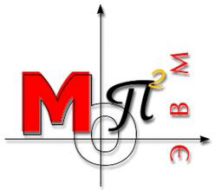
МИНОБРНАУКИ РОССИИ

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 № 2
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

Постановка задачи.

В соответствии с вариантом №6 ставится следующая задача:

На каждом из четырёх филиалов кондитерского объединения могут производиться конфеты четырёх видов «Алёнушка» (А), «Буратино» (Б), «Сласть» (С) и «Детские» (Д). Объём производства равен 120, 80, 160 и 90т в месяц соответственно. Себестоимость каждого из изделий на каждом из филиалов различна и определяется матрицей:

6	1	2	3
3	4	5	2
8	2	-	9
3	4	4	5

Учитывая, что спрос на конфеты А, Б, С, Д в месяц составляет 50, 150, 120 и 130 т, найдите такое распределение выпуска продукции между филиалами, при котором общая себестоимость продукции будет минимальной.

Математическая модель.

Данная задача является транспортной задачей (ТЗ) линейного программирования.

Согласно данным условия задачи имеем:

n=4, m=4,

$$C = (c_{ij}) = \begin{pmatrix} 6 & 1 & 2 & 3 \\ 3 & 4 & 5 & 2 \\ 8 & 2 & – & 9 \\ 3 & 4 & 4 & 5 \end{pmatrix},$$

(a_i) = (120, 80, 160, 90) – запасы,

(b_j)=(50, 150, 120, 130) – потребление.

Переменные ТЗ являются элементами следующей матрицы:

$$X = (x_{ij}) = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix}$$

Общий вид математического представления ТЗ:

$$\begin{cases} Z(X)=\sum_{i=1}^{m}\sum_{j=1}^{n}c_{ij}x_{ij}\to \min, & (1)\\ \sum_{j=1}^{n}x_{ij}=a_i, i=1,\dots,m, & (2)\\ \sum_{i=1}^{m}x_{ij}=b_j,\ j=1,\dots,n, & (3)\\ x_{ij}\ge 0, i=1,\dots,m,\ j=1,\dots,n. \end{cases}$$

Подставляя данные задачи согласно варианту имеем следующее:

целевая функция:

$$\begin{aligned} Z(X)=\ & 6\cdot x_{11}+1\cdot x_{12}+2\cdot x_{13}+3\cdot x_{14}+\\ & +3\cdot x_{21}+4\cdot x_{22}+5\cdot x_{23}+2\cdot x_{24}+\\ & +8\cdot x_{31}+2\cdot x_{32}+0\cdot x_{33}+9\cdot x_{34}+\\ & +3\cdot x_{41}+4\cdot x_{42}+4\cdot x_{43}+5\cdot x_{44} \end{aligned} \qquad (4)$$

из (2):

$$\begin{cases} x_{11}+x_{12}+x_{13}+x_{14}=120,\\ x_{21}+x_{22}+x_{23}+x_{24}=80,\\ x_{31}+x_{32}+x_{33}+x_{34}=160,\\ x_{41}+x_{42}+x_{43}+x_{44}=90, \end{cases} \qquad (5)$$

из (3):

$$\begin{cases} x_{11}+x_{21}+x_{31}+x_{41}=50,\\ x_{12}+x_{22}+x_{32}+x_{42}=150,\\ x_{13}+x_{23}+x_{33}+x_{43}=120,\\ x_{14}+x_{24}+x_{34}+x_{44}=130 \end{cases} \qquad (6)$$

Математическая постановка ТЗ состоит в поиске такого неотрицательного решения систем линейных уравнений (5) и (6), при котором целевая функция (4) достигает минимального значения.

Пусть $a=\sum_{i=1}^{m}a_i$, $b=\sum_{j=1}^{n}b_j$. Тогда a=120+80+160+90=450, b=50+150+120+130=450. Так как a=b, то данная задача является закрытой.

Итак, математическое представление поставленной ТЗ выглядит следующим образом:

$$\begin{cases} Z(X) = 6\cdot x_{11} + 1\cdot x_{12} + 2\cdot x_{13} + 3\cdot x_{14} + \\ \quad +3\cdot x_{21} + 4\cdot x_{22} + 5\cdot x_{23} + 2\cdot x_{24} + \\ +8\cdot x_{31} + 2\cdot x_{32} + 0\cdot x_{33} + 9\cdot x_{34} + \\ +3\cdot x_{41} + 4\cdot x_{42} + 4\cdot x_{43} + 5\cdot x_{44} \to min, \\ x_{11} + x_{12} + x_{13} + x_{14} = 120, \\ x_{21} + x_{22} + x_{23} + x_{24} = 80, \\ x_{31} + x_{32} + x_{33} + x_{34} = 160, \\ x_{41} + x_{42} + x_{43} + x_{44} = 90, \\ x_{11} + x_{21} + x_{31} + x_{41} = 50, \\ x_{12} + x_{22} + x_{32} + x_{42} = 150, \\ x_{13} + x_{23} + x_{33} + x_{43} = 120, \\ x_{14} + x_{24} + x_{34} + x_{44} = 130, \\ x_{ij} \ge 0, i = 1,...,4,\ j = 1,...,4. \end{cases}$$

Алгоритм работы программы.

