製品生産の大規模スケジューリング問題　IP まとめ

弓林　司

2020 年 7 月 16 日

概要

製品生産の大規模スケジューリング問題[*1] の研究を行った。ここでは整数計画法（IP [*2] ）についてまとめた[*3] 。実装は Python 及びその整数計画法モデラー PuLP, ソルバー CBC, Gurobi を用いた。

目次

[*1] 以下スケジューリング問題と呼ぶ。
[*2] Integer Problem
[*3] 赤字は未実装を表す。

第 I 部
設定

1　目的

1.1　目的

製品生産のスケジューリング問題を整数計画法 [1][2][3] を用いて解きたい。

1.2　整数計画法[*4]

整数計画法とは整数変数 $\boldsymbol{x}$ [*5] からなる “目的函数”

$$\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$$

を不等式を含む “制約条件”

$$A\boldsymbol{x} \leq \boldsymbol{b}$$

の下最適化する（$\boldsymbol{x}$ を探索する[*6] ）問題である[*7] 。スケジューリング問題以外にも応用は広い [4][5] 。

$$\begin{aligned}
&\text{最大化} \quad Z = 4x_1 + 5x_2 \\
&\text{条　件} \quad 2x_1 + 2x_2 \leqq 7 \\
&\qquad\quad 3x_1 + 5x_2 \leqq 14 \\
&\qquad\quad x_1, x_2 \geqq 0 \\
&\qquad\quad x_1, x_2 : \text{整数}
\end{aligned}$$

図 1　整数計画問題定式化

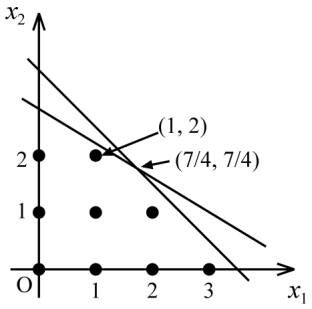


図 2　イメージ図

[*4] ここでは簡単の為線型に限定し議論する。
[*5] 連続変数だけであれば線型計画法（LP）、連続変数と整数変数が混ざっていれば混合整数計画法（MIP）と呼ばれる。
[*6] 制約条件を満たす $\boldsymbol{x}$ は（可能）解と呼ばれ、目的函数を最小（大）化する $\boldsymbol{x}$ を最適解と呼ぶ
[*7] （図を見れば一目瞭然だが）超平面で囲まれた領域（凸領域）上の目的函数の最小（大）値を探す問題。

2 問題設定[*8]

2.1 研究上の設定

ここでは製品生産スケジューリング問題[*9] を考える。問題設定を以下にまとめる。

- 用語：オーダー o（工程)、タスク t（工程を成すための作業)、機械 r（タスクを行うための道具）
- オーダー o_i は一連のタスク系列 t_{ik} からなる。

$$o_i : t_{i1} \to \cdots \to t_{iK_i}$$

- 1 日に処理できるタスクは、オーダーごとに 1 タスクまで。[*10]
- 機械 r_a の 1 日辺りの処理時間には上限 C_a があり、その機械に同一作業日に割り当てられた複数のタスクの処理時間合計は、この上限を超えてはならない。
- オーダーには親子関係がある：子オーダー o_i の全てのタスクが終了すると親オーダー o_{P_i} の特定のタスク $t_{P_iL_i}$ に合流する。（オーダー間の親子関係は何階層あっても良い）
- オーダーには製造、計画オーダーがある：製造オーダー o_i には “納期 D_i” が、計画オーダー o_i には “リードタイム上限 Δ_i” が定められている [*11] 。

2.2 案件上の設定（概要）

- オーダー数：2 3 万, タスク数：8 タスク/1 オーダー
- 機械：4∼500
- 親子：半分とはいかないがかなりの量 $\to$ 4 階層程度
- 製造オーダー、計画オーダー：3,4 段階の優先度が設定
- 納期：厳守ではなく遅延の最小化
- 工程の内容には加工、組み立てなどのタスクがある
  $\to$ 同じ機械の組み立てに用いられる部品は近い日程で出来上がる様に設定
- 多目的最適化：
  - 納期遅延
  - 同機械部品近日程
  - 最長工期

[*8] 記法詳細は次節。
[*9] 各タスク t_{ik} を制約条件を満たしつつ最適なスケジュール（作業日）W_{ik} を割り当てる問題。
[*10] つまり各 t_{ik} に指定可能な作業日 w_s は 1 つだけ。$i.e.\ {}^\forall i, {}^\forall k, k'(k \neq k') \Rightarrow W_{ik} \neq W_{ik'}$
[*11] 納期 D_i は “日付” が割り当てられている。また Δ_i は “リードタイム上限” なので作業（してよい）“期間” の上限である。

3 記法

3.1 基本記法

- **オーダー o_i :**
  オーダー集合 $\mathcal{O} := \{o_i\}_{i=1,\cdots,N}$
- **タスク t_{ik} :**
  オーダー o_i のタスク集合 $\mathcal{T}[i] := \{t_{ik}\}_{k=1,\cdots,K_i}, \left(\mathcal{T} := \bigcup_{i=1}^{N} \mathcal{T}[i]\right)$
- **作業時間 c_{ik}** [*12] **:**
  オーダー o_i のタスク t_{ik} の作業時間 $c_{ik}, (c : t_{ik} \in \mathcal{T} \mapsto c_{ik} \in \mathbb{R}_{>0})$
- **機械 r_a, 機械の 1 日当たりの稼働時間上限 C_a :**
  機械集合 $\mathcal{R} := \{r_a\}_{a=1,\cdots,M}$, 機械の 1 日当たりの稼働時間上限集合 $\mathcal{C} := \{C_a\}_{a=1,\cdots,M}$
- **使用機械 R_{ik} :**
  オーダー o_i のタスク t_{ik} の使用機械 $R_{ik}, (R : t_{ik} \in \mathcal{T} \mapsto R_{ik} \in \mathcal{R})$
- **タスク t_{ik} の機械 r_a を用いた作業時間 C_{ika} :**
  $C_{ika} = \delta_{R_{ik},r_a} c_{ik}$ [*13]
- **作業日 w_s :**
  作業日 $\mathcal{W} := \{w_s\}_{s=1,\cdots,D}$ （D は全体の納期）
- 各インデックスの範囲 :
  $i = 1, \ldots, N, k = 1, \ldots, K_i, a = 1, \ldots, M, s = 1, \ldots, D$

3.2 割り当て

- **作業日 W_{ik} :**
  オーダー o_i のタスク t_{ik} の作業日 W_{ik}（Scheduling : $t_{ik} \in \mathcal{T} \mapsto W_{ik} \in \mathcal{W}$）

3.3 親子関係

- $\mathcal{S}_{\text{pa}} =$ { 親なしオーダーの index 集合 }
- $\mathcal{S}_{\text{ch}} =$ { 親ありオーダーの index 集合 }
- $\mathcal{S}_i =$ { オーダー o_iの子オーダー全体の index 集合 }
- オーダー o_i の親オーダー o_{P_i}
- オーダー o_i の親オーダー o_{P_i} と合流するタスクの index $= L_i \leq K_{P_i} \Rightarrow$ 合流タスク $= t_{P_i L_i}$

[*12] 単位は分を想定。

[*13] δ_{ij} は $i = j$ のとき 1、$i \neq j$ のとき 0 をとるクロネッカーのデルタと呼ばれる函数である。

3.4 オーダー分類

全ての親なしオーダー $o_i \in \mathcal{S}_{\mathrm{pa}}$ は、さらにその確度に応じて、製造オーダー、計画オーダーに分類される。

- $\mathcal{S}_{\mathrm{ma}} =$ { 製造オーダーの index 集合 }
- $\mathcal{S}_{\mathrm{pl}} =$ { 計画オーダーの index 集合 }
- $D_i =$ 製造オーダーの納期, $\quad i \in \mathcal{S}_{\mathrm{ma}}$
- Δ_i 計画オーダーのリードタイムの上限 , $\quad i \in \mathcal{S}_{\mathrm{pl}}$

3.5 入力データ形式[*14]

