МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ “ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ

УНИВЕРСИТЕТ ИМЕНИ П.М.МАШЕРОВА”

Факультет математики и информационных технологий

Кафедра геометрии и математического анализа

КУРСОВАЯ РАБОТА

По дисциплине «Методы оптимизации»

Задача о назначениях. Алгоритм Форда-Фалкерсона

Беляев М.С.,

3 курс, 31 группа

Руководитель:

Кавитова Т.В.

Старший преподаватель кафедры геометрии

и математического анализа

Витебск 2021

Оглавление

[Введение 2](#_Toc91787342)

[Математическая модель задачи 4](#_Toc91787343)

[Алгоритм решения задачи о назначениях 5](#_Toc91787344)

[Реализация 7](#_Toc91787345)

[Пример выполнения 18](#_Toc91787346)

[Ссылки и литература 22](#_Toc91787347)

# Введение

Целью данной курсовой работы является реализация алгоритма Форда-Фалкерсона для решения задачи о назначениях с использованием языков программирования.

Задача о назначениях – одна из фундаментальных задач комбинаторной оптимизации в области математической оптимизации или исследовании операций. Задача состоит в поиске минимальной суммы дуг во взвешенном двудольном графе.

В наиболее общей форме задача формулируется следующим образом:

Имеется некоторое число работ и некоторое число исполнителей. Любой исполнитель может быть назначен на выполнение любой (но только одной) работы, но с неодинаковыми затратами. Нужно распределить работы так, чтобы выполнить работы с минимальными затратами.

Если число работ и исполнителей совпадает, то задача называется линейной задачей о назначениях.

# Математическая модель задачи

Пусть имеется *n* работ и *n* исполнителей, каждый из которых может выполнить любую работу. Известны выплаты, которые следует сделать каждому исполнителю за выполнение любой имеющейся работы. Ставится задача распределения исполнителей по работам таким образом, чтобы каждая работа выполнялась только одним исполнителем, каждый исполнитель выполнял только одну работу, и выплаты при этом были минимальными.

Введем обозначения. Через *I = {1,2…n}* обозначим множество работ, через *J = {1,2…n}* – множество исполнителей, через *cij* - «стоимость» назначения на *i*-ую работу *j*-го исполнителя. Переменные в задаче определим следующим образом:

*xij =*

Ограничения на переменные определяются тем, что каждая работа должна быть выполнена одним исполнителем:

и каждый исполнитель выполняет только одну работу:

Совокупность переменных *х = (),* которая удовлетворяет вышеописанным условиям, назовём **назначением**. Очевидно, что суммарные выплаты, которые следует минимизировать, задаются функцией:

# Алгоритм решения задачи о назначениях

**Предварительный шаг** алгоритма состоит в приведении исходной матрицы, т.е. переход от матрицы С к матрице С2.

**Общий шаг**. 1) На основании матрицы С2 строим двудольный орграф *G=(S*, где *S={}* – множество вершин, которые соответствуют работам *T={}* – множество вершин, соответствующих исполнителям. Дуга (,, если соответствующий элемент матрицы С2 равен нулю.

2) По графу *G* строим расширенную сеть: вводим две дополнительные вершины *s* и *t* – источник и сток, и дополнительные дуги из источника *s* в вершины множества *S* и из вершин множества *Т* в сток *t*.

Пропускные способности всех дуг полагаем равными единице.

3) На полученной сети решаем задачу о нахождении максимального потока из источника *s* в сток *t*.

4) Если величина максимального потока *v = n*, то решение закончено.

Оптимальное назначение соответствует дугам вида (,для которых величина дугового потока равна единице. Оптимальность назначения следует из леммы 2, поскольку каждой такой дуге соответствует нулевой элемент матрицы С2.

5) Если величина максимального потока меньше *n*, то необходимо ввести дополнительные дуги вида (,чтобы увеличить мощность потока. Необходимую для этого информацию можем извлечь из последнего шага алгоритма построения максимального потока.