Для решения поставленной задачи был разработан класс TransportProblemSolver (язык Python), который содержит следующие поля:

cost – матрица стоимости (c_{ij}).

post – матрица поставок (x_{ij}).

chars – матрица характеристик (r_{ij}).

volume – объёмы запасов: a_i (столбец справа от рабочего поля).

cons – потребление: b_j (строка снизу от рабочего поля).

rows, cols – количество строк и столбцов матрицы рабочего поля: n и m.

vp – потенциалы поставщиков (u_i).

cp – потенциалы потребителей (v_j).

Его метод fit выполняет поиск оптимального решения задачи:

```
def fit(self):
        self.check_type()
        self.initial_solution()
        print(self)
        if self.check_plan():
            cond = False
            while not cond:
                self.total_cost()
                self.potential_method()
                self.fill_chars()
                print(self)
                cond = self.check_optimal_criteria()
                if cond:
                    break
                self.recycle()
```

```
                print(self)
            else:
                pass
```

Вначале вызывается метод check_type, который определяет, является задача открытой или закрытой и в случае необходимости производит процедуру закрытия задачи:

```
    def check_type(self):
     v_sum = self.volume.sum()
     c_sum = self.cons.sum()
     if v_sum == c_sum:
           return True
     elif v_sum > c_sum:
           self.cons = np.append(self.cons, v_sum - c_sum)
           self.cost = np.append(self.cost, np.zeros((self.cost.shape[0], 1)),
axis=1)
           self.post = np.append(self.post, np.zeros((self.post.shape[0], 1)),
axis=1)
           self.chars = np.append(self.chars, np.zeros((self.chars.shape[0],
1)), axis=1)
     else:
           self.volume = np.append(self.volume, c_sum - v_sum)
           self.cost = np.append(self.cost, np.zeros((1, self.cost.shape[1])),
axis=0)
           self.post = np.append(self.post, np.zeros((1, self.post.shape[1])),
axis=0)
           self.chars = np.append(self.chars, np.zeros((1,
self.chars.shape[1])), axis=0)
     self.post.fill(np.nan)
     self.chars.fill(np.nan)
     self.rows = len(self.volume)
     self.cols = len(self.cons)
     self.vp = np.zeros(self.rows)
     self.cp = np.zeros(self.cols)
     return False
```

В случае открытой задачи метод возвращает True. В случае закрытой задачи добавляется фиктивный поставщик или фиктивный потребитель (в зависимости от того, что больше: суммарный груз всех поставщиков или суммарная потребность в грузе всех потребителей).

Далее вызывается метод initial_solution, внутри которого вычисляется опорное решение методами северо-западного угла и минимального элемента, сравниваются значения целевой функции, полученные этими методами, и в качестве опорного выбирается то, которому соответствует минимальное значение целевой функции.

Метод северо-западного угла.

Метод северозападного угла (метод NWC North West Corner) построения начального плана ТЗ заключается в том, что опорный план строится за m + n - 1 последовательных шагов, на каждом из которых заполняется только одна левая верхняя (северозападная) клетка незаполненного на текущий момент рабочего поля транспортной таблицы.

Заполнение любой такой клетки «*AiBj*» на пересечении *i* й стоки и *j* то столбца происходит по принципу *xij* = min { *ai*, *bj* }

В программе метод северо-западного угла реализуется в методе NWC:

```
def NWC(self):
    for i in range(self.rows):
        if self.volume[i] == 0:
            continue
        for j in range(self.cols):
            if self.cons[j] == 0:
                continue
            if self.cons[j] <= self.volume[i]:
                self.post[i][j] = self.cons[j]
                self.cons[j] = 0
                self.volume[i] -= self.post[i][j]
            else:
                self.post[i][j] = self.volume[i]
                self.volume[i] = 0
                self.cons[j] -= self.post[i][j]
            if self.volume[i] == 0:
                break
```

Здесь клетке «A_iB_j» (то есть x_{ij}) соответствует `self.post[i][j]`, условию min { a_i, b_j } – проверка условия `self.cons[j] <= self.volume[i]`.