- `order_id` : i
- `task_id` : k
- `parent_id` : P_i [*15]
- `child_id` : 合流されるタスク t_{iL_i} に、合流する子オーダー `order_id` [*16]
- `wc_id` : R_{ik}
  - `upper_bound` : C_a
- `working_time` : c_{ik}
- `delivery_date` : D_i
- `working_date` : W_{ik} [*17]
- `size` : K_i
- `kind` : κ（オーダーの種類）[*18]

[*14] 7 節にデータ例を与えた。詳細はそちらを参照のこと。
[*15] 厳密解法では用いていないので無視して良い。
[*16] i, k を読み取って、子オーダー i' を見つける。ここが 0 なら $p_{iki'} = 0$。
[*17] 元々入っている値は “初期解”。厳密解法では用いていないので無視して良い。
[*18] 提案変数

第 II 部
定式化

4　整数計画法による定式化

以上の設定の下、整数計画法による定式化を行う[*19] 。

4.1　Binary 変数[*20]

整数計画法を適用する為の変数を導入する：

- t_{ik} の作業日 $W_{ik} = w_s$ の割り当ては、次式で定義される Binary 変数 X_{iks} によって記述できる：

$$X_{iks} = \delta_{W_{ik},w_s}{}^{*21}$$

4.2　目的函数[*22]

- 候補 1（最長工期）：

$$W_{\max} := \max_i W_{iK_i}$$

- 候補 2（リードタイム平均値）：平行移動対称性があり、この対称性を無くさないとうまく働かない

$$W_{\text{lead\_mean}} := \Delta\bar{W} := \frac{1}{N}\sum_{i=1}^{N}(W_{iK_i} - W_{i1})$$

- 候補 3（余剰時間平均値[*23] [*24] ）

$$W_{\text{surplus\_mean}} = -\bar{\epsilon} := -\frac{1}{N}\sum_{i=1}^{N}\epsilon_i := -\frac{1}{N}\sum_{i=1}^{N}(D_i - W_{iK_i})$$

- 候補 4（終了時間平均値）

$$W_{\text{mean}} := \frac{1}{N}\sum_{i=1}^{N}W_{iK_i}$$

[*19] Gurobi などの高速ソルバーを使えば整数計画法を高速に解くことができる。従って “整数計画法として問題を定式化する” ことが重要である。

[*20] “整数” 計画法は変数が整数の場合と Binary 変数（0, 1）の場合両方を含んでいる。

[*21] 便利のため作業日 $W_{ik} = \sum_{s=1}^{D} X_{iks}w_s$ を併用する（実装は X_{iks} で）。添字の数が減らせた方が高速だが減らせないか？

[*22] 実装は候補 1, 4 で行った。

[*23] オーダーの終了日 W_{iK_i} と納期 D_i の差分を “余剰時間” と定義し、これ（にマイナスをつけたもの）を最小化。

[*24] これは納期制約を無くせば “納期遅れ平均最小化” と等価であり、納期制約がなくても動く、が、$W_{\max}$ がしっかり小さくならないことがあるため要検討。

4.3 制約条件

- タスクの完備性[*25] ：

$$\sum_{s=1}^{D} X_{iks} = 1, \quad {}^{\forall}i, k$$

- オーダーのタスク順序制限 + 隣接タスク間隔上限[*26] ：

$$\Delta W_{\max} \geq W_{i(k+1)} - W_{ik} = \sum_{s=1}^{D} \left(X_{i(k+1)s} - X_{iks} \right) w_s \geq 1, \quad {}^{\forall}i, k$$

- オーダー間の親子制約[*27]：

$$W_{P_i L_i} - W_{iK_i} = \sum_{s=1}^{D} \left(X_{P_i L_i s} - X_{ik_i s} \right) w_s \geq 1, \quad {}^{\forall}i \in \mathcal{S}_{\mathrm{ch}}$$

- 製造オーダーの納期遵守[*28] ：

$$W_{iK_i} = \sum_{s=1}^{D} X_{iK_i s} w_s \leq D_i, \quad {}^{\forall}i \in \mathcal{S}_{\mathrm{pa}} \cap \mathcal{S}_{\mathrm{ma}}$$

- 機械の稼働時間制約[*29] ：

$$\sum_{i=1}^{N} \sum_{k=1}^{K_i} C_{ika} X_{iks} \leq C_a, \quad {}^{\forall}s$$

- 計画オーダーのリードタイム制約[*30] ：

$$\max_{j \in \{i\} \cup \mathcal{S}_i} \left(W_{iK_i} - W_{j1} \right) = \max_{j \in \{i\} \cup \mathcal{S}_i} \sum_{s=1}^{D} \left(X_{iK_i s} - X_{j1s} \right) w_s \leq \Delta_i, \quad {}^{\forall}i \in \mathcal{S}_{\mathrm{pa}} \cap \mathcal{S}_{\mathrm{pl}}$$[*31]

[*25] 全てのタスクに対して必ず何処かの作業日が 1 つ割り当てられる。

[*26] 後のタスク程、後の作業日が割り当てられる（$W_{i(k+1)} > W_{ik}$）。更に隣接タスク間隔が開きすぎないよう上限を設ける。

[*27] 子オーダーのタスク終了後、1 日以上空けて親タスクに合流する（$W_{P_i L_i} > W_{iK_i}$）。

[*28] 各製造オーダーには納期がある。

[*29] 機械についての 1 日あたりの稼働時間の和の上限。

[*30] 計画オーダーに明確な納期はない。しかしリードタイム上限 Δ_i が存在している。

[*31] 実装上は

$$\left(W_{iK_i} - W_{j1} \right) = \sum_{s=1}^{D} \left(X_{iK_i s} - X_{j1s} \right) w_s \leq \Delta_i, \quad {}^{\forall}i \in \mathcal{S}_{\mathrm{pa}} \cap \mathcal{S}_{\mathrm{pl}}, \quad {}^{\forall}j \in \mathcal{S}_i$$

でよい。

第 III 部

課題と実装

5　課題：親子関係, 計画オーダーについて

以上の実装にはオーダー間親子関係、計画オーダーが反映されていない。どうすれば反映できるか？

5.1　親子間オーダー $p_{iki'}$（実装済）

タスク t_{ik} に合流するオーダー $o_{i'}$ があれば 1、なければ 0 となるバイナリ “定数”$p_{iki'}$ を導入。オーダー間の親子制約は

$$p_{iL_ii'}\left(W_{iL_i} - W_{i'K_{i'}}\right) \geq p_{iL_ii'}$$

と表せる。$p_{iki'}$ はバイナリ定数なので実装上は

$$p_{iki'}\left(W_{ik} - W_{i'K_{i'}}\right) \geq p_{iki'}, \quad {}^{\forall}i, i', k$$

とすれば良い[*32] 。

5.2　オーダー種別 $T_{i\kappa}$ [*33] （未実装）

オーダー o_i が製造オーダーなら 0, 計画オーダーなら 1 となるバイナリ “定数”$T_{i\kappa}$ を導入。納期、及び、計画オーダーのリードタイム制限は共に

$$W_{iK_i} - T_{i\kappa}W_{i1} \leq D_i$$

と書ける[*34] 。

[*32]実際の実装では $p_{iki'}$ を定義し、制約条件を課す際に多重 `for` ループが必要になり、実行時間が長くなってしまう為、$p_{iki'}$ を用いずに工夫が必要になる。詳細は 8.3 参照のこと。

[*33]type の t をとった（task と被るが添字の内容が異なるし、kind, class, category,,, も被るので...）

[*34]納期 $W_{iK_i} \leq D_i$, リードタイム制約 $W_{iK_i} - W_{i1} \leq \Delta_i$

6 実装[*35]

6.1 フォルダ構造：

```
mip_src/
 ├ mip.py
 ├ input/
 |   ├ file_name.csv
 |   └ file_name_r.csv
 └ output/
     └ (file_name)_(n_orders)_(condition)_(objective_function)
          ├ (file_name)_(n_orders)_(condition)_(objective_function)_solution.csv
          ├ (file_name)_(n_orders)_(condition)_(objective_function)_schedule.csv
          └ (file_name)_(n_orders)_(condition)_(objective_function)_capacity.csv
```