Пусть – множество вершин, помеченных на последнем шаге алгоритма (, – множество вершин, непомеченных на последнем шаге алгоритма (. Ясно, что надо ввести новую дугу из в . В матрице С2 рассмотрим элементы на пересечении строк, соответствующих вершинам из . Пусть *с* – минимальный среди них элемент. Вычитаем число *с* из всех элементов, соответствующих вершинам из , и прибавляем *с* ко всем элементам столбцов, соответствующих вершинам из

# Реализация

Алгоритм реализован с использованием средств языка программирования Python.

Исполняемый файл main.py является интерпретируемым. Файл graph.py содержит классы для работы с графом.



В процессе решения задачи о назначениях рассматривались сущности: вершина, дуга, граф. В файле graph.py описаны классы: Node, Edge, Graph.

Класс, описывающий вершину:

class Node:  
 name = ''  
 source = False  
 target = False  
 tagged = False  
 viewed = False  
 tag = []  
  
 neighbors = []  
  
 def \_\_init\_\_(self, name, source=False, target=False):  
 self.name = name  
 self.source = source  
 self.target = target  
  
 def \_\_repr\_\_(self):  
 return "%s" % (self.name)  
  
 def **set\_tag**(self,predecessor,capacity):  
 self.tag = [predecessor,capacity]  
 viewed = True  
 tagged = True  
 pass  
  
 def **get\_tag**(self):  
 return self.tag

Экземпляры класса *Node* имеют поля name, source, target, viewed, tag.

Поле name – хранит в себе название вершины. Поля логического типа, source, target, tagged, viewed, хранят информацию о состоянии вершины: является ли вершина источником, стоком, помеченной, просмотренной. Поле tag предназначено для хранения метки, присвоенной вершине, а neighbors хранит список смежных вершин для заданной вершины.

Методы \_\_init\_\_ и \_\_repr\_\_ выполняют функции представления вершины – создание и отображение соответственно. set\_tag и get\_tag позволяют установить и получить метку.

Класс *Edge* описывает дуги графа.

class Edge:  
 u = ''  
 v = ''  
 capacity = 0  
 flow = 0

direction = ''  
  
 def \_\_init\_\_(self, u, v, capacity,direction):  
  
 self.u = **Node**(u).name  
 self.v = **Node**(v).name  
 self.capacity = capacity  
 self.flow = 0  
 self.direction = '+'  
 def \_\_repr\_\_(self):  
 return "%s->%s" % (self.u, self.v)

Экземпляры классы имеют поля u и v, для хранения информации о направлении дуги. Capacity и flow хранят пропускную способность и дуги и поток, direction – прямая или обратная дуга.

Методы *\_\_init\_\_* и *\_\_repr\_\_* предназначены для создания и отображения дуги.

Класс Graph предназначен для представления сети.

class Graph:  
 def \_\_init\_\_(self):  
 self.nodes = []  
 self.network = {}  
  
 def **get\_source**(self):  
 for node in self.nodes:  
 if node.source == True:  
 return node  
 return None  
  
 def **set\_source**(self, new\_source):  
 for node in self.nodes:  
 if node.source == True:  
 node.source = False  
 for node in self.nodes:  
 if node.name == new\_source:  
 node.source = True  
 return node  
 return None  
  
 def **get\_target**(self):  
 for node in self.nodes:  
 if node.target == True:  
 return node  
 return None  
  
 def **set\_target**(self, new\_target):  
 for node in self.nodes:  
 if node.target == True:  
 node.target = False  
 for node in self.nodes:  
 if node.name == new\_target:  
 node.target = True  
 return node  
 return None  
  
 def **get\_node**(self, name):  
 for node in self.nodes:  
 if name == node.name:  
 return node

В экземпляре класса Graph поле nodes является списком всех вершин в сети, а network хранит всех список дуг.

Методы get\_source, set\_source, get\_target, set\_target позволяют задать и получить исток и сток в сети. Get\_node получает экземпляр класса Node по заданному названию вершины.

