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

(1)

,

*x - целое*

где с ∈ Zn, х = (х1, х2 , . . . , xn)T ∈ Zn — вектор переменных, A ∈ Rm\*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!) x0i0, i0 ∈ J0Б. Здесь J0Б – оптимальный базис текущей задачи ЛП. Положим y’ = e’i0(A0Б)-1, aj =y’Aj, =y’b , fj - дробная часть числа аj, f - дробная часть числа . Сформируем отсекающее ограничение , исходя из ограничения , построенного по правилам предыдущего пункта

Шаг 4. Добавляем отсекающее ограничение и новую (целую!) переменную к задаче ЛП и получаем расширенную (новую) задачу ЛП. Переходим к шагу 1.

**Результат работы**

**Тест 1**

Задание:

Вывод программы:



**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)