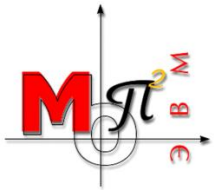
МИНОБРНАУКИ РОССИИ

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 № 3
по курсу «Практикум по ПМОиРД»

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

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

Оценка

_____

«____» _____ 2017 г.

Таганрог 2017

Оглавление

1. Постановка задачи

В соответствии с вариантом №6 ставится следующая задача:

Из Самары в Москву необходимо перевезти оборудование трех типов: b_1 = 1959 ед. типа A, b_2 = 146 ед. типа B и b_3 = 126 ед. типа C. Для перевозки оборудования завод может заказать два вида транспорта: T_1 и T_2. На единицу транспорта вида T_1 может быть погружено оборудования типа A не более a_{11} = 279 ед., оборудования типа B – не более a_{21} = 100 ед., оборудования типа C – не более a_{31} = 9 ед.; на единицу транспорта вида T_2 – не более a_{12} = 7, a_{22} = 21, a_{32} = 10 ед. оборудования соответственно типа A, B, C. Сменные затраты, связанные с эксплуатацией единицы транспорта вида T_1, составляют c_1 = 12 ден. ед., единицы транспорта вида T_2 – c_2 = 9 ден. ед.

	b_1	b_2	b_3	a_{11}	a_{12}	a_{31}	a_{21}	a_{22}	a_{32}	c_1	c_2
6	1953	147	126	279	7	9	100	21	10	12	9

Найти оптимальный план заказа транспорта для перевозки с минимальными затратами.

2. Математическая модель

Пусть x_1, x_2 – количество единиц транспорта T_1 и T_2 соответственно. x = (x1, x2).

n = 2 – количество различных видов транспорта.

m = 3 – количество различных типов оборудования.

$f = c_1 \cdot x_1 + c_2 \cdot x_2$– функция цели (затраты на перевозку).

В соответствии с вариантом: $f = 12x_1 + 9x_2$.

c_1, c2 – сменные затраты, связанные с эксплуатацией единиц транспорта T1 и T2 соответственно. $c = (c_1, c_2)$. В соответствии с вариантом: c = (12, 9)

b = (1953, 147, 126) – вектор ограничений (количество оборудования, которое необходимо перевезти).

$a = \begin{pmatrix} 279 & 7 \\ 100 & 21 \\ 9 & 10 \end{pmatrix}$ – матрица условий (в соответствии с вместимостью различного вида транспорта различными видами оборудования).

По условию задачи получаем следующие ограничения:

1. Количество единиц транспорта – целые неотрицательные числа:

$$x_j \geq 0, x_j \in Z, j = \overline{1,n}.$$

2. Из необходимости перевезти всё оборудование (b_1 ед. типа A, b_2 ед. типа B и b_3 ед. типа C.) следует:

$$\sum_{j=1}^{m} a_{ij} x_j \geq b_j, i = \overline{1,m}.$$

Цель задачи состоит в том, чтобы найти такой вектор $x=(x_1, x_2)$, при котором значение f минимально, то есть $f = c \cdot x \rightarrow \min$.

В соответствии с вариантом математическая модель задачи выглядит следующим образом:

$$f = 12x_1 + 9x_2 \to \min$$
$$\begin{cases} 279x_1 + 7x_2 \geq 1953, \\ -100x_1 + 21x_2 \geq 147, \\ 9x_1 + 10x_2 \geq 126 \\ x_1, x_2 \geq 0,\ x_1, x_2 \in Z \end{cases}$$

3. Алгоритм работы программы.

Программа, решающая поставленную задачу, написана на языке Python.