Методы add\_node – добавление вершины в сеть, get\_neighbors – получение списка смежных вершин для вершины с заданным названием, add\_edge – добавление дуги в сеть с заданным направлением и пропускной способностью, get\_edge – получение дуги с заданным направлением.

def **add\_node**(self, name, source=False, target=False):  
 if source == True and target == True:  
 return "Узел не может быть источником и стоком"  
 if self.**node\_in\_network**(name):  
 return "Узел уже существует"  
 if source == True:  
 if self.**get\_source**() != None:  
 return "Исток уже существует"  
 if target == True:  
 if self.**get\_target**() != None:  
 return "Сток уже существует"  
 new\_node = **Node**(name, source, target)  
 self.nodes.**append**(new\_node)  
 self.network[new\_node.name] = []  
  
 def **get\_neighbors**(self,node\_name):  
 neightbor\_list = []  
 node = self.**get\_node**(node\_name)  
 for edge in self.network[node.name]:  
 for node in self.nodes:  
 if edge.v == node.name:  
 neightbor\_list.**append**(node)  
 return neightbor\_list  
  
 def **add\_edge**(self, u, v, capacity):  
 if u == v:  
 return "Нельзя добавить одинаковые исток и сток"  
 if self.**node\_in\_network**(u) == False:  
 return "Вершина 'u' ещё не добавлена"  
 if self.**node\_in\_network**(v) == False:  
 return "Вершина 'v' ещё не добавлена"  
 new\_edge = **Edge**(u, v, capacity,'+')  
 node = self.**get\_node**(u)  
 self.network[node.name].**append**(new\_edge)  
  
 new\_edge = **Edge**(v, u, capacity, '-')  
 node = self.**get\_node**(v)  
 self.network[node.name].**append**(new\_edge)

def **get\_edge**(self, u,v):  
 for edge in self.network[u]:  
 if edge.u == u and edge.v == v:  
 return edge

Рассмотрим методы описывающие работу алгоритма.

Метод viewing\_node предназначен для просмотра вершины, он возвращает список смежных непомеченных вершин для заданной вершины.

def **viewing\_node**(self, source):  
 not\_tagged = []  
 print("**\n\t**Просматриваем вершину {0}:".**format**(source.name))  
 neighbor\_list = self.**get\_neightbors**(source.name)  
 for node in neighbor\_list:  
 if node.tagged == False:  
 not\_tagged.**append**(node)  
 edge = self.**get\_edge**(source.name, node.name)  
 print(node.name,'[{0}/{1}] – не помечена'.**format**(edge.capacity,edge.flow))  
 return not\_tagged

Tag\_a\_neighbor предназначен для присвоения метки смежной вершине для заданной вершины. Возвращает список всех помеченных вершин в сети.

def **tag\_a\_neighbor**(self, u, not\_tagged):tagged\_list = []

for v in not\_tagged:  
 edge = self.**get\_edge**(u.name, v.name)  
 if edge.flow != edge.capacity:  
 if edge.flow < edge.capacity:  
 v.tag = [u.name,min(u.tag[1], edge.capacity - edge.flow),'+']  
 v.tagged = True  
 if v not in tagged\_list:  
 tagged\_list.**append**(v)  
 print(v.name, '=', v.**get\_tag**())  
 elif edge.flow > 0:  
 v.tag = [u.name, min(u.tag[1], edge.flow), '-']  
 v.tagged = True  
 if v not in tagged\_list:  
 tagged\_list.**append**(v)  
 print(v.name, '=', v.**get\_tag**())  
   
 if v.name == 't':  
 return tagged\_list + ['t']  
 not\_tagged.**clear**()  
 return tagged\_list

Метод increasing\_chain возвращает список дуг (увеличивающую цель), от источника в сток.

def **increasing\_chain**(self, source, target):  
 v = **Node**(target.name)  
 u = **Node**('x')  
  
 chain = []  
 while v.name != source.name:  
 for node in self.nodes:  
 if node.name == target.name:  
 v = node  
 break  
  
 for node in self.nodes:  
 if node.name == target.tag[0]:  
 u = node  
 break  
 edge = self.**get\_edge**(u.name, v.name)  
  
 if v.tag[2] == '+':  
 edge.flow = v.tag[1]  
 chain.**append**(edge)  
  
 v = u  
 target = v  
  
 chain.**reverse**()  
 return chain

Метод ford\_fulkerson\_alg реализует при помощи выше описанных функций алгоритм нахождения максимального потока в сети.

