HW2-Integer Linear Programming

Cloth Problem

Analysis

- Variable
  - 第i个车间生产第j种布料的单位利润，记为r_{ij}
  - 第j种布料的单位价格，记为a_j
- Decision Variable
  - 第i个车间生产第j种布料的米数，记为x_{ij}
  - $x_{ij}\begin{cases} i=1,2,3,4,5 \\ j=1,2,3,4,5,6 \end{cases}$
- Objective Function
  - 总利润，记为z
  - $max \quad \sum_{j=1}^{6}\sum_{i=1}^{5} r_{ij}\cdot x_{ij}$
- Constraints
  - 显性约束
    1. 总资金共400000
       - $\sum_{j=1}^{6}\sum_{i=1}^{5} x_{ij}\cdot a_j \le 400000$
    2. 每个车间生产每种布料的长度均大于1000
       - $x_{ij}\ge 1000 \begin{cases} i=1,2,3,4,5 \\ j=1,2,3,4,5,6 \end{cases}$
    3. 每个车间的生产米数上限为10000
       - $\sum_{j=1}^{6} x_{ij} \le 10000 \quad i=1,2,..,6$
  - 隐性约束
    1. 生产米数为整数且大于0
       - $x_{ij}\in \mathbf{Z}^+ = \begin{cases} i=1,2,3,4,5 \\ j=1,2,3,4,5,6 \end{cases}$

Model

$$max \quad \sum_{j=1}^{6}\sum_{i=1}^{5} r_{ij}\cdot x_{ij}$$

$$s.t. \begin{cases} \sum_{j=1}^{6}\sum_{i=1}^{5} x_{ij}\cdot a_j \le 400000 \\ \sum_{j=1}^{6} x_{ij} \le 10000 \quad i=1,2,..,6 \\ x_{ij}\ge 1000 \\ x_{ij}\in \mathbf{Z}^+ \end{cases}$$

Solve

- Data
  - Profit of producing

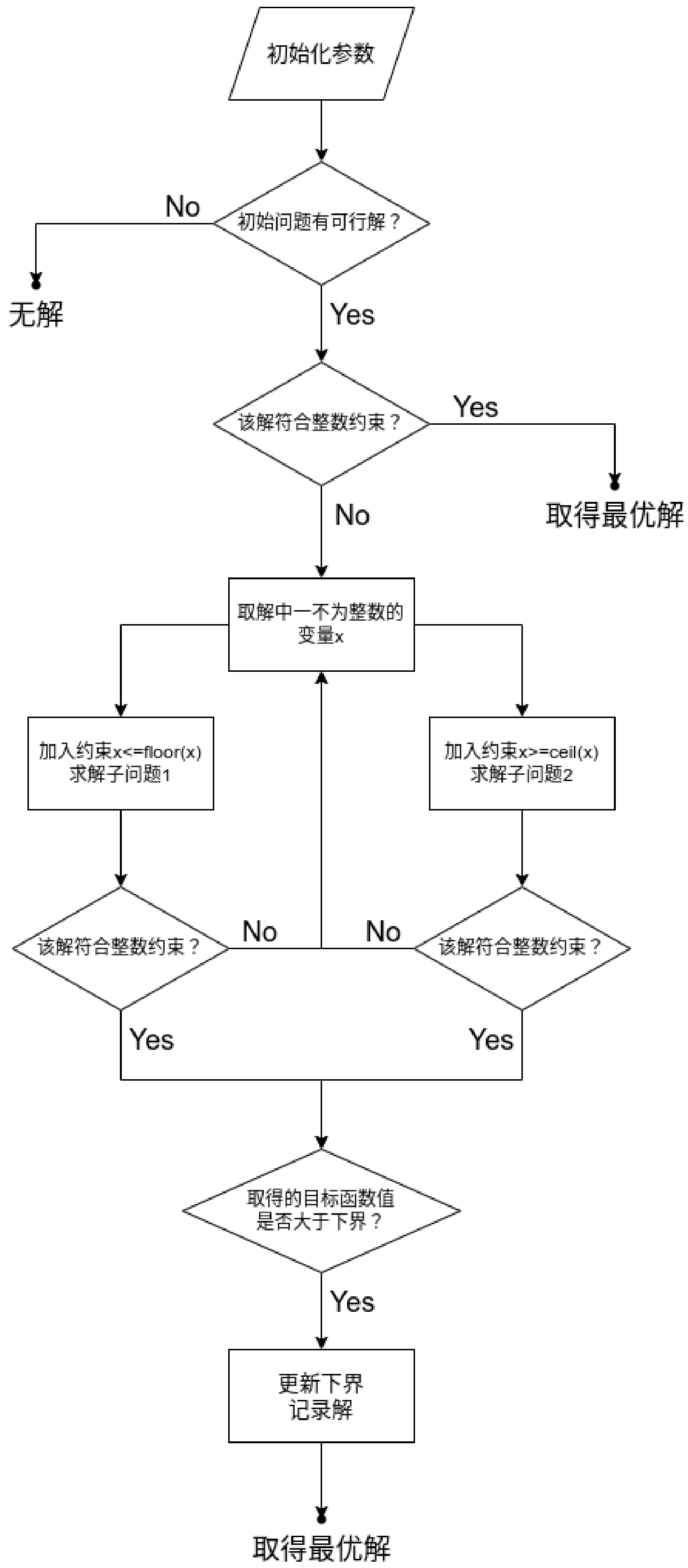
r_{ij}	1	2	3	4	5	6
1	4	3	4	4	5	6
2	3	4	5	3	4	5
3	5	3	4	5	5	4
4	3	3	4	4	6	6
5	3	3	3	4	5	7