Метод минимального элемента

Метод минимального элемента (метод МЭ, метод наименьших затрат или метод наименьшей стоимости) построения начального опорного плана ТЗ отличается от метода NWC правилом выбора клетки кандидата на заполнение. критерием выбора клетки «A_iB_j» для заполнения является минимум себестоимости перевозки c_{ij} минимальный элемент среди незаполненных клеток транспортной таблицы. Правило заполнения выбранной клетки в точности совпадает с аналогичным правилом метода NWC: обеспечить ее максимальную загрузку x_{ij} для удовлетворения текущих потребностей B_j из имеющихся запасов A_i.

Метод МЕ нахождения опорного решения методом минимального элемента:

```
def ME(self):
 while self.volume.sum() != 0 and self.cons.sum() != 0:
       min_ind = None
       for i in range(self.rows):
             if self.volume[i] == 0:
                   continue
             for j in range(self.cols):
                   if self.cons[j] == 0:
                         continue
                   if min_ind is None:
                         min_ind = (i, j)
                   if self.cost[min_ind] >= self.cost[i][j]:
                         min_ind = (i, j)

       min_val = np.minimum(self.volume[min_ind[0]], self.cons[min_ind[1]])
       self.post[min_ind] = min_val
       self.volume[min_ind[0]] -= min_val
```

```
        self.cons[min_ind[1]] -= min_val
```

Метод сперва находит строку и столбец (объект `(i, j)`), соответствующие минимальному элементу матрицы стоимости перевозки cost, записывает эти данные в переменную `min_ind`. После этого в переменную `min_val` записывается минимальное значение из двух значений: соответствующих `min_ind` элемента столбца запасов и строки потребностей. Затем из соответствующих элементов столбца запасов и строки потребностей вычитает значение этого минимального элемента, а соответствующей ячейке матрицы стоимости перевозки присваивается это значение. Всё это происходит до тех пор, пока не исчерпаются все запасы и не удовлетворятся все потребности.

После нахождения опорного плана программа печатает получившуюся таблицу. После этого происходит проверка плана на вырожденность: проверка того, что число базисных переменных в опорном плане равно m + n - 1. Реализует эту проверку метод check_plan:

```
def check_plan(self):
    a = self.rows * self.cols - len(np.where(np.isnan(self.post))[0])
    if a == self.cols + self.rows - 1:
        print("Plan is non-degenerate")
        return True
    else:
        print("Plan is degenerate!")
        return False
```

Метод потенциалов

Далее в методе fit следует цикл определения оптимального решения. В нём сначала считается значение целевой функции методом total_cost:

```
def total_cost(self):
    z = 0
    for i in range(self.rows):
        for j in range(self.cols):
            if not np.isnan(self.post)[i][j]:
                z += self.post[i][j] * self.cost[i][j]
    print("Z =", z)
    return z
```

Данный метод вычисляет z как $Z(X) = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$.

Далее вычисляются потенциалы запасов и потребления методом потенциалов – potential_method:

```
def potential_method(self):
    eq = np.zeros((self.rows + self.cols, self.rows + self.cols))
    vec = np.zeros(self.rows + self.cols)
    counter = 0
    for i in range(self.rows):
        for j in range(self.cols):
            if not np.isnan(self.post[i][j]) and self.post[i][j] != 0:
```

```
                    eq[counter][i] = 1
                    eq[counter][self.rows + j] = 1
                    vec[counter] = self.cost[i][j]
                    counter += 1
            ind = 0
            while counter < (self.rows + self.cols):
                eq[counter][ind] = 1
                counter += 1
            x = np.linalg.solve(eq, vec)
            for i in range(self.rows):
                self.vp[i] = x[i]
            for i in range(self.cols):
                self.cp[i] = x[i + self.rows]
```

Этот метод заполняет поля cv и cp – потенциалы поставщиков и потребителей.

После этого заполняется матрица характеристик, значения ячеек которой рассчитывается по формуле $r_{ij} = u_i + v_j - c_{ij}$ на основании вычисленных ранее потенциалов. Матрица характеристик заполняется в методе fill_chars:

```
def fill_chars(self):
  self.chars.fill(np.nan)
  for i in range(self.rows):
      for j in range(self.cols):
            if np.isnan(self.post[i][j]):
                 self.chars[i][j] = self.vp[i] + self.cp[j] - self.cost[i][j]
```

После все имеющиеся данные снова выводятся на печать.