def **ford\_fulkerson\_algorithm**(self):  
  
 i = 1  
 chain\_list = []  
  
 while True:  
 print('**\t\t\t**ИТЕРАЦИЯ', str(i)+':**\n**')  
 i += 1  
 tagged\_list = []  
 not\_tagged = []  
 auxiliary\_vertices = []  
 u = self.**get\_source**()  
 target = self.**get\_target**()  
  
 *# 0) Все вершины не помечены* for node in self.nodes:  
 node.tagged = False  
 node.tagged = False

*# 1) Вершина s получает метку ['∅', '∞']* u.tag = ['∅', math.inf]  
 u.tagged = True  
 tagged\_list.**append**(u)  
 print(self.**get\_source**() ,"-->",u.**get\_tag**())  
  
  
 route = []  
 while len(tagged\_list) > 0:  
 neighbor\_list = self.**get\_neightbors**(u.name)  
  
 *# 2) Просматриваем вершину:* not\_tagged = self.**viewing\_node**(u)  
  
 *# 3) Помечаем смежные вершины:* if len(not\_tagged) > 0:  
 print("**\n\t**Помечаем вершины для {0}".**format**(u.name))  
 tagged\_list += self.**tag\_a\_neighbor**(u,not\_tagged)  
  
 *# Если помечена вершина 3, переходим к шагу 2 (строим увеличивающую цель)* if 't' in tagged\_list:  
 print("**\n\n\t\t**ЭТАП 2")  
 tagged\_list.**remove**('t')  
 chain = self.**increasing\_chain**(self.**get\_source**(),  
tagged\_list[-1])  
 chain\_list.**append**(chain)  
 print("Увеличивающая цепь -> ",chain,'**\n\n**')  
 break  
  
 *# 4) Помечаем просмотренной вершину, переходим к другой* u.viewed = True  
 tagged\_list.**remove**(u)  
 auxiliary\_vertices.**append**(u)  
  
 *#5) Если вершина просмотрена и нет помеченных* if len(tagged\_list) == 0 and u.viewed == True:  
 print("**\n\t**Вершина {0} просмотрена**\n**".**format**(u))  
 max\_flow = 0  
 source = self.**get\_source**()  
  
 neighbor\_list = self.**get\_neightbors**(source.name)  
 for node in neighbor\_list:  
 edge = self.**get\_edge**(source.name, node.name)  
 max\_flow += edge.flow  
  
 return [max\_flow,auxiliary\_vertices,chain\_list]  
  
  
 print("**\n\t**Вершина {0} просмотрена**\n**".**format**(u))  
 u = tagged\_list[0]

Возвращает значение максимального потока в сети, список вершин, отмеченных на последней итерации, список увеличивающих цепей построенных в заданной сети.

Метод с3\_reduction преобразовывает матрицу С2 в матрицу С3

def **c3\_reduction**(self, C2, Cx, auxiliary\_vertices, n):  
  
 t\_untagged = []  
 s\_tagged = []  
 t\_tagged = []  
  
 for node in auxiliary\_vertices:  
 if 's' in node.name:  
 s\_tagged.**append**(node.name)  
 s\_tagged.**pop**(0)  
 print('Множество s помеченных ->', s\_tagged)  
  
 for node in self.nodes:  
 if 't' in node.name and (node not in auxiliary\_vertices):  
 t\_untagged.**append**(node.name)  
 t\_untagged.**pop**(0)  
 print('Множество t непомеченных ->', t\_untagged)  
  
 for node in self.nodes:  
 if 't' in node.name and (node.name not in t\_untagged):  
 t\_tagged.**append**(node.name)  
 t\_tagged.**pop**(0)  
  
 for i in range(len(t\_tagged)):  
 t\_tagged[i] = t\_tagged[i][1:]  
  
 for i in range(len(s\_tagged)):  
 s\_tagged[i] = s\_tagged[i][1:]  
  
 for i in range(len(t\_untagged)):  
 t\_untagged[i] = t\_untagged[i][1:]  
  
 temp\_list = []  
  
 for i in s\_tagged:  
 for j in t\_untagged:  
 Cx[int(i) - 1][int(j) - 1] = C2[int(i) - 1][int(j) - 1]  
 temp\_list.**append**(C2[int(i) - 1][int(j) - 1])  
  