Для реализации метода ветвей и границ использовалась структура дерево, данные узлов которого представляют собой экземпляры класса Node_data. При инициализации экземпляра класса ему передаются в качестве параметров массивы A, b и c, которые инициализируют соответствующие поля класса. При создании нового узла дерева вызывается метод __init__:

```
        def __init__(self, A, b, c):
            self.n = A.shape[1]
            self.A = A
            self.b = b
            self.c = c
            self.symplex()
            if self.has_solution != True:
                self.is_leaf = True
                self.Z = None
            else:
                index, left = self.find_float()
                self.is_leaf = (index == -1)
                if self.is_leaf:
                    if self.Z < tb.extr_Z:
                        tb.extr_Z = self.Z
                        tb.extr_x = self.x
                        tb.clear()
```

В этом методе вызывается симплекс-метод, в результате которого экземпляры класса Node_data получают в виде полей информацию о том, существует ли решение, а также экстремальное значение целевой функции при имеющихся в данных узла ограничениях и вектор x, при котором это значение достигается. Если решения не существует, то узел помечается как листовой (поле is_leaf), а значению целевой функции присваивается None в знак того, что решение не существует. При этом полю x присваивается значение nan. Узел назначается листовым, когда необходимо остановить процесс дробления соответствующей ему ветки задачи.

В случае, если решение существует, при помощи метода find_float происходит поиск нецелой компоненты вектора x. Метод find_float вернёт -1 в качестве первого возвращаемого значения, если дробные компоненты не были найдены. В случае, если была найдена нецелая компонента, в качестве первого возвращаемого значения возвращается её индекс (присваивается переменной index), а в качестве второго – её целая часть (присваивается переменной left. В том случае, если все компоненты целые, то узел помечается как лист и в дальнейшем у него не будут создаваться потомки. При этом его значение поля Z

сравнивается с текущим экстремальным, и в случае, если оно меньше текущего экстремального, то переназначаются значения текущего экстремального значения целевой функции и вектора x, соответствующего этому значению. После этого вызывается метод clear дерева решений, который все узлы, значения Z которых больше нового экстремального помечает как листья, чтобы эти ветки решений дальше не рассматривались.

Метод `find_float` возвращает данные первой найденной нецелой компоненты:

```
def find_float(self):
    for i in range(self.n):
        left = math.trunc(self.x[i])
        if self.x[i] != left:
            return i, left
    return -1, None
```

Метод `symplex` вызывает метод `SimplexTable`, который возвращает объект, содержащий поля данных о существовании решения, экстремальное значение целевой функции и вектор x, соответствующий этому экстремальному значению. Полученные данные назначаются соответствующим полям узла.

```
def symplex(self):
    s = SimplexTable()
    symplexResult = s(self.c, self.A, self.b)
    self.x = symplexResult.x
    self.Z = symplexResult.fun
    self.has_solution = symplexResult.success
```

Класс узла Node_data также содержит метод split, который осуществляет дробление ветки задачи:

```
  def split(self, tree, parent, index, left):
        new_string = np.zeros(self.n)
        new_string[index] = 1
        new_A = np.append(self.A, [new_string], axis=0)
        new_b = np.append(self.b, left)
        left_tag = parent.tag + '1'
        left_identifier = parent.identifier + '1'
        left_node = Node_data(new_A, new_b, self.c)
        tree.create_node(tag=left_tag,               identifier=left_identifier,
parent=parent, data=left_node)
        new_string = np.zeros(self.n)
        new_string[index] = -1
        new_A = np.append(self.A, [new_string], axis=0)
        new_b = np.append(self.b, -left - 1)
        rigth_tag = parent.tag + '2'
        rigth_identifier = parent.identifier + '2'
        rigth_node = Node_data(new_A, new_b, self.c)
        tree.create_node(tag=rigth_tag,             identifier=rigth_identifier,
parent=parent, data=rigth_node)
        tb.tree.show(data_property='Z')
        tb.tree.show(data_property='x')
        print('extr_Z=', tb.extr_Z)
        if left_node.is_leaf != True:
            if ((parent.tag != '0') and (left_node.Z > parent.data.Z)):
                left_node.is_leaf = True
            else:
                index_float, left = left_node.find_float()
```

```
                if index_float != -1:
                    left_node.split(tree,        tree.get_node(left_identifier),
index_float, left)
        if rigth_node.is_leaf != True:
            if ((parent.tag != '0') and (rigth_node.Z > parent.data.Z)):
                rigth_node.is_leaf = True
            else:
                index_float, left = rigth_node.find_float()
                if index_float != -1:
                    rigth_node.split(tree,      tree.get_node(rigth_identifier),
index_float, left)
```