  - Cost of cloth

    以c_{11}为例，往返的油耗为$c_{11}=100+450\times0.5+450\times0.25+100=537.5$，其余计算均同理，结果列于下表

	1	2	3	4	5	6
c_j	6	6	7	8	9	10

- Branch and Bound Algorithm
  - Flow Chart



  - 我使用了python的PuLP库进行线性规划的求解，具体代码细节已经给了详细的注释。不过在调整代码的过程中，我发现了PuLP库的一个bug，在这里进行记录，我之后再验证一下，如果问题可复现，我到github上发个issue，问题如下：

当我把两个已求解的子问题`problem_low`和`problem_up`送入由`queue.Queue()`创造的一个实例后，我将它取出(branch后的子问题求解)，我发现这两个子问题的解改变了！而`problem_low.status`和`problem_up.status`的值仍然等于1(求解成功的标识符)，这浪费了我一个小时去检查自己的代码和调试，我十分生气，具体过程如图：

- Before pushing into the queue

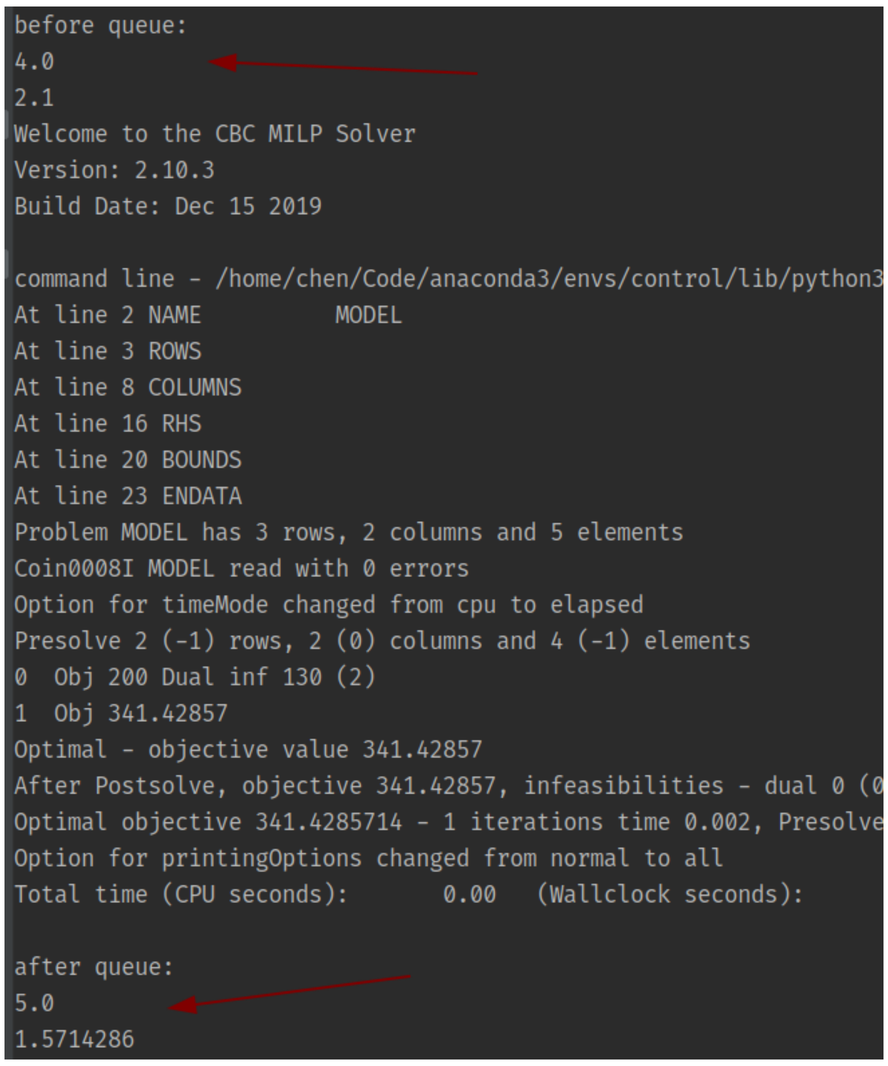
```
## 分支出子问题并求解
prob_low = prob_now.deepcopy()
prob_up = prob_now.deepcopy()
