Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа №2
«Метод отсекающих плоскостей»

Выполнил: ст. гр. 953503
Басенко К. А.
Проверил: Дугинов О. И.

Минск 2022

Постановка задачи

Пусть имеется задача целочисленного линейного программирования

$$\begin{aligned} & c'x -> \max \\ & Ax = b \\ & 0 \le x\ , \\ & x \text{ - целое} \end{aligned} \qquad (1)$$

где $c \in Z^n$, $x = (x_1, x_2, \ldots, x_n)^T \in Z^n$ — вектор переменных, $A \in R^{m*n}$, $rankA = m < n$. Требуется получить оптимальный план задачи или вернуть отсекающее ограничение.

Описание алгоритма метода

Шаг 1. Найти оптимальное решение задачи линейного программирования.

Шаг 2. Прекращаем решение задачи ЦЛП, если все переменные задачи ЛП целые. В противном случае переходим к шагу 3.

Шаг 3. Сформируем отсекающую плоскость (ограничение). Для этого выберем любую дробную переменную (это обязательно базисная переменная!) x^0_{i0}, $i_0 \in J^0_Б$. Здесь $J^0_Б$ – оптимальный базис текущей задачи ЛП. Положим $y' = e'_{i0}(A^0_Б)^{-1}$, $a_j = y'A_j$, $\beta = y'b$, f_j - дробная часть числа a_j, f - дробная часть числа β. Сформируем отсекающее ограничение $\sum_{j=1}^{n}[-f_j]x_j + x_* = -f, x_* \ge 0 -$ целое, исходя из ограничения $\sum_{j=1}^{n} a_j x_j = \beta$, построенного по правилам предыдущего пункта

Шаг 4. Добавляем отсекающее ограничение $\sum_{j=1}^{n}[-f_j]x_j + x_* = -f, x_* \ge 0 -$ целое и новую (целую!) переменную к задаче ЛП и получаем расширенную (новую) задачу ЛП. Переходим к шагу 1.

Результат работы
Тест 1

Задание:

$$7 \cdot x_1 + 10 \cdot x_2 \rightarrow \max$$
$$-x_1 + 3 \cdot x_2 + x_3 \leq 6$$
$$7x_1 + x_2 + x_4 \leq 35$$