В этом методе создаются два потомка родительского узла, данные которых представляют собой экземпляры класса Node_data: left_node и right_node. Каждому из них при создании в качестве аргументов конструктора передаются новые вектора b и матрицы A, которые формируются на основе старых путём добавления нового ограничения: $x_i \le [x_i]$ для левого узла и $x_i \ge [x_i]+1$ для правого. При этом i и $[x_i]$ передаются в метод в качестве параметров index и left соответственно.

Затем на экран выводятся два дерева, характеризующих дерево решений задачи. Первое дерево содержит в вершинах значения целевой функции своих узлов. Второе – значение вектора x. После этого выводится на экран текущее экстремальное значение Z целевой функции.

После этого для левого и правого узлов также вызывается метод split, если значения Z узла меньше текущего экстремального и вектор x содержит нецелые компоненты. В противном случае узел считается листом и дальнейшего дробления соответствующей ему ветки решения не происходит.

Разработан класс TreeAndBorders, который имеет следующие поля и методы:

`A` – матрица условий,

`b` – вектор ограничений,

`c` – коэффициенты целевой функции,

`tree` – дерево решений,

`extr_Z` – экстремальное (минимальное) значение целевой функции,

`extr_x` – вектор x, в котором целевая функция принимает экстремальное значение.

Метод `go`, запускает выполнение алгоритма метода ветвей и границ:

```
    def go(self):
        self.tree.create_node('0',          '0',          parent=None,
data=Node_data(self.A, self.b, self.c))
        index_float, left = self.tree.get_node('0').data.find_float()
        if index_float != -1:
            self.tree.get_node('0').data.split(self.tree,
self.tree.get_node('0'), index_float, left)
        else:
            return self.x, self.Z
        self.tree.show(data_property='Z')
        self.tree.show(data_property='x')
```

В этом методе создаётся корневой узел дерева решений, в качестве данных узлу передаются начальные значения массивов A, b и c. После этого проверяется, содержит ли

вектор x корневого узла дробные компоненты. Если не содержит, то найденное решение считается удовлетворяющим условию. Если же нет, то создаётся два потомка корневого узла при помощи метода split, описанного выше.

4. Результат работы программы

Найденное симплекс-методом решение (6.83828383, 6.44554455) не оказалось целочисленным, поэтому от корневого узла создаются два потомка, разбиение произошло по первой компоненте x_1:

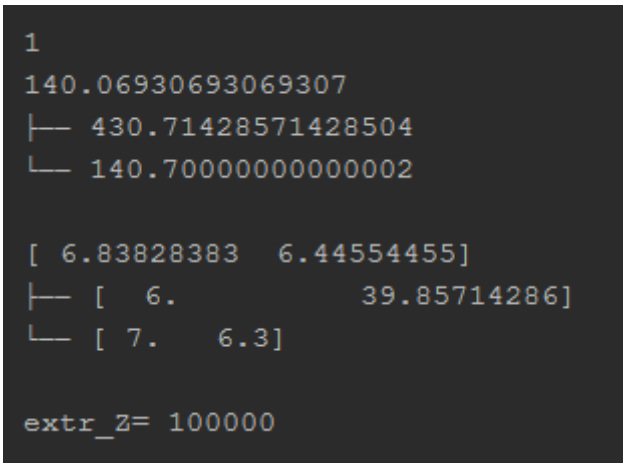


Рисунок 1. – Первая итерация.