 *#self.view\_array(Cx, 'Cx')* min\_value = min(temp\_list)  
 print('Минимальный элемент в матрице Сх ->', min\_value)  
  
 C3 = C2.**copy**()  
 for i in range(n):  
 for j in range(n):  
 if type(Cx[i][j]) == int:  
 C3[i][j] = C2[i][j] - min\_value  
  
 for i in t\_tagged:  
 for j in range(n):  
 if C3[j][int(i) - 1] != 0:  
 C3[j][int(i) - 1] += min\_value  
  
 return C3

Рассмотрим файл main.py.

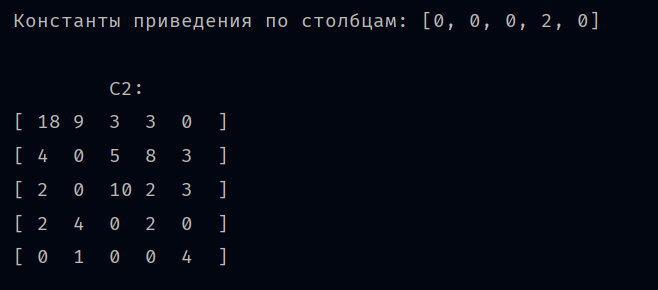
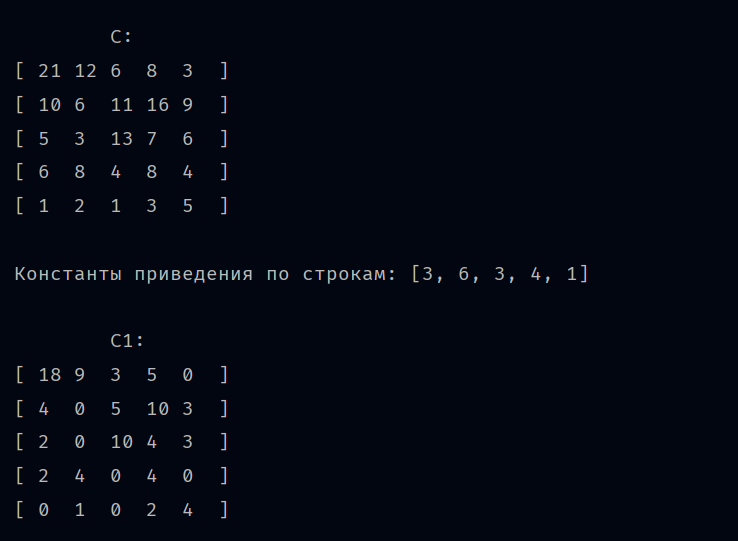
n = 5  
  
C = [ [21,12, 6, 8, 3],  
 [10, 6,11,16, 9],  
 [ 5, 3,13, 7, 6],  
 [ 6, 8, 4, 8, 4],  
 [ 1, 2, 1, 3, 5] ]  
C1 = list()  
C2 = list()  
Cx = list()  
min\_row\_elements = []