6.2 入力, 出力データ

- input ファイル：
  - データ名（file_name）
  - オーダー数（n_orders）
  - 納期型式（condition）
    - d：通常の納期 $W_{iK_i} \leq D_i$
    - nd：納期のスケール変換 $W_{iK_i} \leq d_i := D_{\min} + (D_{\max} - D_{\min}) \frac{D_i - \mathrm{n}_{\mathrm{dead\_min}}}{\mathrm{n}_{\mathrm{dead\_max}} - \mathrm{n}_{\mathrm{dead\_min}}}$
  - 目的函数（objective_function）
    - max：最長工期 $W_{\max}$
    - lead_mean：納期遅れ平均 $W_{\mathrm{lead\_mean}}$
    - surplus_mean：余剰時間平均 $W_{\mathrm{surplus\_mean}}$
    - mean：平均終了工期 W_{mean}
- output ファイル：
  - データ名（file_name）、オーダー数（n_orders）、納期形式まで共通。
  - solution：X_{iks} から定義される t_{ik} の作業日程表
  - schedule：機械 r_a の作業日 w_s に処理するタスク t_{ik} の表
  - capacity：機械 r_a の作業日 w_s における総処理時間 C_{as} の表

[*35] TEX の仕様でコメントがずれて表示されている箇所があるので注意。

6.3 設定：

```
#オーダー数設定#########
n = 1000
#納期調整#####
D_min = D_min #最小値
D_max = D_max #最大値
alpha = alpha
#input, ファイル名用ラベル output
#d -> D[i] + alpha
#nd -> D_min + (D_max−D_min)*(D[i]−n_dead_min)/(n_dead_max−n_dead_min)
name = 'file_name'
condition = 'condition'
objective = 'objective_function' #max or lead_max or surplus or mean
```

6.4 データセットの準備：

```
#####input, パス output#####
import os
input_path = './input/' + name + '.csv'
output_folder = './output/' + name + '_' + str(n) + '_' + condition + '_' + objective
os.makedirs(output_folder, exist_ok=True)
output_path = output_folder + '/output_' + name + '_' + str(n) + '_' + condition + '_' +
    objective
#データ読み込み#########
import pandas as pd
df = pd.read_csv(input_path)
df = df[df['order_id'] <= n ]
df.loc[df['child_id'] > n, 'child_id'] = 0
df_r = pd.read_csv('./input/' + name + '_r.csv')
#高速化用#########
df_child = df.loc[df['child_id'] > 0]
order_list = df_child['order_id'].values.tolist()
task_list = df_child['task_id'].values.tolist()
child_list = df_child['child_id'].values.tolist()
```

6.5 定数設定：

```
import numpy as np
#パラメータ#########
n_orders = df['order_id'].max() # オーダー数（）N
n_task_min = df['size'].min() # タスク数の最小値（min ）K_i
n_task_max = df['size'].max() # タスク数の最大値（max ）K_i
n_child = df[df['child_id'] > 0]['child_id'].count() #合流タスク数 (num of L_i)
n_dead_max = df['delivery_date'].max() #オーダーごとの納期の最大値（max ）D_i
n_dead_min = df['delivery_date'].min() # オーダーごとの納期の最小値（min ）D_i
n_resources = df['wc_id'].max() # リソース数 (M)
if condition == 'nd':
    n_days = D_max # 作業日数（=）n_dead_max
else:
    n_days = df['delivery_date'].max() + alpha
#タスク数 K_i, 納期 D_i:
df_sd = df[['order_id', 'size', 'delivery_date']].drop_duplicates().reset_index()
K = []
D = []
for i in range(n_orders):
    K.append(df_sd['size'][i])
    D.append(df_sd['delivery_date'][i])
#の作業時間 t_ikc_ij, 機械割り当て r_ij, R_ija, 親子関係 p_iki':
c = np.zeros((n_orders, n_task_max))
r = np.zeros((n_orders, n_task_max))
#p = np.zeros((n_orders, n_task_max, n_orders)) 親子間オーダー（#に合流するを指定する）t_iko_i
C = np.zeros((n_orders, n_task_max, n_resources)) # オーダー×タスクの作業時間（初期化）
R = {} # オーダー×タスクごとのリソース（） dictionary
l = 0
for i in range(n_orders):
    for k in range(K[i]):
        c[i][k] = df[(df['order_id']==i+1)&(df['task_id']==k+1)]['working_time'][l]
        r[i][k] = df[(df['order_id']==i+1)&(df['task_id']==k+1)]['wc_id'][l] − 1
        #for j in range(n_orders):
        # if df[(df['order_id']==i+1)&(df['task_id']==k+1)]['child_id'][l] == j + 1:
        # p[i][k][j] = 1
        C[i, k, int(r[i][k])] = c[i][k]
        key = str(i) + '_' + str(k)
        R[key] = int(r[i][k])
        l += 1
#日あたりの機械使用上限：1
C_max = [] #C_a
for a in range(n_resources):
    C_max.append(df_r['upper_bound'][a])
#日程:
w = list(range(n_days))
```

6.6 PuLP による最適化：

```
import pulp
from pulp import GUROBI_CMD
#最適化問題の設定#########
prob = pulp.LpProblem('scheduling', pulp.LpMinimize)
##### 最適化する変数の設定#####
#バイナリ変数## X_{iks}###
X = pulp.LpVariable.dicts('X', (range(n_orders), range(n_task_max), range(n_days)), cat='Binary')
if objective == 'max':
    #最長工期## W_max###
    W_max = pulp.LpVariable('W_max', lowBound=0)
elif objective == 'lead_mean':
    #リードタイム平均## W_lead_mean###
    W_mean = pulp.LpVariable('W_lead_mean', lowBound=0)
elif objective == 'surplus_mean':
    #余剰時間平均## W_surplus_mean###
    W_mean = pulp.LpVariable('W_surplus_mean', lowBound=0)
elif objective == 'mean':
    #終了日の平均## W_mean###
    W_mean = pulp.LpVariable('W_mean', lowBound=0)
#制約条件の設定#########
#タスクの完備性#####
for i in range(n_orders):
    for k in range(K[i]):
        prob += pulp.lpSum(X[i][k][s] for s in range(n_days)) == 1
#オーダーのタスク順序制約#####
for i in range(n_orders):
    for k in range(K[i] - 1):
        prob += pulp.lpSum((X[i][k+1][s] - X[i][k][s]) * w[s] for s in range(n_days)) >= 1
#オーダーの納期制約#####
d = []
for i in range(n_orders):
    if condition == 'd':
        d.append(D[i]+alpha)
    elif condition == 'nd':
        d.append( D_min + ( (D_max −D_min) * ( ( D[i] − n_dead_min ) / ( n_dead_max − n_dead_min ) ) ) )
    prob += pulp.lpSum(X[i][K[i]−1][s] * w[s] for s in range(n_days)) <= int(d[i])
#機械の稼働時間制約#####
for a in range(n_resources):
    for s in range(n_days):
        prob += pulp.lpSum( C[i, k, a] * X[i][k][s] for i in range(n_orders) for k in range(K[i])) <= C_max[a]
#オーダー親子間順序制約#####
#for i in range(n_orders):
# for k in range(K[i]):
```

```
# for j in range(n_orders):
# prob += pulp.lpSum((X[i][k][s] - X[j][K[j]-1][s]) * w[s] * p[i][k][j] for s in range(n_days)) >= p
    [i][k][j]
for i in range(len(order_list)):
    prob += pulp.lpSum((X[order_list[i]-1][task_list[i]-1][s] - X[child_list[i]-1][K[child_list[i
        ]-1]-1][s]) * w[s] for s in range(n_days)) >= 1
#####
#隣接タスク間隔上限#####
#for i in range(n_orders):
for i in range(n_orders):
    for k in range(K[i] - 1):
        prob += pulp.lpSum((X[i][k+1][s] - X[i][k][s]) * w[s] for s in range(n_days)) <= 10
#####
#目的函数の設定#########
if objective == 'max':
    #最大工期の不等式制約による定義#####
    for i in range(n_orders):
        prob += pulp.lpSum(X[i][K[i]- 1][s] * w[s] for s in range(n_days)) <= W_max
    prob += W_max #最大工期を目的函数に設定
elif objective == 'lead_mean':
    #リードタイム平均の不等式制約による定義#####
    prob += pulp.lpSum(pulp.lpSum((X[i][K[i]-1][s] -X[i][0][s])* w[s] / n_orders for s in range(
        n_days)) for i in range(n_orders)) == W_lead_mean
    prob += W_mean #リードタイム平均を目的函数に設定
elif objective == 'surplus_mean':
    #余剰時間平均の不等式制約による定義#####
    prob += pulp.lpSum(pulp.lpSum((X[i][K[i]-1][s] * w[s] - d[i]) / n_orders for s in range(n_days
        )) for i in range(n_orders)) == W_surplus_mean
    prob += W_mean #終了日の平均を目的函数に設定
elif objective == 'mean':
    #終了日の平均の不等式制約による定義#####
    prob += pulp.lpSum(pulp.lpSum(X[i][K[i]-1][s] * w[s] / n_orders for s in range(n_days)) for i
        in range(n_orders)) == W_mean
    prob += W_mean #終了日の平均を目的函数に設定
#求解の実行#########
#実行時間の測定
print(pulp.LpStatus[prob.solve()])
#print(pulp.LpStatus[prob.solve(GUROBI_CMD())]) #用 Gurobi
if objective == 'max':
    #最大工数
    print('Obj␣:␣W_max␣=␣', W_max.value() + 1)
elif objective == 'lead_mean':
    #リードタイム平均時間
    print('Obj␣:␣W_lead_mean␣=␣', W_lead_mean.value() +1)
elif objective == 'surplus_mean':
    #余剰時間平均時間
    print('Obj␣:␣W_surplus_mean␣=␣', W_surplus_mean.value() +1)
elif objective == 'mean':
    #終了平均時間
    print('Obj␣:␣W_mean␣=␣', W_mean.value() +1)
```