prob_low += x[v_i-1][v_j-1] <= side_low
prob_up += x[v_i-1][v_j-1] >= side_up
prob_low.solve()
prob_up.solve()

## 将可行子问题加入队列
if prob_low.status == 1:
Q.put(prob_low)
if prob_up.status == 1:
Q.put(prob_up)
```

- After poping from the queue

```
prob_now = Q.get()
```

- The different between the two solution

  这是我为了演示而使用的一个例子，我已经把这个sample代码贴到附录



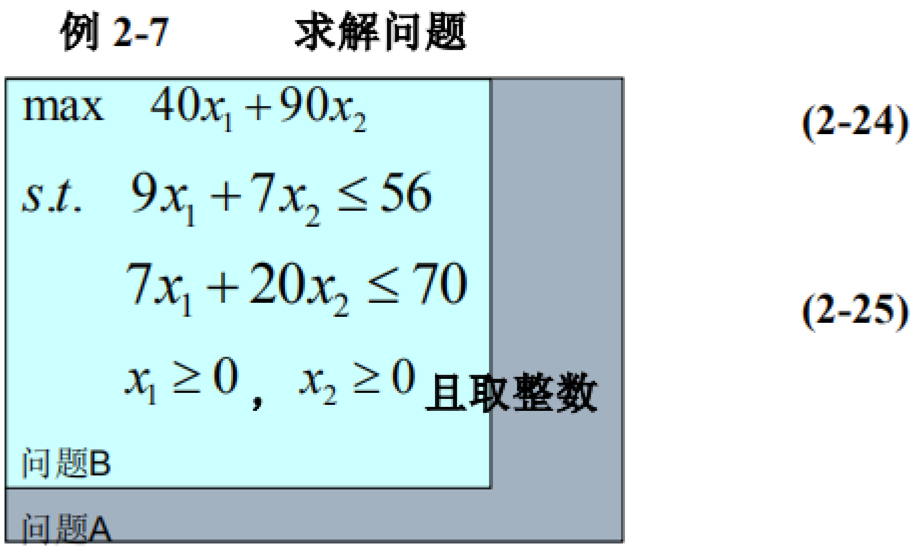
总之，我觉得这是一个非常神奇的bug，我空了也去扒一下PuLP的变量逻辑。文字可能描述不大清楚，我不摸鱼的时候尝试录个屏

- Result
  - Output

x_{ij}	1	2	3	4	5	6
1	1000	1000	1000	1000	1000	5000
2	1000	1000	5000	1000	1000	1000
3	5000	1000	1000	1000	1000	1000
4	1000	1000	1000	1000	5000	1000
5	1000	1000	1000	1000	1000	5000