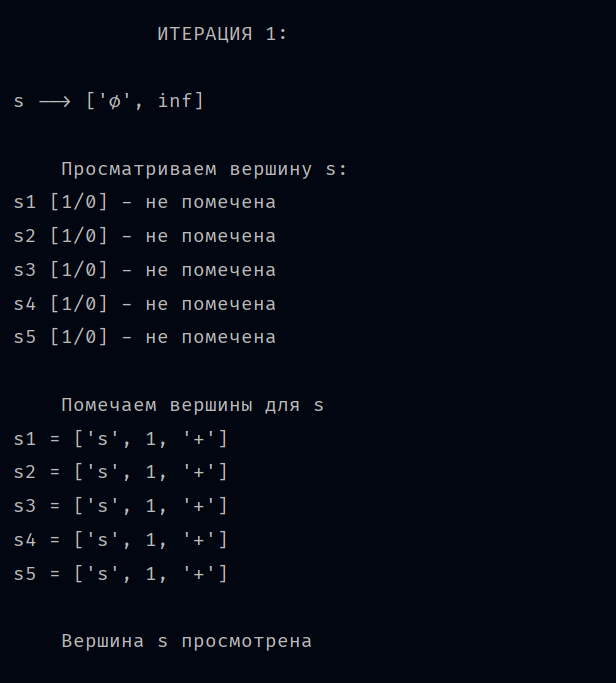
*# Заполнение матриц*for i in range(n):C1.**append**([0\*n]\*n)  
 C2.**append**([0\*n]\*n)  
 Cx.**append**(['.'] \* n)  
  
def **create\_graph\_x**(C):  
Gx = nx.**DiGraph**()  
  
 for i in range(n):  
 for j in range(n):  
 if C2[i][j] == 0:  
 Gx.**add\_edge**("s" + str(i + 1), "t" + str(j + 1), capacity=1.0)  
  
 for i in range(1, n + 1):  
 Gx.**add\_edge**("s", "s" + str(i), capacity=1.0)  
 Gx.**add\_edge**("t" + str(i), "t", capacity=1.0)  
 return Gx  
  
def **create\_graph**(C2):  
 G = m7.**Graph**()  
  
 G.**add\_node**('s')  
 G.**add\_node**('t')  
 G.**set\_source**('s')  
 G.**set\_target**('t')  
  
 for i in range(1, n + 1):  
 G.**add\_node**('s' + str(i))  
 G.**add\_node**('t' + str(i))  
  
 for i in range(1, n + 1):  
 G.**add\_edge**('s', 's' + str(i), 1)  
 G.**add\_edge**('t' + str(i), 't', 1)  
  
 for i in range(n):  
 for j in range(n):  
 if C2[i][j] == 0:  
 G.**add\_edge**('s' + str(i + 1), 't' + str(j + 1), 1)  
 return G

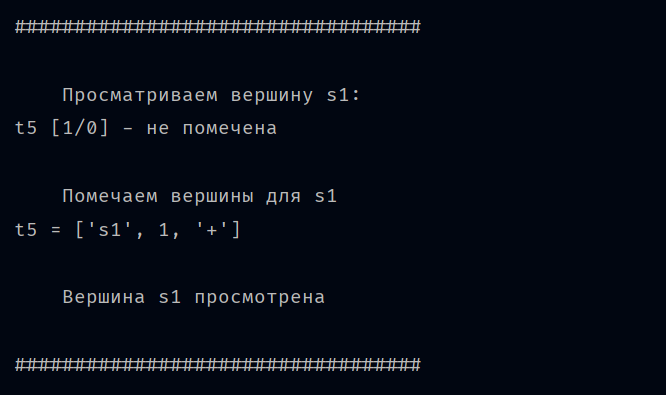
def **matrix\_reduction**(C,C1,C2):  
 *### Выполняем операцию приведения матрицы* **view\_array**(C, "C")  
  
 for i in range(len(C)):  
 min\_row\_elements.**append**(min(C[i]))  
 print("Константы приведения по строкам:", min\_row\_elements)  
  
 *# Заполнение С1* for i in range(n):  
 for j in range(n):  
 C1[i][j] = C[i][j] - min\_row\_elements[i]  
  
 **view\_array**(C1, "C1")  
  
 min\_col\_elements = [min(C1[i][j] for i in range(n)) for j in range(n)]  
 print("Константы приведения по столбцам:", min\_col\_elements)  
  
 *# Заполнение С2* for i in range(n):  
 for j in range(n):  
 C2[j][i] = C1[j][i] - min\_col\_elements[i]