6.7 最適解のテーブル出力（オーダー, タスク, 実施日）

```
out_df = pd.DataFrame(columns=['order_id', 'task_id', 'wc_id', 'day']) #最適解出力用
out_dic = {} #機械 r_a 実施日 w_s の組からタスクを出力する辞書 t_ika_s −> i_k
key_list = [] #キー out_dic
count=0
for i in range(n_orders): #オーダー
    for k in range(K[i]): #タスク
        for s in range(n_days): #実施日
            if X[i][k][s].value() == 1: #日程が存在しているとき以下を実行
                tmp_se = pd.Series([i+1, k+1, 1+np.argmax(C[i,k,:]), s+1], index=out_df.columns,
                    name=str(count)) #t_ik, C[i,kの最大を取るインデックス,:] −> ノンゼロのインデックス a, をメモ s
                out_df = out_df.append(tmp_se) #メモした値をに out_df
                key = str(1+np.argmax(C[i,k,:])) + '_' + str(s+1) #の機械、実施日を（）t_ikaskeya_s
                value = str(i+1) + '_' + str(k+1) #値を i_k
                if key in key_list: #のかぶりがあったらカンマをつけて a_s', i_kを追加'a_s −> i_k, i'_k',,,,
                    out_dic[key] += ', ' + value
                else: #のかぶりがなかったら a_s'i_kを追加'
                    out_dic[key] = value
                    key_list.append(key)
                count += 1
out_df.to_csv(output_path + '_solution.csv') #出力
print('out_df', out_df)

w_max = out_df['day'].max()
w_mean = out_df.groupby('order_id').max()['day'].mean()
print('W_max = ', w_max)
print('W_mean = ', w_mean)
```

6.8 最適解によるスケジュール表（割り当て）

```
import math
schedule_columns = list(np.array(['w_' + str(i+1) for i in range(w_max)])) #スケジュール表のフィールド
capacity_columns = list(np.hstack([np.array(['w_' + str(i+1) for i in range(w_max)]), 'C_max'])) #で配列の列結合（スケジュール日程 hstack+の表を C_max
schedule_tbl = np.array(['w_' + str(i+1) for i in range(w_max)])
capacity_tbl = np.hstack([np.array(['w_' + str(i+1) for i in range(w_max)]), 'C_max'])
for a in range(n_resources): #機械ごと
    temp_list1 = []
    temp_list2 = []
    for s in range(w_max): #日程ごと
        key = str(a+1) + '_' + str(s+1)
        amnt = 0
        if key in out_dic.keys(): #が登録されていたら、に keyindex_list', ごとに分割し、 'index を登録 i_k
            temp = out_dic[key]
            index_list = temp.split(',␣')
            for i in range(len(index_list)): #ごと i_k
                sub_list = index_list[i].split('_') #i, を分割 k
                sub1 = int(sub_list[0]) − 1 # i
                sub2 = int(sub_list[1]) − 1 # k
                amnt += C[sub1,sub2,a] # i, k, の合計（の実施日の合計値を算出）ar_aw_sC_as
        else: #登録されていなかったら空っぽ
            temp = ''
        temp_list1.append(temp) # を追加 index
        temp_list2.append(amnt) # i, k, を追加 a
    temp_list2.append(C_max[a]) #を追加 C_max
    temp_list1 = np.array(temp_list1) #np.に変換 array
    schedule_tbl = np.vstack([schedule_tbl, temp_list1]) #で配列の行結合 vstack
    capacity_tbl = np.vstack([capacity_tbl, temp_list2])
schedule_tbl = np.delete(schedule_tbl, 0, 0)
capacity_tbl = np.delete(capacity_tbl, 0, 0)
#名 index, フィールド名
header1 = schedule_columns
header2 = capacity_columns
resource_name = ['r_' + str(a+1) for a in range(n_resources)]
#から型へ listDataFrame
schedule_df = pd.DataFrame(schedule_tbl, columns=header1, index=resource_name)
capacity_df = pd.DataFrame(capacity_tbl, columns=header2, index=resource_name)
#print('schedule_df', schedule_df)
#print('capacity_df', capacity_df)
schedule_df.to_csv(output_path + '_schedule.csv') #出力
capacity_df.to_csv(output_path + '_capacity.csv') #出力
```

6.9 検算

```
import pandas as pd
import matplotlib.pyplot as plt
from pylab import rcParams
# タスクの完備性
calc_list1 = []
for i in range(n_orders):
    for k in range(K[i]):
        tmp = 0
        for s in range(n_days):
            tmp += X[i][k][s].value()
        calc_list1.append(int(tmp))
# オーダーのタスク順序制約
calc_list2 = []
for i in range(n_orders):
    tmp = 0
    for k in range(K[i] - 1):
        for s in range(n_days):
            tmp += (X[i][k+1][s].value() - X[i][k][s].value()) * w[s]
        tmp -= 1
    calc_list2.append(int(tmp))
# オーダーの納期制約
calc_list3 = []
for i in range(n_orders):
    tmp =0
    for s in range(n_days):
        tmp += X[i][K[i]-1][s].value() * w[s]
    if condition == 'd':
        calc_list3.append(int(D[i] - tmp))
    elif condition == 'nd':
        calc_list3.append(int(d[i] - tmp))
    elif condition == 'k':
        calc_list3.append(int(D[i] + K[i] - tmp))
    elif condition == 'n':
        calc_list3.append(int(D[i] + n_orders - tmp))
# 機械の稼働時間制約
calc_list4 = []
for a in range(n_resources):
    for s in range(n_days):
        tmp = 0
        for i in range(n_orders):
            for k in range(K[i]):
                tmp += C[i,k,a] * X[i][k][s].value()
        if tmp > 0:
            calc_list4.append(int(C_max[a] - tmp))
# オーダー親子間順序制約
calc_list5 = []
```

```
#for i in range(n_orders):
# for k in range(K[i]):
# for j in range(n_orders):
# tmp = 0
# for s in range(n_days):
# tmp += (X[i][k][s].value() - X[j][K[j]-1][s].value()) * w[s] * p[i][k][j]
# if p[i][k][j] > 0:
# calc_list5.append(int(tmp - p[i][k][j]))
for i in range(len(order_list)):
    tmp = 0
    for s in range(n_days):
        tmp += (X[order_list[i]-1][task_list[i]-1][s].value() - X[child_list[i]-1][K[child_list[i
            ]-1]-1][s].value()) * w[s]
    calc_list5.append(int(tmp - 1))
titles = ['completeness of tasks', 'ordering of tasks', 'time to delivery', 'resource
    capacity', 'ordering of parent']
xlabels = ['number of days', 'interval between tasks', 'number of days', 'margin capacity
    ', 'interval between tasks']
ylabels = ['number of tasks', 'number of orders', 'number of orders', 'cumulative number
    of resources', 'number of orders']
lists = [calc_list1, calc_list2, calc_list3, calc_list4, calc_list5]
fig = plt.figure(figsize=(12,3))
for i in range(len(lists)):
  ax1 = fig.add_subplot(1,5,i+1)
  ax1.grid(True, which='major')
  ax1.hist(lists[i], bins=10, alpha=0.5, histtype='stepfilled', color='b')
  ax1.set_title(titles[i])
  ax1.set_ylabel(ylabels[i])
  ax1.set_xlabel(xlabels[i])
plt.tight_layout()
#plt.show()
fig.savefig(output_path + '_graph.png')
```