  我用自带的整数规划做了一下，与此一致，也和书上的答案一致

  - 一般来说到这里就over了，但我想对比下线性规划和整数规划的区别，于是直接求了下松弛问题，然后发现连续变量解出来就是整数，也就是这个样例根本没有用到分支定界法的核心就结束了。这让我对自己代码的正确性感到惶恐，因此我使用了如下例子来验证我的代码：



  在调试的过程中我确实发现了不少bug(包括上面提到的那个)，最后调试出的输出为$x_1=4\quad x_2=2$，与答案一致。我也尝试在几个LP问题上加入整数约束，比较我写的算法和内部求解器的输出有误差异，结果均正确，代码附于附录(即上面提到的测试代码)

Knapsack problem

Analysis

- Variable
  - 第j件物品的重量，记为a_j
  - 第j件物品的价值，记为c_j
- Decision Variable
  - 是否装第j件物品，记为x_j
  - $x_j\begin{cases}1 & 装了第j件物品\\ 0 & 没装第j件物品\end{cases}\quad j=1,2,..,n$
- Objective Function
  - 背包中物品的最大价值，记为z
  - $max\quad z=\sum_{j=1}^{n}c_j\cdot x_j$
- Constraints
  - 显性约束
    1. 背包物品总质量小于b
       - $\sum_{j=1}^{n}x_j\cdot a_j\le b$
  - 隐性约束
    1. x_j为$0-1$变量

- $x_j \in \{0,1\} \quad j = 1,2,..,n$

Model

$$max \quad z = \sum_{j=1}^{n} c_j \cdot x_j$$

$$s.t. \begin{cases} \sum_{j=1}^{n} x_j \cdot a_j \le b \\ x_j \in \{0,1\} \end{cases}$$

Solve

- Data
  - Items

序号	1	2	3	4	5
质量	1	2	3	4	5
价值	2	4	4	5	6

  - Knapsack
    - 容量为6

- Algorithm
  - Pulp自带整数规划求解器

```
Knapsack_Problem:
MAXIMIZE
2*x1 + 4*x2 + 4*x3 + 5*x4 + 6*x5 + 0
SUBJECT TO
_C1: x1 + 2 x2 + 3 x3 + 4 x4 + 5 x5 <= 6

VARIABLES
0 <= x1 <= 1 Integer
0 <= x2 <= 1 Integer
0 <= x3 <= 1 Integer
0 <= x4 <= 1 Integer
0 <= x5 <= 1 Integer
```

  - 上题中写的分支定界算法(已附于附录)

