МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра Математического обеспечения и применения ЭВМ

ОТЧЁТ

по лабораторной работе № 1
по курсу «Практикум по ПМОиРД»

Выполнили:
студенты группы КТмо2-3
Шепель И.О.
Куприянова А.А.

Проверила:
преподаватель каф. МОП ЭВМ
Пирская Л.В.

Оценка

____

«____» ___________ 2017 г.

Таганрог 2017

Оглавление

1. Постановка задачи

В соответствии с вариантом №6 ставится следующая задача:

На швейной фабрике для изготовления четырех видов изделий может быть использована ткань трех артикулов. Нормы расхода тканей всех артикулов на пошив одного изделия, имеющееся в распоряжении фабрики общее количество тканей каждого артикула и цена одного изделия данного вида представлены в таблице 1.

Таблица 1.

Артикул ткани	Норма расхода ткани, м, на одно изделие вида				Общее количество ткани, м
	1	2	3	4	
I	1	–	2	1	180
II	–	1	3	2	210
III	4	2	–	4	800
Цена одного изделия, р.	9	6	4	7	

Определить, сколько изделий каждого вида должна произвести фабрика, чтобы стоимость изготовленной продукции была максимальной.

2. Математическая модель

2.1. Переменные задачи

x_1, x_2, x_3, x_4 – свободные переменные (количество изготавливаемых изделий).

x_5, x_6, x_7 – базисные переменные.

$f = 9x_1 + 6x_2 + 4x_3 + 7x_4$ – функция цели (стоимость изготовленной продукции).

$c = (c_1, c_2, c_3, c_4, c_5, c_6, c_7) = (9, 6, 4, 7, 0, 0, 0)$ – вектор весовых коэффициентов (исходя из цены каждого изделия).

$A_0 = (180, 210, 800)$ – вектор ограничений (исходя из общего количества ткани).

$a = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 & 0 & 0 \\ 0 & 1 & 3 & 2 & 0 & 1 & 0 \\ 4 & 2 & 0 & 4 & 0 & 0 & 1 \end{pmatrix}$ – матрица норм расхода (матрица условий).

2.2. Ограничения

1. Количество изделий неотрицательно:

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0,\ x_4 \geq 0.$$

2. Ограничения, следующие из ограниченности ресурсов (ткани):

$$\begin{cases} x_1 + 2x_2 + x_4 \le 180, \\ x_2 + 2x_3 + 2x_4 \le 210, \\ 4x_1 + 2x_2 + 4x_4 \le 800. \end{cases}$$

2.3. Цель задачи

Цель задачи состоит в том, чтобы найти такой вектор x=(x_1, x_2, x_3, x_4, x_5, x_6, x_7), при котором значение f максимально, то есть f = c·x→max.

2.4. Приведение задачи к канонической форме

Чтобы преобразовать в уравнения неравенства ограничений пункта 2 параграфа 2.2, необходимо прибавить к левым частям неравенств базисные переменные:

$$\begin{cases} x_1 + 2x_2 + x_4 + x_5 = 180, \\ x_2 + 2x_3 + 2x_4 + x_6 = 210, \\ 4x_1 + 2x_2 + 4x_4 + x_7 = 800. \end{cases}$$

Приведём полученную систему уравнений к каноническому виду:

$$\begin{cases} 1\cdot x_1 + 2\cdot x_2 + 0\cdot x_3 + 1\cdot x_4 + 1\cdot x_5 + 0\cdot x_6 + 0\cdot x_7 = 180, \\ 0\cdot x_1 + 1\cdot x_2 + 3\cdot x_3 + 2\cdot x_4 + 0\cdot x_5 + 1\cdot x_6 + 0\cdot x_7 = 210, \\ 4\cdot x_1 + 2\cdot x_2 + 0\cdot x_3 + 4\cdot x_4 + 0\cdot x_5 + 0\cdot x_6 + 1\cdot x_7 = 800. \end{cases}$$

Коэффициенты левой части системы представляют собой матрицу норм расхода:

$$a = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 & 0 & 0 \\ 0 & 1 & 3 & 2 & 0 & 1 & 0 \\ 4 & 2 & 0 & 4 & 0 & 0 & 1 \end{pmatrix}.$$

3. Алгоритм решения задачи симплекс-методом

3.1. Симплекс-таблица

Симлекс-таблица, для данного варианта в общем виде выглядит так, как представлено в таблице 2.