7　入出力データの例

7.1　入力

7.1.1　df

オーダー (order_id)、タスク (task_id)、合流オーダー (child_id)、利用機械 (wc_id)、納期 (delivery_date)、タスク数 (size)。

	order_id	task_id	child_id	wc_id	working_time	delivery_date	size
0	1	1	0	3	10	50	2
1	1	2	2	2	10	50	2
2	2	1	0	1	10	40	3
3	2	2	3	2	10	40	3
4	2	3	0	1	10	40	3
5	3	1	0	3	10	30	2
6	3	2	0	1	10	30	2

7.1.2　df_r

機械 (wc_id)、稼働時間上限 (upper_bound)。

	wc_id	upper_bound
0	1	20
1	2	20
2	3	20

7.1.3　df_resources

df から C_{ika}（od_i,task_k の利用機械 wc_id）を抜き出した表。

	task_1	task_2	task_3
od_1	3	2	NA
od_2	1	2	1
od_3	3	1	NA

7.1.4　df_c

df から c_{ij}（od_i,task_k の稼働時間 working_time）を抜き出した表。

	task_1	task_2	task_3
od_1	10.0	10.0	0.0
od_2	10.0	10.0	10.0
od_3	10.0	10.0	0.0

7.2 出力

7.2.1 df_solution

`order_id,task_id.wc_id` の作業日 (`day`) 表。

	order_id	task_id	wc_id	day
0	1	1	3	1
1	1	2	2	5
2	2	1	1	1
3	2	2	2	3
4	2	3	1	4
5	3	1	3	1
6	3	2	1	2

7.2.2 df_schedule

実際の機械 (`r_a`) の作業日 (`w_s`) に行うタスク (`(od_id)_(task_id)`) 表。

	w_1	w_2	w_3	w_4	w_5
r_1	2_1	3_2		2_3	
r_2			2_2		1_2
r_3	1_1,3_1				

7.2.3 df_capacity

実際の機械 (`r_a`) の作業日 (`w_s`) に行う作業時間 ($\sum_{i=1}^{N}\sum_{k=1}^{K_i} C_{ika}X_{iks}$) 表。一番左の列に稼働時間上限 `upper_bound` が書いてあり上限を超えていないか一眼で確認ができる。

	w_1	w_2	w_3	w_4	w_5	C_max
r_1	10.0	10.0	0.0	10.0	0.0	20.0
r_2	0.0	0.0	10.0	0.0	10.0	20.0
r_3	20.0	0.0	0.0	0.0	0.0	20.0

第 IV 部

実験メモ

8　実験メモ

8.1　オーダー数と処理時間の関係

- オーダー数と処理時間の関係は $T = e^{an_{orders}}$ とすると
  - T_cbc：a = 1.141
  - T_gurobi：a = 0.420

  となる。
- CBC は最初は大人しい振る舞いをするが 500 オーダーで数時間かけても処理できなかった。
- Gurobi は 1800 オーダー までは大人しい振る舞いをするが 1900 オーダーで急激に処理時間が増加し、また、2000 オーダー（1950 オーダー でも）ではエラーで処理できなかった。

オーダー数変化

オーダー数	T_cbc	T_total	W_max	W_mean	T_gurobi	T_total	W_max	W_mean	T_cbc/T_gurobi
100	11.49	25.75	26	22.17	2.2	27.02	26	12.71	5.222727273
200	26.69	49.25	26	21.65	4.66	53.74	26	12.23	5.727467811
300	56.44	94.23	26	21	3.57	47.04	26	11.68	15.80952381
400	206.93	259.43	28	22.84	5.4	61.76	28	11.63	38.32037037
500					21.05	114.69	39	28.87	0
600					42.72	153.43	39	33.02	0
700					83.47	216.27	39	33.24	0
800					97.55	243.84	39	33.09	0
900					87.85	246.18	39	32.6	0
1000					95.84	276.61	39	31.58	0
1100					110.13	319.59	39	32.36	0
1200					153.64	376.23	39	31.99	0
1300					184.42	427.68	39	32.78	0
1400					223.84	514.1	39	32.93	0
1500					363.14	640.34	39	33.57	0
1600					879.94	1187.13	39	33.99	0
1700					580.24	904.92	39	33.95	0
1800					771.39	1112.29	39	33.93	0
1900					2738.99	2782	39	34.49	0
2000									0

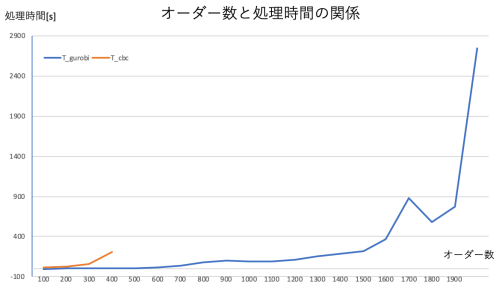


図 3　100-1900 オーダー までのオーダー数と処理時間の関係

8.2 CBC と Gurobi の比較

- CBC は W_mean が高速（平均 2.42 倍）。
- Gurobi は目的函数で処理速度に差はなし（平均 0.98 倍）[*36] 。
- W_max はソルバー、目的函数関係なく結果が一致した。
- W_mean は CBC は大きく、Gurobi は小さく出た[*37]
- CBC と Gurobi の処理時間は W_max で平均 8.4 倍、W_mean で平均 3.31 倍となった[*38] 。
- ΔT は `for` 文などの実行時間でソルバーの能力に関係ないはず。しかし Gurobi サーバーの数値の方が大きい。つまりサーバーの計算速度は自前 PC より遅いようだ[*39] 。

CBC

W_max

	W_max	W_mean	T_cbc	T_total	ΔT
data_100			未了		
25	25.00	21.90	16.05	27.35	11.30
17	28.00	23.71	20.30	33.38	13.08
9	40.00	32.86	61.01	77.75	16.74
2000_1000			未了		

W_mean

	W_max	W_mean	T_cbc	T_total	ΔT
data_100			未了		
25	25.00	10.23	12.87	23.79	10.92
17	28.00	11.15	11.30	23.50	12.20
9	40.00	11.75	14.46	32.46	18.00
2000_1000			未了		

Gurobi

W_max

	W_max	W_mean	T_gurobi	T_total	ΔT
data_100	42.00	28.34	4.23	27.69	23.46
25	25.00	10.45	3.70	39.50	35.80
17	28.00	11.15	3.74	41.02	37.28
9	40.00	11.75	3.95	43.80	39.85
2000_1000	39.00	33.08	272.31	471.85	199.54

W_mean

	W_max	W_mean	T_gurobi	T_total	ΔT
data_100			未了		
25	25.00	10.23	3.72	38.91	35.19
17	28.00	11.15	3.80	41.39	37.59
9	40.00	11.75	4.12	42.60	38.48
2000_1000			結果出ず		