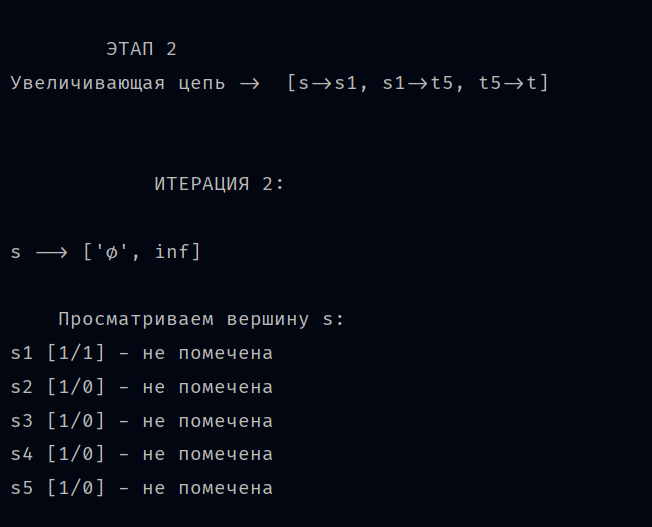
def **assignment\_task**():  
 performers = []  
  
 **matrix\_reduction**(C, C1, C2)  
  
 Gx = **create\_graph\_x**(C2)  
 G = **create\_graph**(C2)  
  
 result = G.**ford\_fulkerson\_algorithm**()  
  
 max\_flow = result[0]  
 auxiliary\_vertices = result[1]  
  
 print('---------- Максимальный поток =', max\_flow, '----------**\n**')  
  
 while max\_flow < n:  
 C3 = G.**c3\_reduction**(C2, Cx, auxiliary\_vertices, n)  
  
 **view\_array**(Cx, 'Cx')  
 **view\_array**(C3, 'C3')  
 **view\_graph**(Gx)  
  
 Gx = **create\_graph\_x**(C3)  
 G = **create\_graph**(C2)  
 result = G.**ford\_fulkerson\_algorithm**()  
 max\_flow = result[0]  
 print('---------- Максимальный поток =', max\_flow, '----------**\n**')  
 chain\_list = result[2]  
 for chain in chain\_list:  
 print(chain[1])  
 print('**\n**-----------------**\n**')  
 i = 1  
 for chain in chain\_list:  
 performers.**append**(str(chain[1])[5:6])  
 print('На {0}-ю работу назначен {1}-й исполнитель'.**format**(i,performers[i-1]))  
 i+=1