На рисунке 1 показано состояние дерева решений на первой итерации: первое дерево в узлах содержит значения целевой функции, второе – вектор x, ему соответствующий. Внизу отображено текущее экстремальное значение. Так как ставится задача минимизации, при инициализации экстремальное значение было достаточно большому числу. Так как целочисленных решений ещё не найдено, оно не изменилось.

На второй итерации произошло разбиение левого потомка корневого дерева по компоненте x_2:

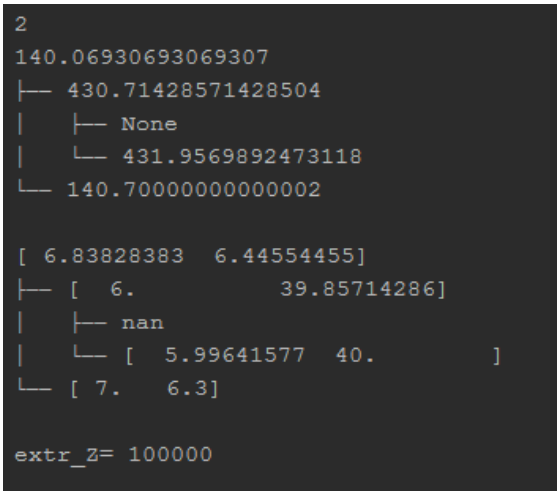


Рисунок 2. – Вторая итерация.

При этом задача минимизации для данных левого потомка решений не имеет, поэтмоу в узлах стоит None и nan.

На третьей итерации произошло разбиение правого потомка корневого узла по второй (единственной нецелой) компоненте x_2:

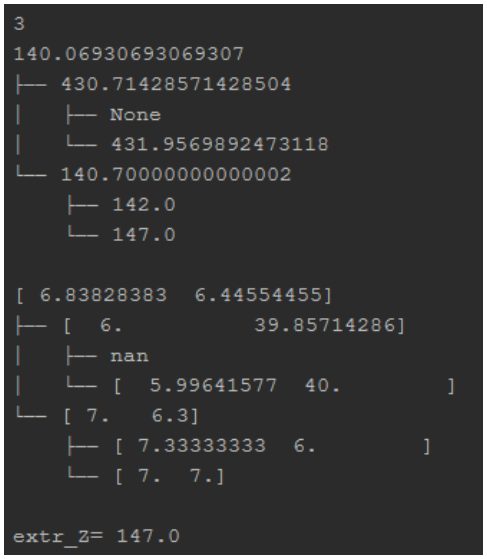


Рисунок 3. – Третья итерация.

При этом в правом потомке было получено первое целочисленное решение, поэтому текущее экстремальное значение функции изменилось. Левый потомок при этом не дал целочисленного решения, а значение целевой функции в нём меньше, чем в родителе, значит процесс ветвления этой подзадачи закончен. При этом в остальных узлах значения целевой функции больше, чем 147, а значит найдено оптимальное целочисленное решение. Результат выводится на экран:

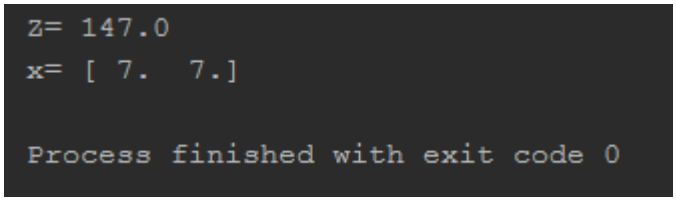


Рисунок 4. – Завершение работы программы.

5. Заключение

В результате выполнения лабораторной работы был реализован метод ветвей и границ задачи целочисленного 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,
- перебора элементов.

Для работы с векторами была использована библиотека treelib. Методы библиотеки применялись для создания дерева и его узлов.

В ходе выполнения лабораторной работы были получены навыки решения задачи целочисленного линейного программирования методом ветвей и границ.