После этого проверяется условие оптимальности решения: все характеристики должны быть неположительны. Данное условие проверяется в методе check_optimal_criteria:

```
        def check_optimal_criteria(self):
            for i in range(self.rows):
                for j in range(self.cols):
                    if not np.isnan(self.chars[i][j]):
                        if self.chars[i][j] > 0:
                            print("Non-optimal")
                            return False
            print("Optimal")
            return True
```

В случае если это условие выполняется, выполнение цикла вычисления оптимального решения завершается. В противном случае выполняется переход к новому опорному плану. Для этой цели строится цикл перерасчёта (замкнутая ломаная в рабочей области таблицы транспортной таблице, у которой начальная и конечная вершины располагаются в свободной клетке с наименьшей характеристикой, остальные вершины располагаются в занятых клетках, все углы прямые), и для каждой из вершин цикла происходит пересчёт значений поставок (прибавляется или отнимается минимальное значение поставки). Выполняется это в методе recycle:

```
def recycle(self):
      free_cell_ind = np.unravel_index(np.nanargmax(self.chars),
self.chars.shape)
      row_cand = []
```

```
      for j in range(self.rows):
            if j == free_cell_ind[0]:
                  continue
            if not np.isnan(self.post[j][free_cell_ind[1]]):
                  row_cand.append(j)
      nfree_cell_ind = None
      for j in row_cand:
            for i in range(self.cols):
                  if not np.isnan(self.post[j][i]):
                        if not np.isnan(self.post[free_cell_ind[0]][i]):
                              nfree_cell_ind = (j, i)
                              break
            if nfree_cell_ind is not None:
                  break
      cell_ind_list =[free_cell_ind, (free_cell_ind[0], nfree_cell_ind[1]),
                              nfree_cell_ind, (nfree_cell_ind[0],
free_cell_ind[1])]
      m_arr = np.array([self.post[cell_ind_list[1]],
self.post[cell_ind_list[2]], self.post[cell_ind_list[3]]])
      vm = np.nanmin(m_arr)
      plus_minus = [1, -1, 1, -1]
      for (i, j), p in zip(cell_ind_list, plus_minus):
            if np.isnan(self.post[i][j]):
                  self.post[i][j] = 0
            self.post[i][j] += p * vm
```

После этого происходит переход к новой итерации цикла определения оптимального решения.

После выхода из этого цикла матрица cost будет содержать значения перевозок, соответствующие оптимальному решению транспортной задачи.

Результат работы программы

Сначала находится опорный план методом северо-западного угла, транспортная таблица при этом выглядит так, как показано на рисунке 1.

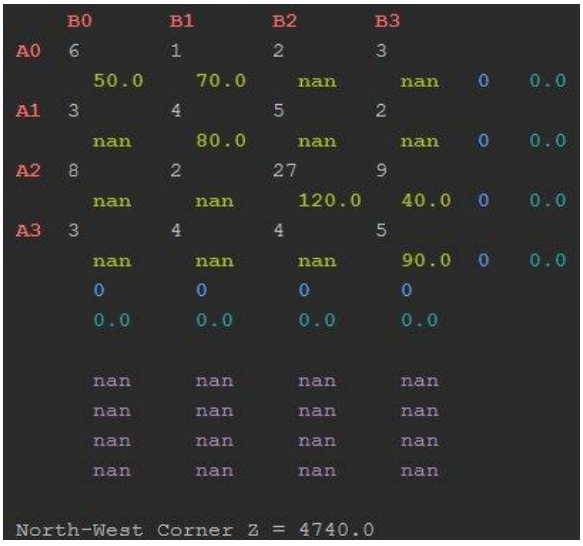


Рисунок 1 – Опорный план, полученный методом северо-западного угла.

Белым цветом показано значение матрицы себестоимости перевозок cost (c_{ij}), жёлтым – матрицы поставок post (x_{ij}), синий столбец справа – объёмы запасов volume (a_i), синяя строка внизу – объёмы потребностей cons (b_j), зелёный столбец справа – потенциалы объёмов запасов vp (u_i), зелёная строка внизу – потенциалы потребностей cp (v_j). Фиолетовая таблица внизу – матрица характеристик chars (r_{ij}).

Ячейки, содержащие nan – свободные клетки транспортной таблицы. В последней строке выводится вычисленное значение целевой функции: Z=4740.

Далее вычисляется опорное решение методом наименьшего элемента. Результат представлен на рисунке 2.

Рисунок 2 – Опорный план, полученный методом минимального элемента.

Значение целевой функции, полученное этим методом, равно 3260. Это меньше, чем значение, полученное методом северо-западного угла, поэтому в качестве опорного плана будет приниматься опорный план, полученный методом минимального элемента.