Таблица 2.

БП	$C_Б$	A_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	Симплексные отношения
			c_1	c_2	c_3	c_4	c_5	c_6	c_7	
x_5	0	$b_1 = A_0[1]$	a_{11}	a_{12}	a_{13}	a_{14}	1	0	0	b_1/a_{1r}
x_6	0	$b_2 = A_0[2]$	a_{21}	a_{22}	a_{23}	a_{24}	0	1	0	b_2/a_{2r}
x_7	0	$b_3 = A_0[3]$	a_{31}	a_{32}	a_{33}	a_{43}	0	0	1	b_3/a_{3r}
Z_j - c_j		Δ_0	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	

Последняя строка симлекс-таблицы – индексная строка (она же строка оценок). Значения в ней рассчитываются по формулам: $\Delta_0 = C_Б \cdot A_0$, $\Delta j = C_Б \cdot Aj - c_j$.

На начальном этапе решения задачи симплекс-таблица для данного варианта представлена в таблице 3.

Таблица 3.

БП	$C_Б$	A_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	Симплексные отношения
			9	6	4	7	0	0	0	
x_5	0	180	1	0	2	1	1	0	0	180
x_6	0	210	0	1	3	2	0	1	0	-
x_7	0	800	4	2	0	4	0	0	1	200
Z_j - c_j		0	-9	-6	-4	-7	0	0	0	

Разрешающий столбец r выбирается по наибольшему по подулю отрицительному значению Δ_r. Разрешающая строка – по наименьшему симплексному отношению. Разрешающий элемент – элемент разрешающей строки и разрешающего столбца. При переходе к следующей итерации по правилу треугольника пересчитываются элементы матрицы a и вектора A_0.

3.2. Реализация решения задачи симплекс-методом на языке Python.

Для решения задачи симплекс-методом, был разработан класс SimplexTable, который содержит следующие поля и методы:

free_cnt – количество свободных переменных

basis_cnt – количество базисных переменных

free_coef – вектор свободных коэффициентов в симплекс-таблице (вектор c).

basis_coef – вектор базисных коэффициентов (вектор $C_Б$).

cond_table – матрица норм расхода a.

cond_coef – вектор ограничений b (при инициализации равен A_0).

delta_coef – индексная строка (вектор Δ).

Z – значение целевой функции (функции f).

simplex_div – вектор симплексных отношений.

basis_indexes – индексы переменных, входящих в список базисных.

iter_cnt – счётчик итераций.

__init__ – конструктор (инициализация полей при создании экземпляра класса).

Вначале вызывается метод init, который принимает на вход сперва вектор c в качестве вектора свободных коэффициентов, матрицу норм расхода a, а также вектор ограничений A_0. Количество базисных и свободных переменных, а также индексы переменных, входящих в состав список базисных, определяются исходя из размерности матрицы a. Номеру итерации присваивается нулевое значение.

```
def init(self, free_coef, cond_table, cond_coef):
    self.free_cnt = cond_table.shape[1] - cond_table.shape[0]
    self.basis_cnt = cond_table.shape[0]
    self.free_coef = free_coef
    self.basis_coef = np.array(free_coef[-self.basis_cnt:])
    self.cond_table = cond_table
    self.cond_coef = cond_coef
    self.delta_coef = np.zeros(self.free_cnt + self.basis_cnt)
    self.simplex_div = np.zeros(self.basis_cnt)
    self.basis_indexes = np.zeros(self.basis_cnt)
    for i in range(self.basis_cnt):
        self.simplex_div[i] = np.nan
        self.basis_indexes[i] = self.free_cnt + i
     self.iter_cnt = 0
```

Основной метод класса fit вначале вычисляет значение целевой функции (метод calc_z) на данной итерации, затем вычисляет строку оценок (метод calc_delta), после чего вычисляет разрешающий элемент (метод find_resolving_element), а также номер разрешающей строки и разрешающего столбца. После этого пересчитывается значение целевой функции. После чего метод iterate либо повторяет итерацию, либо выполнение метода fit завершается.

```
 def fit(self):
    self.calc_z()
    self.calc_delta()
    e, r, c = self.find_resolving_element()
    print("\x1b[31;1m" + str(self.iter_cnt) + "\x1b[0m")
    print(self)
    self.recalculate_table(e, r, c)
    self.iter_cnt = 1
    self.calc_z()
    while not self.iterate():
        pass
```