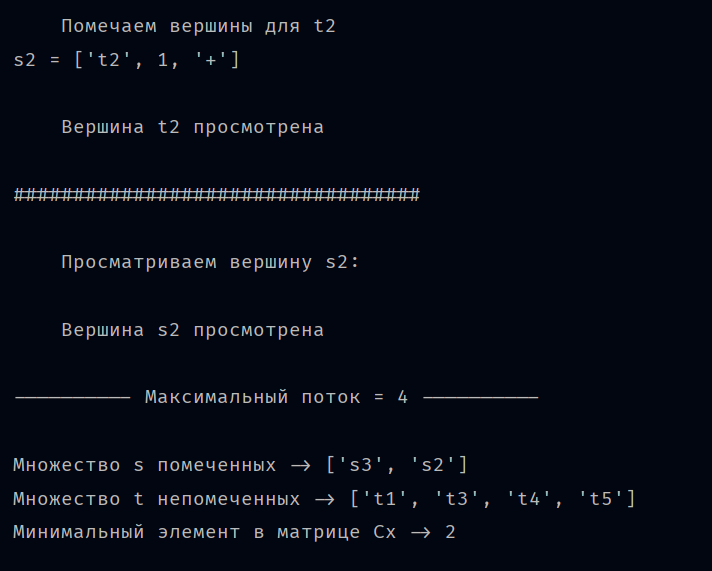
# Пример выполнения

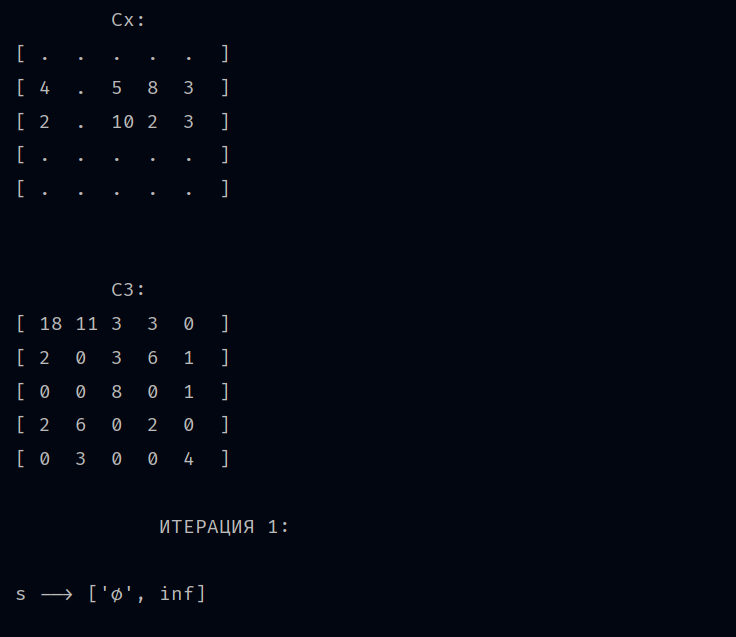




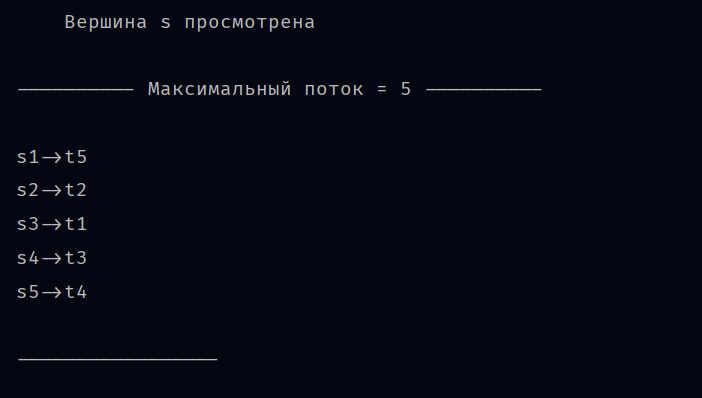


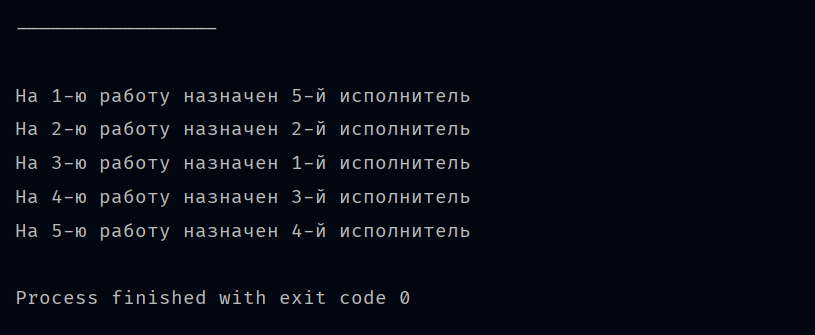






Перейдём к последней итерации.





# Ссылки и литература

1. Акоф, Р. Л. Основы исследования операций / Р. Л. Акоф, М. В. Сасиени. М., 1971.

2. Вагнер, Г. Основы исследования операций : в 3 т. / Г. Вагнер. М., 1972-1973. 3 т.

3. Вентцель, Е. С. Исследование операций / Е. С. Вентцель. М., 1972.

4. Краснопрошин, В. В. Исследование операций : учеб. пособие / В. В. Краснопрошин, Н. А. Лепешинский. − Минск : БГУ, 2013. − 191 с. – (Классическое университетское издание).

5. Командина, Л.В. Исследование операций. Задачи на графах и сетях, 2010.