$x_1, x_2, x_3, x_4 \in \mathbb{Z}$

Вывод программы:

```
0 0 0.9545454545454546 0.13636363636363638 -1 |  4.5
```

Код

```
import math
import numpy as np


def get_cb(Jb, c):
    Cb = np.zeros((len(Jb)))
    i = 0
    while i < len(Jb):
        Cb[i] = c[Jb[i] - 1]
        i = i + 1
    return Cb



def get_delta(A, u, c):
    delta = np.zeros((A.shape[1]))
    i = 0
    while i < A.shape[1]:
        Aj = np.array([A[:, i]]).transpose()
        delta[i] = np.dot(u, Aj) - c[i]
        i = i + 1
    return np.around(delta, decimals=6)



def get_negative_elements_indexes(A, delta):
    negative_elems = []
    i = 0
    while i < A.shape[1]:
        if delta[i] < 0:
            negative_elems.append(i+1)
        i = i + 1
    return negative_elems



def get_j0(negative_elems, delta, Jh):
    minimum = 1
    j0 = -1
    j = 0
    while j < len(negative_elems):
        if minimum > delta[negative_elems[j] - 1] and negative_elems[j]
in Jh:
            minimum = delta[negative_elems[j] - 1]
            j0 = negative_elems[j]
        j = j + 1
    return j0
```

```
def get_new_x(x, minimum, j0, z, Jb, Jh):
    new_x = np.copy(x)
    for i in Jh:
        new_x[i - 1] = 0
    new_x[j0 - 1] = minimum
    i = 0
    while i < len(Jb):
        new_x[Jb[i] - 1] = x[Jb[i] - 1] - minimum * z[i]
        i = i + 1
    return new_x



def get_new_Jb(Jb, s, j0):
    new_Jb = np.copy(Jb)
    new_Jb[s - 1] = j0
    return new_Jb



def get_new_Jh(new_Jb, count):
    new_Jh = []
    i = 0
    while i < count:
        if not (i + 1) in new_Jb:
            new_Jh.append(i + 1)
        i = i + 1
    return new_Jh



def get_new_Ab_and_B(A, Ab, B, s, j0):
    new_Ab = np.copy(Ab)
    new_Ab[:, s - 1] = A[:, j0 - 1]
    a = np.array([A[:, j0 - 1]]).transpose()
    l = np.dot(B, a)
    ls = float(l[s - 1])
    if ls == 0:
        return -2, -2 # matrix degenerate
    l[s - 1] = -1
    new_l = -1 / ls * l
    Q = np.eye((Ab.shape[0]))
    Q[:, s - 1] = new_l[:, 0]
    new_B = np.dot(Q, B)
    return new_Ab, new_B
```

```
def get_min_and_s(x, z, Jb):
    minimum = 1000
    s = -1
    i = 0
    while i < len(z):
        if z[i] > 0:
            if minimum > x[Jb[i] - 1] / z[i]:
                minimum = x[Jb[i] - 1] / z[i]
                s = i + 1
        i = i + 1
    return minimum, s



def new_iteration(A, b, c, Ab, B, Jb, Jh, x):
    Cb = get_cb(Jb, c)
    u = np.dot(Cb, B)
    delta = get_delta(A, u, c)
    negative_elems = get_negative_elements_indexes(A, delta)
    if len(negative_elems) == 0:
        return x, Jb # end of recursion
    j0 = get_j0(negative_elems, delta, Jh)
    z = np.dot(B, A[:, j0 - 1])
    positive = False
    for el in z:
        if el > 0:
            positive = True
    if not positive:
        print("STOP")
        print("задача не имеет решения в силу неограниченности сверху
целевой функции на множестве планов")
        return None, None
    minimum, s = get_min_and_s(x, z, Jb)
    new_x = get_new_x(x, minimum, j0, z, Jb, Jh)
    new_Jb = get_new_Jb(Jb, s, j0)
    new_Jh = get_new_Jh(new_Jb, A.shape[1])
    new_Ab, new_B = get_new_Ab_and_B(A, Ab, B, s, j0)
    if type(new_Ab) == type(-2):
        print("STOP")
        print("Матрица вырожденная")
        return None, None
    return new_iteration(A, b, c, new_Ab, new_B, new_Jb, new_Jh, new_x)
```

```
def simplex_method(A, b, c, x):
    Jb = []
    i = 1
    while i <= A.shape[1]:
        if x[i-1] != 0:
            Jb.append(i)
        i = i + 1
    Jh = []
    i = 1
    while i <= A.shape[1]:
        if x[i-1] == 0:
            Jh.append(i)
        i = i + 1
    Ab = np.zeros((len(Jb), len(Jb)))
    i = 0
    while i < len(Jb):
        j = 0
        while j < len(Jb):
            Ab[i][j] = A[i][Jb[j] - 1]
            j = j + 1
        i = i + 1
    B = np.linalg.inv(Ab)
    answer, Jb = new_iteration(A, b, c, Ab, B, Jb, Jh, x)
    if answer is None:
        print("Something went wrong")
        return None
    return answer, Jb



def get_float_index(a):
    for i in range(len(a)):
        if isinstance(a[i], float):
            return i
    return None



def cutoff_method(A, b, c):
    x = [0]*(len(c) - len(b))
    for i in b:
        x.append(i)


    x_plan, Jb = simplex_method(np.array(A),
                                np.array(b).reshape((2, 1)),
                                np.array(c),
                                np.array(x)
```

```
                                )


B = [i - 1 for i in Jb]
N = []
for i in range(len(x)):
    if i not in B:
        N.append(i)


Ab = []
for i in B:
    Ab.append(np.array(A).transpose()[i])


Ab = np.array(Ab).transpose().tolist()
Ab_1 = np.linalg.inv(Ab)


An = []
for i in N:
    An.append(np.array(A).transpose()[i])


An = np.array(An).transpose().tolist()


L = Ab_1.dot(An)
index = get_float_index(x_plan)
if index is None:
    return True, x_plan


k = B.index(index)
l = [i for i in L[k]]


result = [0]*(len(x))
for i, j in zip(N, l):
    result[i] = j - math.floor(j)


result.append(-1)
result.append(x_plan[index])
return False, result
```

```
if __name__ == "__main__":
    c = [7., 10., 0., 0.]
    A = [
        [-1, 3, 1, 0],
        [7, 1, 0, 1]
    ]
    b = [6., 35.]
    is_plan, result = cutoff_method(A, b, c)
    if not is_plan:
        for i in range(len(result) - 1):
            print(result[i], end=" ")
        i = i + 1
        print("| ", result[i])


    else:
        print("Plan: ", result)
```