Метод iterate вначале проверяет условие выхода из цикла (метод check_delta), затем после вычисления разрешающего элемента пересчитывает строку оценок (recalculate_delta), затем матрицу норм расхода (recalculate_table), вычисляет значение целевой функции (calc_z) и если условие выхода из цикла выполняется, то печатает результат (print_res), затем наращивает счётчик итераций и возвращает флаг выхода из цикла.

```
def iterate(self):
    print("\x1b[31;1m" + str(self.iter_cnt) + "\x1b[0m")
    res = False
    if self.check_delta():
        res = True
    e, r, c = self.find_resolving_element()
    print(self)
    self.recalculate_delta(e, r, c)
    self.recalculate_table(e, r, c)
    self.calc_z()
    if res:
        self.print_res()
    self.iter_cnt += 1
    return res
```

Метод find_resolving_element находит разрешающий элемент как элемент матрицы cond_table, соответсвующий разрешающему столбцу r_col и разрешающей строке r_row. Разрешающий столбец вычисляется по минимуму массива delta_coef. После этого рассчитываются симплексные отношения simplex_div. Для предотвращения деления на 0 и отрицательных симплексных отношений соответствующим таким случаям элементам массива simplex_div присваивается значение nan. Среди остальных ищется минимум и по нему определяется разрешающая строка r_row.

```
def find_resolving_element(self):
    # индекс разрешающего столбца
    r_col = np.argmin(self.delta_coef)
    for i, ac in enumerate(self.cond_coef):
        if self.cond_table[i][r_col] == 0:
            self.simplex_div[i] = np.nan
```

```
                continue
            self.simplex_div[i] = ac / self.cond_table[i][r_col]
            if self.simplex_div[i] < 0:
                self.simplex_div[i] = np.nan
        r_row = np.nanargmin(self.simplex_div)

        return self.cond_table[r_row][r_col], r_row, r_col
```

Метод triangle_rule пересчитывает таблицу а по правилу треугольника. Сперва создаётся новая матрица new_table той же размерности, что и матрица cond_table. Разрешающая строка переносится в новую матрицу без изменений. Элемент, стоящий на месте разрешающего, принимает нулевое значение. Все остальные элементы рассчитываются по правилу треугольника. Затем матрице cond_table присваивается значение new_table.

```
def triangle_rule(self, res_elem, res_row, res_col):
    new_table = np.zeros(self.cond_table.shape)
    for i in range(self.cond_table.shape[0]):
        if i == res_row:
            new_table[i] = self.cond_table[i]
            continue
        for j in range(self.cond_table.shape[1]):
            if j == res_col:
                new_table[i][j] = 0
                continue
            new_table[i][j]        =        self.cond_table[i][j]         -
\(self.cond_table[res_row][j] * self.cond_table[i][res_col]) / res_elem
    self.cond_table = new_table
```

Метод recalculate_table пересчитывает всю оставшуюся часть симплекс-таблицы: заменяет базисную переменную, соответствующую разрешающей строке, на переменную, соответствующую разрешающему столбцу. Следом за этим заменяет соответствующие замененной переменной коэффициенты. Также пересчитывает разрешающую строку и коэффициенты ограничивающих условий.

```
def recalculate_table(self, res_elem, res_row, res_col):
        # заменяем базисную переменную
        self.basis_indexes[res_row] = res_col
        self.basis_coef[res_row] = self.free_coef[res_col]

        res_cond = self.cond_coef[res_row]

        # по правилу треугольника пересчитываем коэффициенты ограничивающих
условий
        for i in range(self.basis_cnt):
            if i == res_row:
                continue
            self.cond_coef[i]  -=  res_cond  *  self.cond_table[i][res_col]  /
res_elem

        self.triangle_rule(res_elem, res_row, res_col)
        # делим разрешающую строку на разрешающий элемент
        self.cond_coef[res_row] /= res_elem
        self.cond_table[res_row] /= res_elem
```

Метод recalculate_delta пересчитывает строку оценок, при этом элементу этой строки, который соответствует разрешающему столбцу, присваивается нулевое значение.

```
        def recalculate_delta(self, res_elem, res_row, res_col):
            res_delta = self.delta_coef[res_col]
            for i in range(self.free_cnt + self.basis_cnt):
                if i == res_col:
                    self.delta_coef[i] = 0
                    continue
                self.delta_coef[i] -= self.cond_table[res_row][i] * res_delta /
res_elem
```