Далее происходит проверка плана на невырожденность и оптимальность (рис. 3).

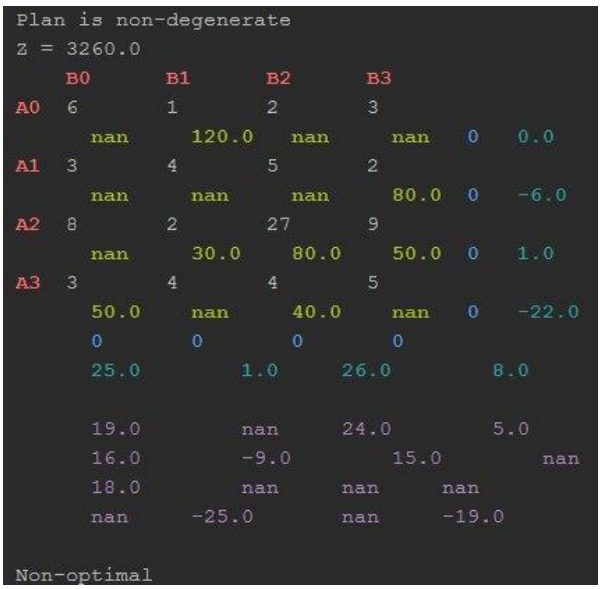


Рисунок 3 – Проверка оптимальности опорного плана.

В первой строке выводится текст «Plan is non-degenerate», что говорит о том, что полученный опорный план невырожден. Действительно, количество занятых ячеек транспортной таблице равно 7, что равно количеству строк (4) плюс количество столбцов (4) минус 1.

Также на этом этапе вычисляются значения потенциалов (зелёные столбец и строка) и характеристик (фиолетовая матрица внизу). Среди характеристик есть положительные числа, поэтому решение не является оптимальным, о чём говорит надпись в последней строке: «Non-optimal».

Далее переходим к новому опорному плану (рис. 4).

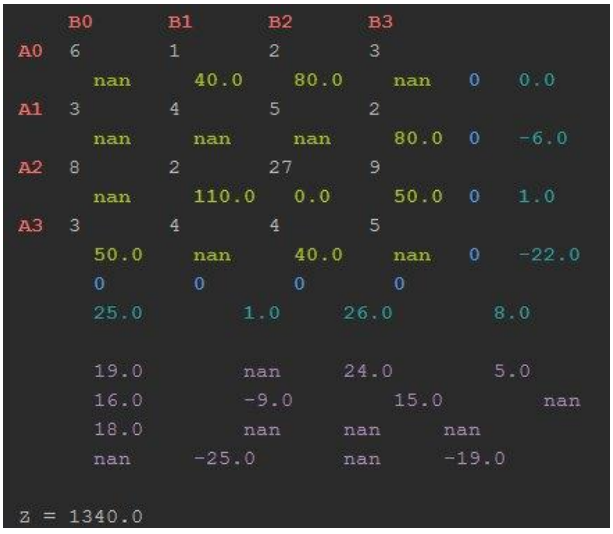


Рисунок 4 – Пересчёт рабочей области транспортной таблицы, первая итерация.

Получили новые значение матрицы cost и соответствующее ему новое значение целевой функции. Оно меньше, чем на предыдущем шаге.

Далее снова следует проверка на оптимальность (рис. 5).

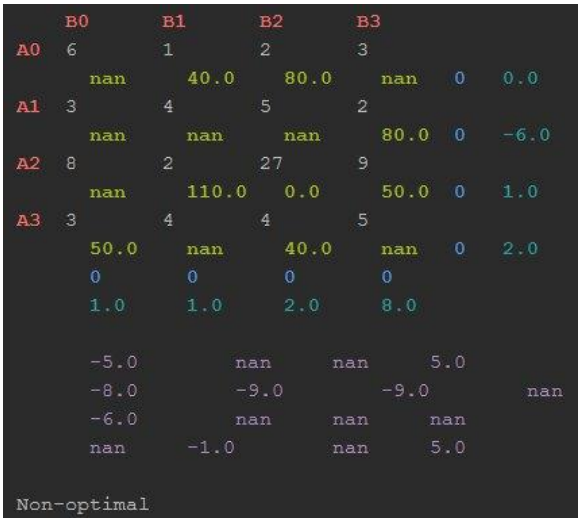


Рисунок 5 – Проверка оптимальности решения, первая итерация.

Снова полученное решение неоптимально, поэтому необходима следующая итерация. Происходит переход к новому опорному плану (рис. 6).