- Result

x1	x2	x3	x4	x5
1	1	1	0	0

手工验证一下，发现一致

附录

Prob1 Code

服装厂问题的分支定界算法

这个算法只是按课本上的思路顺序写下来，没有任何剪枝，因此十分原始。。有空的话我加一些优化

```
import numpy as np
from pulp import *
import queue

if __name__ == "__main__":
    ## data
    m, n = 5, 6                          # i上限m, j上限n
    r = [
        [4,3,4,4,5,6],
        [3,4,5,3,4,5],
        [5,3,4,5,5,4],
        [3,3,4,4,6,6],
        [3,3,3,4,5,7]
    ]                                    # 第i个车间生产第j种布料的利润(r_ij)

    a = [6,6,7,8,9,10]                   # 第j种布料的单价(a_j)

    fund = 4e5                           # 总资金
    cloth_min, cloth_max = 1e3, 1e4      # 加工上下限


    ## Problem
    prob = LpProblem('Cloth_Distribution', LpMaximize)
    ## Variables
    x = [ [LpVariable("x"+str(i+1)+str(j+1),lowBound=cloth_min) for j in
range(n)] for i in range(m) ]
    ## Objective Function
    obj = lpSum(r[i][j]*x[i][j] for j in range(n) for i in range(m))
    prob += obj, 'Objective Function'
    ## Fixed Constraints
    prob += lpSum(x[i][j]*a[j] for i in range(m) for j in range(n)) <= fund
    for i in range(m):
        prob += lpSum(x[i][j] for j in range(n)) <= cloth_max


    ## Branch and Bound
    # 初始化
    low_bound = 0            # 初始下界,一定大于0
    best_x = None
    Q = queue.Queue()
    # 解初始问题
    prob.solve()
    if prob.status !=1:
        raise ValueError('Insoluble!')       # 无可行解
    else:
        Q.put(prob)                          # 可解,放入队列
```

```python
# 主递归流程
def _BB(Q, low_bound, opt_max, best_x):
    lb = low_bound
    om = opt_max
    # 循环遍历所有子问题
    while not Q.empty():
        prob_now = Q.get()
        prob_now.solve()
        obj_now = value(prob_now.objective)

        # 若该区域的最大值小于下界,则直接排除
        if obj_now < lb:
            continue

        # 遍历,寻找第一个非整数解
        flag = 1
        for v in prob_now.variables():
            if not value(v).is_integer():
                flag = 0
                break

        # 整数解,与下界比较更新
        if flag == 1:
            if lb < obj_now:
                lb = obj_now
            if om is None or obj_now > om:
                om = obj_now
                best_x = [value(v) for v in prob_now.variables()]

        # 非整数解,需要分枝
        else:
            branch_v = None
            for v in prob_now.variables():
                if not value(v).is_integer():
                    branch_v = v
                    break

            # 新约束
            side_low = np.floor(value(branch_v))
            side_up = np.ceil(value(branch_v))
            v_i, v_j = int(str(branch_v)[1]), int(str(branch_v)[2])

            # 子问题
            prob_low = prob_now.deepcopy()
            prob_up = prob_now.deepcopy()
            prob_low += x[v_i-1][v_j-1] <= side_low
            prob_up += x[v_i-1][v_j-1] >= side_up

            # 求解
```

```python
            prob_low.solve()
            prob_up.solve()

            # 加队列
            if prob_low.status == 1:
                Q.put(prob_low)
            if prob_up.status == 1:
                Q.put(prob_up)

        return best_x, om

    x,obj = _BB(Q,low_bound,up_bound, best_x)
    print("x:",x)
    print("obj:",obj)
```

Prob1 Code Test

分支定界法的测试代码

```python
import numpy as np
from pulp import *
import queue

if __name__ == "__main__":

    ## Problem
    prob = LpProblem('Test', LpMaximize)
    ## Variables
    eps = np.finfo(np.float64).eps
    x1 = LpVariable("x1",lowBound=0+eps)
    x2 = LpVariable("x2",lowBound=0+eps)
    x = [x1, x2]
    ## Objective Function
    obj = 40*x1 + 90*x2
    prob += obj, 'Objective Function'
    ## Fixed Constraints
    prob += 9*x1 + 7*x2 <= 56
    prob += 7*x1 + 20*x2 <= 70


    ## Branch and Bound
    # 初始化
    low_bound = 0            # 初始下界,一定大于0
    opt_max = None
    best_x = None
    Q = queue.Queue()
    # 解初始问题
    prob.solve()
    if prob.status !=1:
        raise ValueError('Insoluble!')       # 无可行解
```

```python
    else:
        Q.put(prob)                                  # 可解,放入队列

# 主递归流程
def _BB(Q, low_bound, opt_max, best_x):
    lb = low_bound
    om = opt_max
    # 循环遍历所有子问题
    while not Q.empty():
        prob_now = Q.get()
        """注意这行代码注释与否的效果"""
        # prob_now.solve()

        # print("after queue:")
        # for v in prob_now.variables():
        #     print(value(v))
        obj_now = value(prob_now.objective)

        # 若该区域的最大值小于下界,则直接排除
        if obj_now < lb:
            continue

        # 遍历,寻找第一个非整数解
        flag = 1
        for v in prob_now.variables():
            tmp = value(v)
            if not value(v).is_integer():
                flag = 0
                break

        # 整数解,与下界比较更新
        if flag == 1:
            if lb < obj_now:
                lb = obj_now
            if om is None or obj_now > om:
                om = obj_now
                best_x = [value(v) for v in prob_now.variables()]

        # 非整数解,需要分枝
        else:
            branch_v = None
            for v in prob_now.variables():
                if not value(v).is_integer():
                    branch_v = v
                    break
            # 新约束
            side_low = np.floor(value(branch_v))
            side_up = np.ceil(value(branch_v))
            v_i = int(str(branch_v)[1])
```

```
                # 子问题
                prob_low = prob_now.deepcopy()
                prob_up = prob_now.deepcopy()
                prob_low += x[v_i-1] <= side_low
                prob_up += x[v_i-1] >= side_up

                # 求解
                prob_low.solve()
                # print("before queue:")
                # for v in prob_low.variables():
                #     print(value(v))
                prob_up.solve()

                # 加队列
                if prob_low.status == 1:
                    Q.put(prob_low)
                if prob_up.status == 1:
                    Q.put(prob_up)

        return best_x, om

    x,obj = _BB(Q,low_bound,opt_max, best_x)
    print("x:",x)
    print("obj:",obj)
```