Приложение. Листинг программы.

```
import numpy as np
from simplex import SimplexTable
from treelib import Node, Tree
import math
c = np.array([12.0, 9.0])
A = np.array([
    [-279.0, -7.0],
    [-100.0, -21.0],
    [-9.0, -10.0]
])
b = np.array([-1953.0, -147.0, -126.0])

class Node_data:

    def __init__(self, A, b, c):
        self.n = A.shape[1]
        self.A = A
        self.b = b
        self.c = c
        self.symplex()
        if self.has_solution != True:
            self.is_leaf = True
            self.Z = None
        else:
            index, left = self.find_float()
            self.is_leaf = (index == -1)
            if self.is_leaf:
                if self.Z < tb.extr_Z:
                    tb.extr_Z = self.Z
                    tb.extr_x = self.x
                    tb.clear()

    def find_float(self):
        for i in range(self.n):
            left = math.trunc(self.x[i])
            if self.x[i] != left:
                return i, left
        return -1, None

    def symplex(self):
        s = SimplexTable()
        symplexResult = s(self.c, self.A, self.b)
        self.x = symplexResult.x
        self.Z = symplexResult.fun
        self.has_solution = symplexResult.success


    def split(self, tree, parent, index, left):
        tb.counter +=1
        print()
        print(tb.counter)
        new_string = np.zeros(self.n)
        new_string[index] = 1
        new_A = np.append(self.A, [new_string], axis=0)
        new_b = np.append(self.b, left)
        left_tag = parent.tag + '1'
        left_identifier = parent.identifier + '1'
        left_node = Node_data(new_A, new_b, self.c)
```

```
        tree.create_node(tag=left_tag,                identifier=left_identifier,
parent=parent, data=left_node)
        new_string = np.zeros(self.n)
        new_string[index] = -1
        new_A = np.append(self.A, [new_string], axis=0)
        new_b = np.append(self.b, -left - 1)
        rigth_tag = parent.tag + '2'
        rigth_identifier = parent.identifier + '2'
        rigth_node = Node_data(new_A, new_b, self.c)
        tree.create_node(tag=rigth_tag,              identifier=rigth_identifier,
parent=parent, data=rigth_node)
        tb.tree.show(data_property='Z')
        tb.tree.show(data_property='x')
        tb.tree.show(data_property='is_leaf')
        print('extr_Z=', tb.extr_Z)
        if left_node.is_leaf != True:
            if ((parent.tag != '0') and (left_node.Z > parent.data.Z)):
                left_node.is_leaf = True
            else:
                print('left_node.is_leaf=', left_node.is_leaf)
                index_float, left = left_node.find_float()
                print('index_float=', index_float, ', Z=', left_node.Z)
                if index_float != -1:
                    left_node.split(tree,         tree.get_node(left_identifier),
index_float, left)
        if rigth_node.is_leaf != True:
            if ((parent.tag != '0') and (rigth_node.Z > parent.data.Z)):
                rigth_node.is_leaf = True
            else:
                index_float, left = rigth_node.find_float()
                if index_float != -1:
                    rigth_node.split(tree,       tree.get_node(rigth_identifier),
index_float, left)


class TreeAndBorders:

    def init(self, A, b, c):
        self.counter = 0;
        self.A = A
        self.b = b
        self.c = c
        self.tree = Tree()
        self.extr_Z = 100000
        self.extr_x = None

    def clear(self):
        nodes = self.tree.all_nodes_itr()
        for node in nodes:
            if node.data.is_leaf:
                continue
            if node.data.Z > tb.extr_Z:
                node.data.is_leaf = True

    def go(self):
        self.tree.create_node('0',   '0',   parent=None,   data=Node_data(self.A,
self.b, self.c))
        index_float, left = self.tree.get_node('0').data.find_float()
        if index_float != -1:
            self.tree.get_node('0').data.split(self.tree,
self.tree.get_node('0'), index_float, left)

tb = TreeAndBorders()
tb.init(A, b, c)
tb.go()
print('Z=', tb.extr_Z)
```

```
print('x=', tb.extr_x)
```