Метод check_delta проверяет условия выхода из цикла: все элементы строки оценок неотрицательны.

```
        def check_delta(self):
            return np.all(self.delta_coef >= 0)
```

Метод print_res выводит на печать максимум целевой функции и вектор x, при котором он достигается.

```
   def print_res(self):
        res_str = "SOLVED: f = " + str(self.Z) + " at ("
        for i in range(self.free_cnt):
            if i >= self.basis_cnt:
                res_str += '0'
                continue
            if  len(self.cond_coef[np.argwhere(self.basis_indexes==i)])!=
0:
                res_str +=
str(self.cond_coef[np.argwhere(self.basis_indexes == i)][0][0])
            else:
                res_str += '0'
            res_str += ', '
        res_str += ")\n"
        print(res_str)
```

Метод __str__ выводит на печать симплекс-таблицу.

```
    def __str__(self):
        s = "\x1b[31;1m" + "B_i\tC_b\tA_j\t" + "\x1b[0m"
        for i in self.free_coef:
            s += "\x1b[32;1m" + str(i) + "\x1b[0m" + "\t"
        s += "\x1b[31;1m" + "Simplex\n" + "\x1b[0m"
        for i in range(self.cond_table.shape[0]):
            s += "\x1b[32;1m" + str(self.basis_indexes[i]) + "\t" +
str(self.basis_coef[i]) + "\x1b[0m" + "\t" +\
                str(self.cond_coef[i]) + "\t"
            for j in range(self.cond_table.shape[1]):
                s += str(self.cond_table[i][j]) + "\t"
            s += "\x1b[34;1m" + str(self.simplex_div[i]) + "\x1b[0m" +
"\n"
```

```
        s += "\x1b[31;1m" + "    \tZ_j\t" + "\x1b[0m" + "\x1b[34;1m" +
str(self.Z)
        for i in self.delta_coef:
            s += "\t" + str(i)
        s += "\x1b[0m" + "\n"
        return s
```

4. Результат работы программы

Программа решает поставленную задачу линейного программирования за 5 итераций. На рисунке 1 представлен вывод на печать последних итераций и решение задачи. По результатам программы была решена поставленная ЗЛП, в последней строке напечатан ответ задачи.

```
2
B_i C_b A_j 9.0 6.0 4.0 7.0 0.0 0.0 0.0 Simplex
0.0 9.0 180.0   1.0 0.0 2.0 1.0 1.0 0.0 0.0 nan
5.0 0.0 210.0   0.0 1.0 3.0 2.0 0.0 1.0 0.0 210.0
6.0 0.0 80.0    0.0 2.0 -8.0    0.0 -4.0    0.0 1.0 40.0
    Z_j 1620.0  0.0 -6.0    14.0    2.0 9.0 0.0 0.0

3
B_i C_b A_j 9.0 6.0 4.0 7.0 0.0 0.0 0.0 Simplex
0.0 9.0 180.0   1.0 0.0 2.0 1.0 1.0 0.0 0.0 90.0
5.0 0.0 170.0   0.0 0.0 7.0 2.0 2.0 1.0 -0.5    24.2857142857
1.0 6.0 40.0    0.0 1.0 -4.0    0.0 -2.0    0.0 0.5 nan
    Z_j 1860.0  0.0 0.0 -10.0   2.0 -3.0    0.0 3.0

4
B_i C_b A_j 9.0 6.0 4.0 7.0 0.0 0.0 0.0 Simplex
0.0 9.0 131.428571429   1.0 0.0 0.0 0.428571428571  0.428571428571  -0.285714285714 0.142857142857  306.666666667
2.0 4.0 24.2857142857   0.0 0.0 1.0 0.285714285714  0.285714285714  0.142857142857  -0.0714285714286    85.0
1.0 6.0 137.142857143   0.0 1.0 0.0 1.14285714286   -0.857142857143 0.571428571429  0.214285714286  nan
    Z_j 2102.85714286   0.0 0.0 0.0 4.85714285714   -0.142857142857 1.42857142857   2.28571428571

5
B_i C_b A_j 9.0 6.0 4.0 7.0 0.0 0.0 0.0 Simplex
0.0 9.0 95.0    1.0 0.0 -1.5    0.0 0.0 -0.5    0.25    95.0
4.0 0.0 85.0    0.0 0.0 3.5 1.0 1.0 0.5 -0.25   nan
1.0 6.0 210.0   0.0 1.0 3.0 2.0 0.0 1.0 0.0 nan
    Z_j 2115.0  0.0 0.0 0.5 5.0 0.0 1.5 2.25

B_i C_b A_j 9.0 6.0 4.0 7.0 0.0 0.0 0.0 Simplex
0.0 9.0 95.0    1.0 0.0 -1.5    0.0 0.0 -0.5    0.25    95.0
4.0 0.0 85.0    0.0 0.0 3.5 1.0 1.0 0.5 -0.25   nan
1.0 6.0 210.0   0.0 1.0 3.0 2.0 0.0 1.0 0.0 nan
    Z_j 2115.0  0.0 0.0 0.5 5.0 0.0 1.5 2.25

SOLVED: f = 2115.0 at (95.0, 210.0, 0, 0)
```

