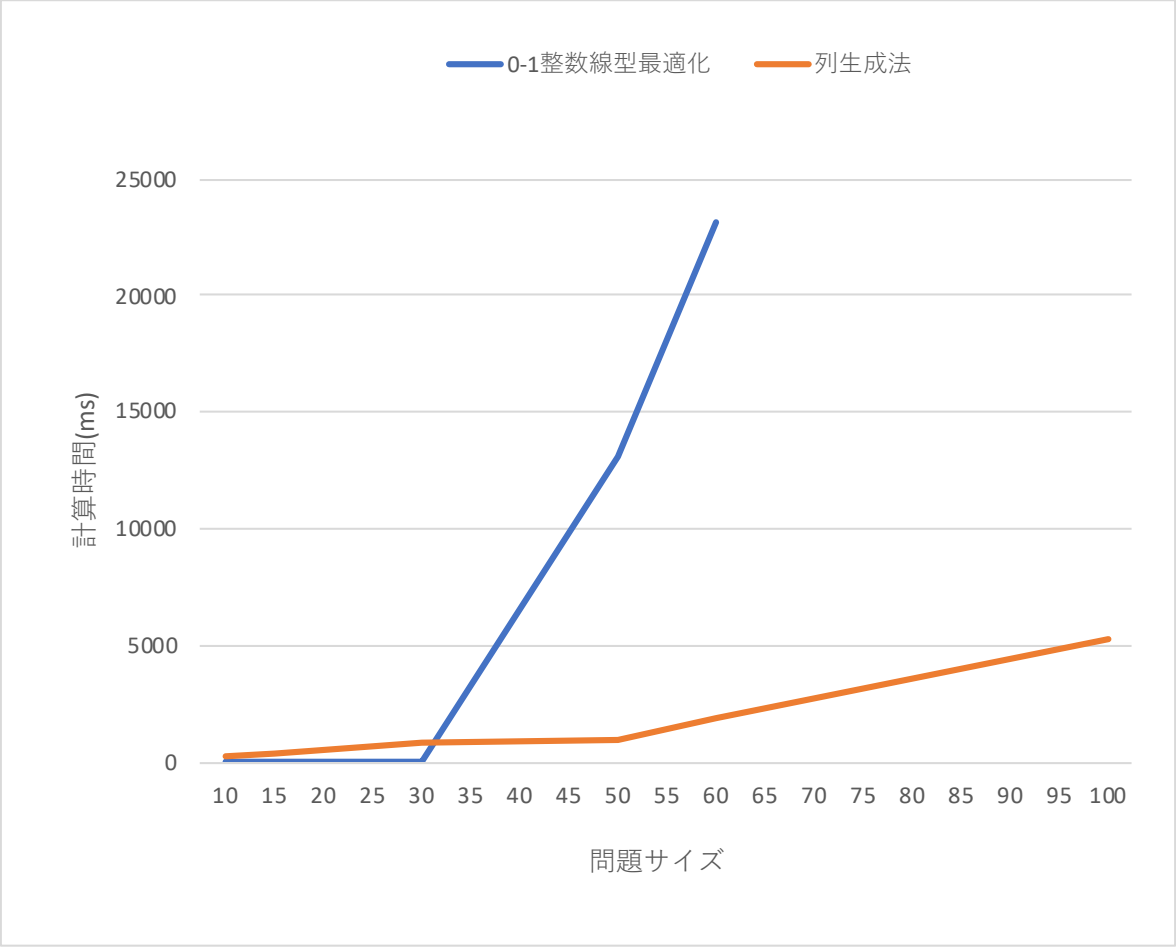
組み合わせは異なるが、最適値は同じ3であり、3つの容器が必要だとわかる。他の例題についても、組み合わせは異なるが、最適値は同じであることが確認できた。そのため、正しい最適値を求めることができたと言える。



次に、実行時間についての考察をする。5.1節及び5.2節の例題1〜3では、荷物の個数が同じで、容量、荷物の重さを大きくした。その際の計算時間をグラフにすると、次のようになる。

このグラフからも分かるように、荷物の個数が同じである場合、容量、荷物の重さは計算時間に影響しないということがわかる。また、荷物の個数が少ない場合には、0-1整数線形最適化のソルバーの方が、列生成法のソルバーよりも計算時間が短いということがわかる。

続いて、荷物の個数が10、15、30、50、60である場合の計算時間をグラフにすると、次のようになる。



このグラフからもわかるように、問題サイズが大きくなるほど計算時間の差が大きく開いていくことがわかる。ビンパッキング問題の 0-1 整数線形最適化は、大きいnに関しては実時間に解けないという問題が生じるので、小さい問題に対する、3.2.2節で説明する列生成法を用いた最適化問題の確かめ算として使うべきであると考えらえる。