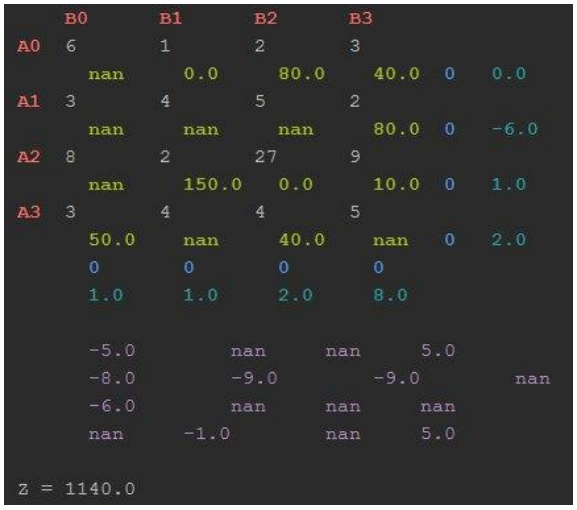


Рисунок 6 – Пересчёт рабочей области транспортной таблицы, вторая итерация.

Снова значение целевой функции меньше, чем на предыдущей итерации.

Далее снова следует проверка на оптимальность (рис. 7).

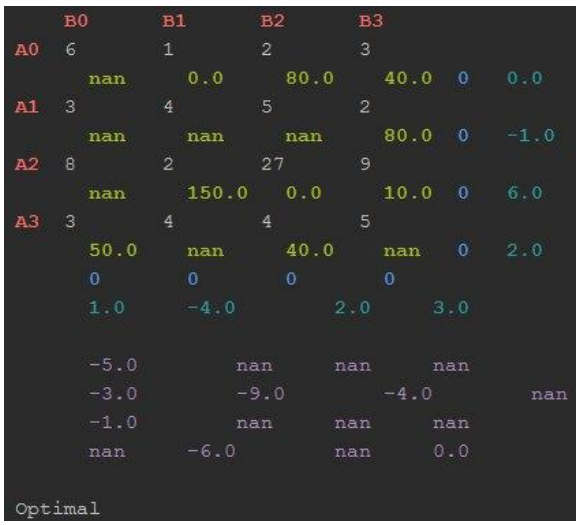


Рисунок 7 – Завершение работы программы.

Среди характеристик нет положительных, значит полученное решение оптимально. Об этом говорит надпись «Oprimal». На этом программа завершает своё выполнение. Соответствующее минимальное значение целевой функции: 1140. При этом

$$X = (x_{ij}) = \begin{pmatrix} 0 & 0 & 80 & 40 \\ 0 & 0 & 0 & 80 \\ 0 & 150 & 0 & 10 \\ 50 & 0 & 40 & 0 \end{pmatrix}.$$

Заключение.

В ходе выполнения лабораторной работы была разработана программа решения транспортной задачи на языке Python, которая:

- находит опорное решение методом северо-западного угла;
- находит опорное решение методом минимального элемента;
- в качестве опорного решения принимает минимальное из решений, полученных методами северо-западного угла и минимального элемента;
- вычисляет оптимальное решение методом потенциалов.

Для выполнения операций с векторами и матрицами (работа с размерностью, перебор элементов, поиск минимума и максимума) была использована библиотека nympy.

В ходе выполнения лабораторной работы были получены навыки решения транспортной задачи линейного программирования.