図 4　Solver 別 目的函数別 ファイル別 W_max, W_mean, T_solver, T_total

	CBC: T_max / T_mean
	0
	1.25
	1.80
	4.22
	0
平均	2.42

	W_max: T_cbc/T_gurobi
	0
	4.34
	5.43
	15.45
	0
平均	**8.40**

	Gurobi: T_max / T_mean
	0
	0.99
	0.98
	0.96
	0
平均	0.98

	W_mean: T_cbc/T_gurobi
	0
	3.46
	2.97
	3.51
	0
平均	**3.31**

図 5　Solver 別 目的函数別 ファイル別 W_max, W_mean, T_solver, T_total

[*36] しかし 2000_1000 において W_max で結果が出るものが、W_mean で結果が出ない。
[*37] つまり、何も厳密であるはずだが、解が一意的でないことの影響が垣間見える。
[*38] 但し、納期を一律最適な値に設定している為、CBC でも処理速度が高速化されている。
[*39] その意味では同一スペックマシンであれば Gurobi の速度差はより大きいと考えて良い様だ。

8.3 目的函数別比較

	W_max	W_mean	W_lead_mean → 開始日を設定？	W_surplus_mean（納期あり）	W_surplus_mean（納期なし）
025（100オーダー, 20機械） 025.csv 22 KB 025_r.csv 170 B	W_max = 25 W_mean = 15.29 Time_G = 12.78[s] Time_tot = 49.08[s] f_25_100_d_max.zip 49 KB	W_max = 25 W_mean = 10.23 Time_G = 13.91[s] Time_tot = 48.90[s] f_25_100_d_mean.zip 48 KB	W_max = 90 W_mean = 35.44 Time_G = 60.31[s] Time_tot = 99.23[s] f_025_100_d_lead_mean.zip 50 KB	W_max = 25 W_mean = 10.23 Time_G = 13.86[s] Time_tot =50.98[s] f_025_100_d_surplus_mean_nouki.zip 47 KB	W_max = 25 W_mean = 10.23 Time_G = 12.58[s] Time_tot = 50.16[s] f_025_100_d_surplus_mean.zip 47 KB
017（100オーダー, 20機械） 017.csv 23 KB 017_r.csv 170 B	W_max = 28 W_mean = 12.94 Time_G = 11.49[s] Time_tot = 51.88[s] f_017_100_d_max.zip 51 KB	W_max = 28 W_mean = 11.15 Time_G = 9.38[s] Time_tot = 49.51[s] f_017_100_d_mean.zip 52 KB	W_max = W_mean = Time_G = [s] Time_tot = [s]	W_max = 28 W_mean = 11.15 Time_G = 9.40 Time_tot = 50.64[s] f_017_100_d_surplus_mean_nouki.zip 51 KB	W_max = **90*** W_mean = **11.77** Time_G = 10.25[s] Time_tot = 51.47[s] f_017_100_d_surplus_mean_nouki.zip 51 KB
009（100オーダー, 20機械） 009.csv 24 KB 009_r.csv 170 B	W_max = 40 W_mean = 16.93 Time_G = 15.56[s] Time_tot = 58.32[s] f_009_100_d_max.zip 49 KB	W_max = 40 W_mean = 11.75 Time_G = 10.65[s] Time_tot = 52.30[s] f_009_100_d_mean.zip 51 KB	W_max = W_mean = Time_G = [s] Time_tot = [s]	W_max = 40 W_mean = 11.75 Time_G = 10.31[s] Time_tot = 52.04[s] f_009_100_d_surplus_mean_nouki.zip 51 KB	W_max = **70*** W_mean = **12.19** Time_G = 10.63[s] Time_tot = 53.77[s] f_009_100_d_surplus_mean.zip 52 KB
2000_1000（1000オーダー, 20機械, 納期一律100） 2000_1000.csv 306 KB 2000_1000_r.csv 306 KB	W_max = 39 W_mean = 29.68 Time_G = 143.89[s] Time_tot =344.35[s] f_2000_1000_nd_max.zip 137 KB	W_max = W_mean = Time_G = [s] Time_tot = [s] **-> 13575s実行の末結果出ず（エラー）**	W_max = W_mean = Time_G = [s] Time_tot = [s]	W_max = W_mean = Time_G = Time_tot = **-> 12951s実行の末結果出ず（エラー）**	W_max = W_mean = Time_G = Time_tot = 納期ありより時間がかかると考え実行せず
46_1300（1300オーダー, 15機械） 46_1300.csv 398 KB 46_1300_r.csv 398 KB	W_max = 37 W_mean = 31.75 Time_G = 770.05[s] Time_tot =1037.82[s] f_46_1300_d_max.zip 161 KB	W_max = W_mean = Time_G = Time_tot = **-> 24000s実行の末結果出ず（各ステップに時間がかかりエラーも出ず）**	W_max = W_mean = Time_G = Time_tot =	W_max = W_mean = Time_G = Time_tot = **-> 24000s実行の末結果出ず（各ステップに時間がかかりエラーも出ず）**	W_max = W_mean = Time_G = Time_tot = 納期ありより時間がかかると考え実行せず

- `W_surplus_mean` は隣接タスク間隔上限を強めることで性能が良くなることがあるが物足りない：
  - 017：隣接上限=5 で `W_max`=39, `W_mean`=11.33, 隣接上限=2 で `W_max`=37, `W_mean`=11.26
  - 009：隣接上限=5 で `W_max`=57, `W_mean`=12.06, 隣接上限=2 で `W_max`=40, `W_mean`=11.81
- やはり問題サイズが大きくなると `W_max` くらいしか動かない。

8.4 高速化の工夫

8.4.1 納期の変換[*40]

納期設定は処理時間に影響を及ぼす。理由は少なくとも 2 つ。まず変数の数。最適化変数 X_{iks} の数は、n_orders × n_task_min × n_days であるから、納期の最大値である n_days が小さければ変数が少なくなり、処理時間が高速化される（はず）。次に自由度。1 つ目の理由と似ているが、最適な納期を探索する際、納期がゆるいと探索領域が広がり、探索に時間が掛かってしまう（はず）[*41] [*42] 。

8.4.2 多重 for ループの除去

Gurobi は非常に高速であることがわかった。しかし、Gurobi の処理が数秒でも、プログラム全体の実行時間が数十分から数時間かかることもしばしばで、その理由は親子間オーダーを与える $p_{iki'}$ の多重 `for` ループの処理時間であると考えられる。以下にこれを解消する方法について記す。

- $p_{iki'}$ の利用停止：

  $p_{iki'}$ は n_orders × n_task_max × n_orders ループ必要になる[*43] ため、利用を停止し、代わりにリスト、

  `df_child=df.loc[df['child_id']>0]`

  `order_list=df_child['order_id'].values.tolist()`

  `task_list=df_child['task_id'].values.tolist()`

  `child_list=df_child['child_id'].values.tolist()` を定義すれば、

  $$p_{iki'}=1 \quad \Rightarrow \quad [i,k,i']=[\text{order\_list[m]},\text{task\_list[m]},\text{child\_list[m]}], \quad {}^{\exists}m \in \text{range(len(order\_list))}$$

  となるから、

```
for m in range(len(order_list)):
    prob += pulp.lpSum((X[order_list[i]-1][task_list[i]-1][s]
                -X[child_list[i]-1][K[child_list[i]-1]-1][s])*w[s] for s in range(n_days)) <= 1
```

  によって親子間制約を課すことができる[*44] 。とすると、$p_{iki'}$ 定義の `for` と、制約条件の `for` がなくなるため高速化される。

[*40] Gurobi では必要ないかもしれない。

[*41] （あるファイルでは）元々の納期設定ではオーダー数を 20 に制限しても数時間かけても結果が出なかった。そこで上記のように全納期を-40 して初めて 842 秒で実行完了し。更に-50 して 577 秒。更にそこから条件を `nd` に変更し $D_{\min}=\frac{1}{2}D_{\max}=$ n_orders することで 10 秒まで改善された。ただしこの結果は CBC での結果である。