Рисунок 1 – Результат работы программы.

Заключение

В результате выполнения лабораторной работы был реализован симплекс-метод решения задачи линейного программирования на языке высокого уровня Python.

Для работы с векторами и матрицами была использована библиотека numpy. Методы библиотеки применялись для:

- инициализации массивов,
- работы с размерностью массивов,
- перебора элементов.

В ходе выполнения лабораторной работы были получены навыки решения задачи линейного программирования при помощи симплекс-таблиц.

Приложение. Листинг программы.

```
import numpy as np

# Problem No. 6

# начальные условия
c = np.array([9., 6., 4., 7., 0., 0., 0.])
A0 = np.array([180., 210., 800.])
a = np.array([
    [1., 0., 2., 1., 1., 0., 0.],
    [0., 1., 3., 2., 0., 1., 0.],
    [4., 2., 0., 4., 0., 0., 1.]
])


# флаг печати таблиц после каждой итерации
PRNT_DBG = True

class SimplexTable:
    # инициализация по умолчанию
    def __init__(self):
        self.free_cnt = 0
        self.basis_cnt = 0
        self.free_coef = np.array([])
        self.basis_coef = np.array([])
        self.cond_table = np.array([])
        self.cond_coef = np.array([])
        self.delta_coef = np.array([])
        self.Z = 0
        self.simplex_div = np.array([])
        self.basis_indexes = np.array([])

    # создание экземпляра класса
    def init(self, free_coef, cond_table, cond_coef):
```

```
    self.free_cnt = cond_table.shape[1] - cond_table.shape[0]
    self.basis_cnt = cond_table.shape[0]
    self.free_coef = free_coef
    self.basis_coef = np.array(free_coef[-self.basis_cnt:])
    self.cond_table = cond_table
    self.cond_coef = cond_coef
    self.delta_coef = np.zeros(self.free_cnt + self.basis_cnt)
    self.simplex_div = np.zeros(self.basis_cnt)
    self.basis_indexes = np.zeros(self.basis_cnt)
    for i in range(self.basis_cnt):
        self.simplex_div[i] = np.nan
        self.basis_indexes[i] = self.free_cnt + i

    self.iter_cnt = 0

# вычисление значения целевой функции
def calc_z(self):
    self.Z = 0
    for b, A in zip(self.basis_coef, self.cond_coef):
        self.Z += b * A

# вычисление элементов строки оценок
def calc_delta(self):
    for i, c in enumerate(self.free_coef):
        self.delta_coef[i] = self.Z - c

# нахождение разрешающего элемента и симплексных отношений
def find_resolving_element(self):
    # индекс разрешающего столбца
    r_col = np.argmin(self.delta_coef)
    for i, ac in enumerate(self.cond_coef):
        if self.cond_table[i][r_col] == 0:
            self.simplex_div[i] = np.nan
            continue
        self.simplex_div[i] = ac / self.cond_table[i][r_col]
        if self.simplex_div[i] < 0:
            self.simplex_div[i] = np.nan
    r_row = np.nanargmin(self.simplex_div)

    return self.cond_table[r_row][r_col], r_row, r_col

# пересчёт матрицы a по методу треугольника
def triangle_rule(self, res_elem, res_row, res_col):
    new_table = np.zeros(self.cond_table.shape)
    for i in range(self.cond_table.shape[0]):
        if i == res_row:
            new_table[i] = self.cond_table[i]
            continue
        for j in range(self.cond_table.shape[1]):
            if j == res_col:
                new_table[i][j] = 0
                continue
```

```
                new_table[i][j]      =      self.cond_table[i][j]      -
(self.cond_table[res_row][j] *\
                    self.cond_table[i][res_col]) / res_elem
        self.cond_table = new_table

    # пересчет симплекс таблицы
    def recalculate_table(self, res_elem, res_row, res_col):