Листинг программы.

```
import numpy as np
from copy import deepcopy as dc

matrix = np.array([
    [6, 1, 2, 3],
    [3, 4, 5, 2],
    [8, 2, 27, 9],
    [3, 4, 4, 5]
])
volume = np.array([120, 80, 160, 90])
consumption = np.array([50, 150, 120, 130])

class TransportProblemSolver:
    def __init__(self, cost=None, volume=None, cons=None):
        self.cost = cost
        self.post = np.zeros(self.cost.shape)
        self.post.fill(np.nan)
        self.chars = np.zeros(self.cost.shape)
        self.chars.fill(np.nan)
        self.volume = volume
        self.cons = cons
        self.rows = len(self.volume)
        self.cols = len(self.cons)
        self.vp = np.zeros(self.rows)
        self.cp = np.zeros(self.cols)

    def check_type(self):
        v_sum = self.volume.sum()
        c_sum = self.cons.sum()
        if v_sum == c_sum:
            return True
        elif v_sum > c_sum:
            self.cons = np.append(self.cons, v_sum - c_sum)
            self.cost  =  np.append(self.cost,  np.zeros((self.cost.shape[0],
1)), axis=1)
            self.post  =  np.append(self.post,  np.zeros((self.post.shape[0],
1)), axis=1)
            self.chars                =                np.append(self.chars,
np.zeros((self.chars.shape[0], 1)), axis=1)
        else:
            self.volume = np.append(self.volume, c_sum - v_sum)
            self.cost       =       np.append(self.cost,       np.zeros((1,
self.cost.shape[1])), axis=0)
            self.post       =       np.append(self.post,       np.zeros((1,
self.post.shape[1])), axis=0)
            self.chars      =      np.append(self.chars,      np.zeros((1,
self.chars.shape[1])), axis=0)
        self.post.fill(np.nan)
        self.chars.fill(np.nan)
        self.rows = len(self.volume)
        self.cols = len(self.cons)
        self.vp = np.zeros(self.rows)
        self.cp = np.zeros(self.cols)
        return False

    def initial_solution(self):
        p = self.NWC(solve=False)
        temp_post = dc(self.post)
        temp_vol = dc(self.volume)
        temp_cons = dc(self.cons)
```

```
        self.post = dc(p)
        self.volume.fill(0)
        self.cons.fill(0)
        print(self)
        self.post = dc(temp_post)
        self.volume = dc(temp_vol)
        self.cons = dc(temp_cons)
        nwc_score = self.total_cost(post=p, prnt="North-West Corner")
        p1 = self.ME(solve=False)
        temp_post = dc(self.post)
        temp_vol = dc(self.volume)
        temp_cons = dc(self.cons)
        self.post = dc(p1)
        self.volume.fill(0)
        self.cons.fill(0)
        print(self)
        self.post = dc(temp_post)
        self.volume = dc(temp_vol)
        self.cons = dc(temp_cons)
        me_score = self.total_cost(post=p1, prnt="Minimal Element")
        if me_score <= nwc_score:
            self.ME()
        else:
            self.NWC()

    def check_plan(self):
        a = self.rows * self.cols - len(np.where(np.isnan(self.post))[0])
        if a == self.cols + self.rows - 1:
            print("Plan is non-degenerate")
            return True
        else:
            print("Plan is degenerate!")
            return False

    def total_cost(self, post=None, prnt=None):
        z = 0
        if post is None:
            for i in range(self.rows):
                for j in range(self.cols):
                    if not np.isnan(self.post)[i][j]:
                        z += self.post[i][j] * self.cost[i][j]
        else:
            for i in range(self.rows):
                for j in range(self.cols):
                    if not np.isnan(post)[i][j]:
                        z += post[i][j] * self.cost[i][j]
        if prnt is None:
            print("Z =", z)
        else:
            print(prnt, "Z =", z)
        return z

    def NWC(self, solve=True):
        post = dc(self.post)
        cons = dc(self.cons)
        volume = dc(self.volume)

        for i in range(self.rows):
            if volume[i] == 0:
                continue
            for j in range(self.cols):
                if np.isnan(self.cost[i][j]):
                    continue
```

```
                if cons[j] == 0:
                    continue
                if cons[j] <= volume[i]:
                    post[i][j] = cons[j]
                    cons[j] = 0
                    volume[i] -= post[i][j]
                else:
                    post[i][j] = volume[i]
                    volume[i] = 0
                    cons[j] -= post[i][j]
                if volume[i] == 0:
                    break
        if not solve:
            return post
        else:
            self.post = dc(post)
            self.cons = dc(cons)
            self.volume = dc(volume)

    def ME(self, solve=True):
        post = dc(self.post)
        cons = dc(self.cons)
        volume = dc(self.volume)

        while volume.sum() != 0 and cons.sum() != 0:
            min_ind = None
            for i in range(self.rows):
                if volume[i] == 0:
                    continue
                for j in range(self.cols):
                    if np.isnan(self.cost[i][j]):
                        continue
                    if cons[j] == 0:
                        continue
                    if min_ind is None:
                        min_ind = (i, j)
                    if self.cost[min_ind] >= self.cost[i][j]:
                        min_ind = (i, j)

            min_val = np.minimum(volume[min_ind[0]], cons[min_ind[1]])
            post[min_ind] = min_val