[*42] 納期の変換については 6.2 を参照のこと。

[*43] n_orders $=2000$, n_task_max $=15$ とすると 6×10^7 ループ。

[*44] 実はこの方法は高速化前の解と高速化後の解が全く同じものにはならない。しかし目的函数は一致する。一考の余地あり。

第 V 部

課題

9　課題

- Binary 変数の設定（添字の数減らせる？）や、制約条件（隣接タスク間隔上限、のように効率を上げられる制約条件は他にないか）、目的函数（結局 $W_{\max}$ しかうまく機能しない）は適切か。
- 整数計画法と多目的最適化？ $\sum_i \gamma_i W_i$ みたいにできる？（重要なほど重みを大きくし、できると思う by 魚井さん）
- 余剰時間制約を目的函数にして納期制約をなくせば問題が緩和されいつでも解ける？（できる、と思ったが $W_{\max}$ がやたらと大きくなることがあるなど、まだ問題あり）
- Gurobi の問題か不明だが、オーダー数の変化に対して急激に処理時間が変化する、また、問題の難易度（実行不可能？）によって処理時間が変化する為、問題によって処理時間が予測できない（？）
- ソルバーごとの目的函数の得意不得意（？）
- 謎のエラー　→　どうも実行不可能？

10　メモ

- 「条件をキツくする-¿探索領域が狭くなる」と単純に考えていたが、線型計画法だと領域の頂点に最適解が存在することから、条件設定が目的函数の値に直接影響する（一般の凸計画なら領域内部に帯域的最適解が存在する場合があり、それならば条件がきついほど探索領域が狭くなり、処理速度が上がる（はず））。
- しかし、`W_max` などは納期設定を変化させてもしっかり同じ値に最適化されているので、ここはどう捉えるか？

第 VI 部
Appendix

11 Appendix A. PuLP メモ [6][7][8]

最適化問題をコンピュータを用いて解かせるには CBC [*45] 、Gurobi [*46] など[*47] のソルバーが必要になる。これらのソルバーを利用するにはモデラーが必要になる。ここではモデラー “PuLP” [*48] を用いて最適化を行う。

手順

- **PuLP** のインポート：

```
import pulp
```

- （線型計画）問題オブジェクトを作る：

```
problem = pulp.LpProblem('Problem Name', option)
```

`option` は `pulp.LpMaximize` で最大化、`pulp.LpMinimize` で最小化。

- 変数オブジェクトを作る：[*49]

```
var = pulp.LpVariable.dicts('VAR', ([1,2], ['a', 'b']), 0, 1, 'Continuous')
```

  - 変数名の prefix (Ex. 'VAR')
  - 変数名の postfix のリストのタプル（変数で与えることができる）(Ex. ([1,2], [' a' , ' b']))
  - 変数の最小値 (Ex. 0)
  - 変数の最大値 (Ex. 1)
  - 変数の種類（'Continuous'：連続変数, 'Integer'：離散変数）

- 目的函数を作る（問題名に加算する形で定義）：

```
problem += 変数の線型函数
```

右辺は変数オブジェクトで書かれた目的函数が入る。

[*45] `https://projects.coin-or.org/Cbc` などを参照のこと。無料。

[*46] `https://www.focus-s.com/focus-s/media/1-Gurobi.pdf` などを参照のこと。優秀だが有料。

[*47] 他にも `https://qiita.com/keisukesato-ac/items/0049f33e10aab9d87734` に依れば癖はあるが規模が大きい問題を解かせるにはソルバー “MIPCL” が “使い物になる” らしい。一考の余地あり。

[*48] `https://pypi.org/project/PuLP/`

[*49] このコードを実行すると、最小値 0, 最大値 1, で連続値をとる変数群 `VAR_1_a,VAR_1_b,VAR_2_a,VAR_2_b` を持つ辞書が手に入る。アクセス方法は `var[1]['a']` など。

- 制約条件を作る（問題名に加算する形で定義）：

```
problem += 変数の線型函数>= 0
problem += 変数の線型函数== 0
```

右辺は変数オブジェクトで書かれた制約式が入る。

- 解く：

```
problem.solve()
```

制約条件によっては解けない場合があるので、結果を知りたいときは、

```
status = problem.solve()
print(pulp.LpStatus[status])
```

とし、出力が `Optimal` となることを確認すれば良い[*50] 。

- 結果：（変数名 `.value()` で変数の値を取り出せる）

```
print(var[1][a].value())
```

[*50]制約条件を満たせない時、出力が実行不可能 `Infeasible` となる。

12 Appendix B. Gurobi メモ

- 数理最適化モデラーで数理最適化をモデリング済みであるとする。
- Gurobi Optimizer は、数理最適化（線形計画法／整数計画法）の最新技術を取り入れた、最高性能の線形／整数計画ソルバーです。数理最適化においてだけではなく、マルチコアプロセッサーへも対応など、最新のハードウェア性能を充分に発揮できるように設計されています[*51] 。
- Gurobi 7.02 が 2 台のサーバーにインストール済み
  - IP: 172.16.1.25（開発用）
  - IP: 172.16.1.28（本番用）
- 社内、或いは、BP Backup VPN に繋いだ状態で利用可能（手続き必要なし）
- ssh でサーバーにアクセス[*52] ：
  - `sshuser_id@IP_address`
    `user_pass`
- パスを通す → .bashrc を編集：
  - `vi~/.bashrc`
  - i で編集モード。以下を最下部に貼り付けたら `esc,:wq` で保存。
    `exportGUROBI_HOME="/opt/gurobi702/linux64"`
    `exportPATH="${PATH}:${GUROBI_HOME}/bin"`
    `exportLD_LIBRARY_PATH="${LD_LIBRARY_PATH}${GUROBI_HOME}/lib"`
  - `source~/.bashrc` [*53]
- sftp でファイルをアップロード[*54] ；
  - `sftpuser_id@IP_address`
    `user_pass`
  - `putfile_name`
- `pulp.LpStatus[prob.solve()]` を `pulp.LpStatus[prob.solve(GUROBI_CMD)` に変更[*55]

[*51] `http://www.engineering-eye.com/GUROBI/` より抜粋。画像は省略。
[*52] `user_id,user_pass` は自分の PC のものと同じ。
[*53] 編集結果を反映。再度 ssh でログインしても良い。
[*54] 元々ユーザーディレクトリにいるので mkdir, cd などを用いて必要なディレクトリ構造を作成、必要な位置に移動しファイルをアップロードすること。
[*55] PuLP で実装するより Gurobipy で実装する方が変数、係数などの定義で `for` 文を使わずに済み、高速化が見込めるらしい。

13　Appendix C. Gurobi で厳密解が得られないときの対策

13.1　概要：

Gurobi で（混合）整数計画問題を扱う際、問題（～ 入力ファイル）のサイズが小さくても、簡単に（時間がかかったりエラーが出たり）厳密解が得られないことがよくある。このときの対処法を考えてみる。

13.2　対処法：

13.2.1　1. 制約条件を強くする（探索空間を小さくする）

制約条件は $A\boldsymbol{x} \leq \boldsymbol{b}$ の形で与えられており、単純には $\boldsymbol{b}$ が小さければ可能な $\boldsymbol{x}$ の範囲が狭くなり、探索にかかる時間が小さくなる[*56]

例：今回扱った問題（AI プラクティス部研究手伝い（スケジューリング最適化））であれば、納期や機械稼働時間上限などで与えられる定数の制約がつくので、それらを”スケール変換”し、小さくした。

$$b' = b'_{\min} + \frac{(b'_{\max} - b'_{\min})}{(b_{\max} - b_{\min})}(b - b_{\min})$$

$$[b_{\min}, b_{\max}] \to [b'_{\min}, b'_{\max}]$$

探索空間が大きい時に比べ適切な探索空間に設定すると数十倍処理時間が小さくなることも（あった）。簡単の為 $b'_{\min} = b'_{\max}$ としてもよい。

13.2.2　2. 制約条件を弱くする（探索空間を大きくし可能解を与える）

1. の場合（或いはそもそも）制約条件が強すぎて可能解が存在しない場合がある。その場合は制約条件を弱くする、即ち (1) 式を用いて定数の範囲を広げ可能解を探す。