        # заменяем базисную переменную
        self.basis_indexes[res_row] = res_col
        self.basis_coef[res_row] = self.free_coef[res_col]

        res_cond = self.cond_coef[res_row]

        #    по    правилу    треугольника    пересчитываем    коэффициенты
ограничивающих условий
        for i in range(self.basis_cnt):
            if i == res_row:
                continue
            self.cond_coef[i] -= res_cond * self.cond_table[i][res_col] /
res_elem

        self.triangle_rule(res_elem, res_row, res_col)
        # делим разрешающую строку на разрешающий элемент
        self.cond_coef[res_row] /= res_elem
        self.cond_table[res_row] /= res_elem

    # перерасчет строки оценок
    def recalculate_delta(self, res_elem, res_row, res_col):
        res_delta = self.delta_coef[res_col]
        for i in range(self.free_cnt + self.basis_cnt):
            if i == res_col:
                self.delta_coef[i] = 0
                continue
            self.delta_coef[i] -= self.cond_table[res_row][i] * res_delta
/ res_elem

    # проверка условия выхода: неотрицательность элементов строки оценок
    def check_delta(self):
        return np.all(self.delta_coef >= 0)

    # итерация
    def iterate(self):
        print("\x1b[31;1m" + str(self.iter_cnt) + "\x1b[0m")
        res = False
        if self.check_delta():
            res = True
        e, r, c = self.find_resolving_element()
        print(self)
        self.recalculate_delta(e, r, c)
        self.recalculate_table(e, r, c)
        self.calc_z()
        if res:
```

```
            self.print_res()
        self.iter_cnt += 1
        return res

    # главная функция, выполняющая максимизацию по методу симплекс таблицы
    def fit(self):
        self.calc_z()
        self.calc_delta()
        e, r, c = self.find_resolving_element()
        print("\x1b[31;1m" + str(self.iter_cnt) + "\x1b[0m")
        print(self)
        self.recalculate_table(e, r, c)
        self.iter_cnt = 1
        self.calc_z()
        while not self.iterate():
            pass

    # функция печати результата
    def print_res(self):
        res_str = "SOLVED: f = " + str(self.Z) + " at ("
        for i in range(self.free_cnt):
            if i >= self.basis_cnt:
                res_str += '0'
                continue
            if len(self.cond_coef[np.argwhere(self.basis_indexes == i)]) != 0:
                res_str += str(self.cond_coef[np.argwhere(self.basis_indexes == i)][0][0])
            else:
                res_str += '0'
            res_str += ', '
        res_str += ")\n"
        print(res_str)

    # перегрузка функции печати для класса
    def __str__(self):
        s = "\x1b[31;1m" + "B_i\tC_b\tA_j\t" + "\x1b[0m"
        for i in self.free_coef:
            s += "\x1b[32;1m" + str(i) + "\x1b[0m" + "\t"
        s += "\x1b[31;1m" + "Simplex\n" + "\x1b[0m"
        for i in range(self.cond_table.shape[0]):
            s += "\x1b[32;1m" + str(self.basis_indexes[i]) + "\t" + str(self.basis_coef[i]) + "\x1b[0m" + "\t" +\
                str(self.cond_coef[i]) + "\t"
            for j in range(self.cond_table.shape[1]):
                s += str(self.cond_table[i][j]) + "\t"
            s += "\x1b[34;1m" + str(self.simplex_div[i]) + "\x1b[0m" + "\n"
        s += "\x1b[31;1m" + "    \tZ_j\t" + "\x1b[0m" + "\x1b[34;1m" + str(self.Z)
        for i in self.delta_coef:
```

```
            s += "\t" + str(i)
        s += "\x1b[0m" + "\n"
        return s

st = SimplexTable()
st.init(c, a, A0)
st.fit()
```