Prob2 Code with internal solver

背包问题的算法（内部求解器）

```
from pulp import *
import numpy as np

## Data
n = 5                           # n个物品(j)
a = [1,2,3,4,5]                 # 第j个物品的质量(a_j)
c = [2,4,4,5,6]                 # 第j个物品的价值(c_j)

b = 6                           # 背包容积

## Problem
problem = LpProblem('Knapsack_Problem', LpMaximize)
## Variables
x = [LpVariable('x'+str(j+1),cat=LpBinary) for j in range(n)]
## Objective Function
obj = lpSum(c[j]*x[j] for j in range(n))
problem += obj, 'Objective Function'
## Constraints
problem += lpSum(x[j]*a[j] for j in range(n)) <= b

## Solve
```

```
print(problem)
problem.solve()

## Print
for v in problem.variables():
    print(value(v))
```

Prob2 Code with my BB solver

背包问题的算法（我的分支定界算法）

```
from pulp import *
import numpy as np
import queue

## Data
n = 5                               # n个物品(j)
a = [1,2,3,4,5]                     # 第j个物品的质量(a_j)
c = [2,4,4,5,6]                     # 第j个物品的价值(c_j)

b = 6                               # 背包容积

## Problem
problem = LpProblem('Knapsack_Problem', LpMaximize)
## Variables
x = [LpVariable('x'+str(j+1),lowBound=0) for j in range(n)]
## Objective Function
obj = lpSum(c[j]*x[j] for j in range(n))
problem += obj, 'Objective Function'
## Constraints
problem += lpSum(x[j]*a[j] for j in range(n)) <= b
for j in range(n):
    problem += x[j]<=1



## Solve
# 初始化
low_bound = 0               # 初始下界,一定大于0
opt_max = None
best_x = None
Q = queue.Queue()
# 解初始问题
problem.solve()


if problem.status !=1:
    raise ValueError('Insoluble!')
else:
    Q.put(problem)
```

```python
# 主递归流程
def _BB(Q, low_bound, opt_max, best_x):
    lb = low_bound
    om = opt_max
    # 循环遍历所有子问题
    while not Q.empty():
        prob_now = Q.get()
        """注意这行代码注释与否的效果"""
        prob_now.solve()
        obj_now = value(prob_now.objective)

        # 若该区域的最大值小于下界,则直接排除
        if obj_now < lb:
            continue

        # 遍历,寻找第一个非整数解
        flag = 1
        for v in prob_now.variables():
            tmp = value(v)
            if not value(v).is_integer():
                flag = 0
                break

        # 整数解,与下界比较更新
        if flag == 1:
            if lb < obj_now:
                lb = obj_now
            if om is None or obj_now > om:
                om = obj_now
                best_x = [value(v) for v in prob_now.variables()]

        # 非整数解,需要分枝
        else:
            branch_v = None
            for v in prob_now.variables():
                if not value(v).is_integer():
                    branch_v = v
                    break
            # 新约束
            side_low = np.floor(value(branch_v))
            side_up = np.ceil(value(branch_v))
            v_i = int(str(branch_v)[1])

            # 子问题
            prob_low = prob_now.deepcopy()
            prob_up = prob_now.deepcopy()
            prob_low += x[v_i-1] <= side_low
            prob_up += x[v_i-1] >= side_up

            # 求解
```

```
            prob_low.solve()
            prob_up.solve()

            # 加队列
            if prob_low.status == 1:
                Q.put(prob_low)
            if prob_up.status == 1:
                Q.put(prob_up)

    return best_x, om

x,obj = _BB(Q,low_bound,opt_max, best_x)
print("x:", x)
print("obj:", obj)
```