例：今回扱った問題であれば納期や機械稼働時間上限などが”あまりにきつく”設定されているとファイルサイズに関わらず問題を解く時間が大きくなることも（あった）。そこで一律で納期を緩めたところ（それでも同サイズの問題の処理時間に比べ数千倍時間がかかっているが）厳密解を求めることができた。

制約条件を弱くすることで厳密解が得られた場合、実務上お客様に、納期の再設定や、機械の導入などを提案する必要があるかもしれない。

13.2.3　3. 1, 2 のある種の組み合わせ

制約条件をキツくし可能解が存在しなくともその中で目的函数の極値の値を見つけ出す。　→　目的函数の値付近に制約条件を設定する。

[*56] 簡単の為 $\boldsymbol{x}$ は下限を持つ変数とし $\boldsymbol{b}$ は正とした。

13.2.4　4. 目的函数を吟味する

一般に（混合）整数計画法は解の一意性が保証されていない。従って用いるソルバーに依っても得られる解が異なることがある（勿論目的函数の最適値は一致する）。また、用いるソルバーによっても、目的函数の得意、不得意がある（かもしれない）。

例：今回扱った問題であれば最終作業日（$W\_\max$）、作業終了日平均（W_mean）を目的函数に設定したが以下のような知見が得られた：

- CBC ソルバーでは W_mean が、Gurobi ソルバーでは $W\_\max$ が得意であるようだった
- $W\_\max$ を目的函数に設定すると CBC ソルバーと Gurobi ソルバーで W_mean の値が異なった。
- W_mean を目的函数に設定するとどちらのソルバーでも $W\_\max$, W_mean の値は一致し、また、$W\_\max$ を目的函数に設定した場合と $W\_\max$ が一致した。

ここから分かることは、ソルバーに応じてうまい目的函数を設定すること、また、解の一意性を保証するようなうまい目的函数を設定することが重要であるかもしれない。→　$W\_\max$ の様な大域的情報を目的函数に設定すると、一般に解の一意性が保証されないので、W_mean のような局所的情報を目的函数に設定すると良いかもしれない、が、Gurobi は W_mean を得意としていないように見えるので、もっと良い目的函数を設定する必要があるかもしれない（制約条件さえ満たせば良いなら $W\_\max$ で十分だが）。

13.2.5　5. ソルバーを見直す

（某氏の話によれば）Gurobi は混合整数計画問題には向いているが、単なる整数計画問題には向いていないかもしれない[*57] 。

13.2.6　6. 解法を考える（案） →　1,2,3 の別案

メタヒューリスティクスで解く。

→　緩和問題に可能解がなければ、どこの制約条件がきついかを調べ、それを緩めたい。

→　メタヒューリスティクスで解くと制約条件（例えばペナルティ函数の形で）が満たされなかったり大局的極解になっていないかもしれないが一旦解は出てくる。その解を制約条件の観点から見直し、制約条件を満たせるように、制約条件を緩める。

例：今回扱った問題であれば `20200630\_SA`（改良版）実施結果（`2000orders,20workcenters`）にあるようなヒートマップを書かせ、納期制約を守れていないオーダーを探したり、機械稼働上限を守れていない機械を探し、超えているものを緩める。

[*57] `https://www.msi-jp.com/localsolver/`, `http://www.orsj.or.jp/whatisor/admission.html`

14　Appendix D. 整数計画法の解法 [2][3] [*58]

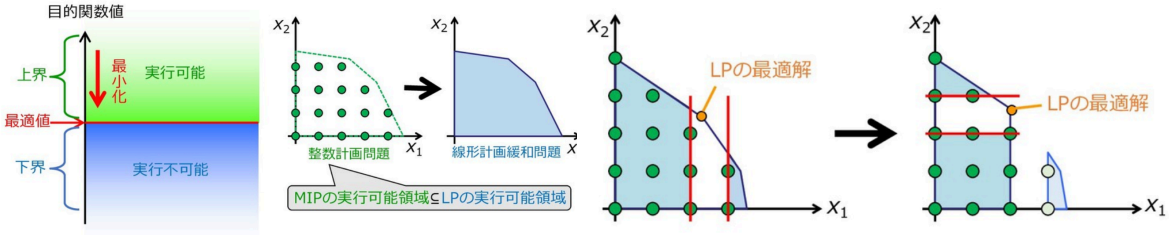
14.1　整数計画問題を解く

- 計算困難な最適化問題では最適値が簡単に求まらない。
- 最適解の上界と下界を求めて最適値の取り得る範囲を限定する。
  実行可能な範囲と実行不可能な範囲から挟んでゆく[*59] 。
- 緩和問題：原問題の一部の制約条件の緩めた問題。
  整数変数の整数条件を外す　→　線型計画緩和問題[*60]

整数計画問題の実行可能領域 $\subseteq$ 線型計画緩和問題の実行可能領域

14.2　分枝限定法

- 最適化問題に対する汎用的な厳密解法。
- 分枝操作：
  実行可能解を分割して部分問題を生成する[*61] 。
- 限定操作：
  緩和問題から得られる下界値を用いて**最適解が得られる見込みのない部分問題**を省く[*62] 。



[*58] 今の所 [2][3] の抜粋。

[*59] 最小化問題なら実行可能解から上界を定め、実行不可能解から下界を定め、収束先を探す。

[*60] 一般に線型計画法は凸超多面体領域内部の線型目的函数の極値問題なので（詳細は省略するが）凸超多面体の頂点に最適解が存在することが知られている。従って効率的に解を求めるアルゴリズムも存在しており線型緩和問題から整数計画問題の最適解の “当たり” をつける。

[*61] 緩和問題の解が実数値をとる変数 x を選んで $x \leq \lfloor \bar{x} \rfloor$ を追加した問題と $x \geq \lceil \bar{x} \rceil$ を追加した問題に分割する。ここで $\bar{x}$ は緩和解、$\lfloor \bar{x} \rfloor$ は床函数、$\lceil \bar{x} \rceil$ は天井函数である。

[*62] 分枝前の下界値 $\bar{z} \geq$ 分枝後の下界値 $\min\{\bar{z}_1, \bar{z}_2\}$。ここで $\bar{z}_i$ は部分問題 $i, i = 1, 2$ の下界。

参考文献

[1] 藤江 哲也, 整数計画法による定式化入門, オペレーションズリサーチ.
http://web.tuat.ac.jp/~miya/fujie_ORSJ.pdf

[2] 梅谷 俊治, 60 分で学ぶ数理最適化
https://speakerdeck.com/umepon/mathematical-optimization-in-60-minutes

[3] 梅谷 俊治, 整数計画問題の定式化と解法
https://speakerdeck.com/umepon/an-introduction-to-integer-programming

[4] 斉藤 努, データ分析ライブラリーを用いた 最適化モデルの作り方 (Python による 問題解決シリーズ), 近代科学社 (2018).

[5] 久保 幹雄, あたらしい数理最適化: Python 言語と Gurobi で解く, 近代科学社（2012）.

[6] PuLP による線型計画問題の解き方ことはじめ：
https://qiita.com/mzmttks/items/82ea3a51e4dbea8fbc17

[7] PuLP によるモデル作成方法：
https://docs.pyq.jp/python/math_opt/pulp.html

[8] PuLP による数理最適化超入門：
http://www.nct9.ne.jp/m_hiroi/light/pulp01.html

[9] 斎藤 努, データ分析ライブラリーを用いた最適化モデルの作り方, オペレーションズ・リサーチ
https://www.orsj.or.jp/archive2/or63-12/or63_12_784.pdf

[10] 品野 勇治，藤井 浩一, 使ってみよう線形計画ソルバ, オペレーションズ・リサーチ
http://www.orsj.or.jp/archive2/or64-4/or64_4_238.pdf

[11] 岩瀬 弘和, ジョブの納期の異なるフローショップスケジューリング問題, 東京成徳大学研究紀要 —人文学部・応用心理学部— 第 17 号（2010）
https://tsu.ac.jp/Portals/0/research/22/081-102.pdf

[12] 上野 信行, 矢後 諒智, 川崎 雅也, 奥原 浩之, 内示情報を用いた生産計画,
https://www.jstage.jst.go.jp/article/jasmin/2009f/0/2009f_0_7/_pdf