            volume[min_ind[0]] -= min_val
            cons[min_ind[1]] -= min_val
        if not solve:
            return post
        else:
            self.post = dc(post)
            self.cons = dc(cons)
            self.volume = dc(volume)

    def potential_method(self):
        eq = np.zeros((self.rows + self.cols, self.rows + self.cols))
        vec = np.zeros(self.rows + self.cols)
        counter = 0
        for i in range(self.rows):
            for j in range(self.cols):
                if not np.isnan(self.post[i][j]) and self.post[i][j] != 0:
                    eq[counter][i] = 1
                    eq[counter][self.rows + j] = 1
                    vec[counter] = self.cost[i][j]
                    counter += 1
        ind = 0
        while counter < (self.rows + self.cols):
```

```
                eq[counter][ind] = 1
                counter += 1
            x = np.linalg.solve(eq, vec)
            for i in range(self.rows):
                self.vp[i] = x[i]
            for i in range(self.cols):
                self.cp[i] = x[i + self.rows]

        def fill_chars(self):
            self.chars.fill(np.nan)
            for i in range(self.rows):
                for j in range(self.cols):
                    if np.isnan(self.post[i][j]):
                        self.chars[i][j]   =   self.vp[i]   +   self.cp[j]   -
self.cost[i][j]

        def check_optimal_criteria(self):
            for i in range(self.rows):
                for j in range(self.cols):
                    if not np.isnan(self.chars[i][j]):
                        if self.chars[i][j] > 0:
                            print("Non-optimal")
                            return False
            print("Optimal")
            return True

        def recycle(self, dbg=False):
            free_cell_ind      =      np.unravel_index(np.nanargmax(self.chars),
self.chars.shape)
            if dbg:
                print(free_cell_ind)
            row_cand = []
            for j in range(self.rows):
                if j == free_cell_ind[0]:
                    continue
                if not np.isnan(self.post[j][free_cell_ind[1]]):
                    row_cand.append(j)
            if dbg:
                print(row_cand)
            nfree_cell_ind = None
            for j in row_cand:
                for i in range(self.cols):
                    if not np.isnan(self.post[j][i]):
                        if not np.isnan(self.post[free_cell_ind[0]][i]):
                            nfree_cell_ind = (j, i)
                            break
                if nfree_cell_ind is not None:
                    break
            if dbg:
                print(nfree_cell_ind)
            cell_ind_list          =[free_cell_ind,          (free_cell_ind[0],
nfree_cell_ind[1]),
                             nfree_cell_ind,                (nfree_cell_ind[0],
free_cell_ind[1])]
            if dbg:
                print(cell_ind_list)
            m_arr           =           np.array([self.post[cell_ind_list[1]],
self.post[cell_ind_list[3]]])
            vm = np.nanmin(m_arr)
            if dbg:
                print(vm)
            plus_minus = [1, -1, 1, -1]
            for (i, j), p in zip(cell_ind_list, plus_minus):
```

```
            if np.isnan(self.post[i][j]):
                self.post[i][j] = 0
            self.post[i][j] += p * vm

    def fit(self):
        self.check_type()
        self.initial_solution()
        #print(self)
        if self.check_plan():
            cond = False
            while not cond:
                self.total_cost()
                self.potential_method()
                self.fill_chars()
                print(self)
                cond = self.check_optimal_criteria()
                if cond:
                    self.print_coef()
                    break
                self.recycle()
                print(self)
        else:
            pass

    def print_coef(self):
        st = "Target function coeffs:\n"
        for i in range(self.rows):
            for j in range(self.cols):
                if self.post[i][j] != 0 and not np.isnan(self.post[i][j]):
                    st += "x[" + str(i + 1) + "][" + str(j + 1) + "]\t"
        print(st)

    def __str__(self):
        def rstr(k):
            return str(np.round(k, decimals=2))

        s = "\x1b[31;1m" +"\t"
        for i in range(self.cols):
            s += "B" +rstr(i) + "  \t"
        s += "\t\x1b[0m" + "\n"
        for i in range(self.rows):
            s += "\x1b[31;1m" + "A" + rstr(i) + "\x1b[0m" + "\t"
            for j in self.cost[i]:
                s += rstr(j) + "   \t"
            s += "\n  \t"
            for j in self.post[i]:
                s += "\x1b[32;1m" + "  " + rstr(j) + "\x1b[0m" + "\t"
            s += "\x1b[34;1m" + rstr(self.volume[i]) + "\x1b[0m" + "\t"
            s += "\x1b[1;36;1m" + rstr(self.vp[i]) + "\x1b[0m" + "\n"
        s += "  \t" + "\x1b[34;1m"
        for i in self.cons:
            s += "  " + rstr(i) + "  \t"
        s += "\x1b[0m" + "\n  \t" + "\x1b[1;36;1m"
        for i in self.cp:
            s += "  " + rstr(i) + "  \t"
        s += "\x1b[0m" + "\n\n" + "\x1b[0;35m"
        for i in range(self.rows):
            s += "\t"
            for j in self.chars[i]:
                s += "  " +rstr(j) + "  \t"
            s += "\n"
        s += "\x1b[0m"
        return s
```

```
slvr = TransportProblemSolver(cost=matrix, volume=volume, cons=consumption)
slvr.fit()
```