**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**КАФЕДРА МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Построение и Анализ Алгоритмов»**

**Тема: Задача Коммивояжера**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1303 |  | Смирнов Д. Ю. |
| Преподаватель |  | Фирсов М. А. |

Санкт-Петербург

2023

**Цель работы.**

Изучение задачи Коммивояжера. Решение данной задачи методом ветвей и границ, с помощью алгоритма включения ближайшего города

**Задание.**

Решить ЗК двумя методами в соответствии с вариантом:

1) Методом ВиГ.

2) Приближённым методом.

Дано: матрица весов графа, все веса неотрицательны; стартовая вершина.

Найти: путь коммивояжёра (последовательность вершин) и его стоимость.

2. МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем вершинами; 2) веса МОД.  
Приближённый алгоритм: АВБГ.

Замечание к варианту 2: при оценке оставшегося пути учитывать, что  
оставшиеся вершины должны быть соединены не только между собой, но и с построенной цепочкой.

Вариант **2ц** - "ц" - должна быть решена замкнута ЗК (с циклом).

**Основные теоретические положения**

Метод ветвей и границ — общий алгоритмический метод для нахождения оптимальных решений различных задач оптимизации, особенно дискретной и комбинаторной оптимизации. Метод является развитием метода полного перебора, в отличие от последнего — с отсевом подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Алгоритм включения ближайшего города (АВБГ):

Если есть цепочка – – … – – , то следующим выбирается город , ближайший к этой цепочке, т.е. имеющий минимальную из стоимостей (для ), и этот город вставляется в текущий маршрут вслед за городом . Тогда цепочку будет иметь вид: – – …– – … – –

**Выполнение работы.**

Весь код программ представлен в приложении А.

**Описание алгоритмов.**

Метод ветвей и границ:

Первым делом находим нижнюю границу (полусумму весов двух легчайших рёбер по всем вершинами + вес МОД) и добавляем начальную вершину текущий путь, затем рекурсивно начинаем строить маршрут. Если стоимость текущего маршрута становится больше, чем нижняя граница, при этом не все вершины были добавлены в текущее решение, то решение отбрасывается, иначе, найденное решение сравнивается с рекордом, если оно лучше, то оно записывается как рекорд. Ветвление осуществляется следующим образом: в текущее решение добавляется вершина, которой ещё нет в текущем решении, после чего снова запускается рекурсия уже от этого решения, после того как выходим из рекурсии удаляется вершина и добавляется новая. Получается полный перебор дерева решений.

Алгоритм включения ближайшего города:

В текущий путь добавляем стартовую вершину. Затем запускаем поиск минимального ребра, исходящего из вершины, уже добавленной в путь, и приходящий в вершину, которую ещё не добавили. Если такое ребро нашли, то добавляем новую вершину в путь после вершины, из которой пришли в данную. Если при поиске очередной вершины оказывается, что минимальный вес ребра – бесконечность, значит пути нет, на этом моменте прекращаем поиск. Повторяем процедуру поиска до тех пор, пока в путь не будут добавлены все вершины, либо не будет доказано, что пути нет. В конце проверяем, что есть возможность путь зациклить (проверяем, что ребро из последней вершины пути в начальную имеет стоимость не бесконечность).

**Описание переменных и функций.**

Для представления графа (матрица весов) используется двумерный список *matrix*, где элемент a[i][j] – вес ребра из вершины *i* в *j*.

**Генерация, сохранение и чтение матриц**

* *readMatrix(filename)* – функция, считывает матрицу из файла. Аргументы функции: *filename* – название файла, из которого считывают матрицу. Функция возвращает двумерный список *matrix*.
* *dataReader()* ***–*** функция запрашивает и считывает все необходимые данные для генерации матрицы весов. Аргументов нет. Функция возвращает кортеж из количества вершин, минимального значения веса, максимального значения веса, и флаг симметричности.
* *generateMatrix()* – функция вызывает dataReader(), и относительно этих данные генерирует матрицу. Аргументов нет. Функция возвращает двумерный список *matrix*, сгенерированный согласно полученным данным.
* *saveMatrix(filename, matrix)* – функция записывает матрицу весов в файл. Аргументы функции: *filename* – название файла, в который будет идти запись, *matrix* – матрица весов. Функция ничего не возвращает.

**Описание классов**

***class methodBB***

Класс, отвечает за решение задачи коммивояжёра методом ветвей и границ.

Поля класса:

* *\_\_matrix* – двумерный список (матрица весов)
* *\_\_startVertex* – стартовая вершина
* *\_\_recordPath* – решение с минимальным весом
* *\_\_recordWeight* – вес минимального решения
* *\_\_DEBUG* – флаг, отвечающий за вывод дополнительной информации.
* *\_\_estimation* – значение нижней границы (описано в разделе “описание алгоритмов”)

Методы класса:

* *\_\_solve(self, currentPath, currentWeight)* – отвечает за сам алгоритм. Аргументы метода: *currentPath* – текущее решение, *currentWeight* – вес текущего решения. Метод ничего не возвращает.
* *\_\_strVertexArray(array)* – статический метод класса, выполняет приведение решения (массива чисел) к массиву строк. Аргументы метода: *array* – массив, который нужно привести. Функция возвращает массив вершин, где каждый элемент является строкой.
* *\_\_call\_\_(self)* – метод находит нижнюю границу и сохраняет её в \_\_estimation, после чего выполняет запускает решение. Метод не принимает аргументов. Метод возвращает найденное решение и вес этого решения.

***class AlgorithmIncludeNearestCity***

Класс, отвечает за решение задачи коммивояжёра методом Алгоритма включения ближайшего города (АВБГ).

Поля класса:

* *\_\_matrix* – двумерный список (матрица весов)
* *\_\_startVertex* – начальная вершина
* *\_\_path* – список из вершин текущего пути
* *\_\_DEBUG* - флаг, отвечающий за вывод дополнительной информации

Методы класса:

* *\_\_nextCity(self)* – метод осуществляет включения минимального ребра (описано в разделе “описание алгоритмов”). Аргументов не принимает. Метод ничего не возвращает.
* *\_\_strVertexArray(array)* – статический метод класса, выполняет приведение решения (массива чисел) к массиву строк. Аргументы метода: *array* – массив, который нужно привести. Функция возвращает массив вершин, где каждый элемент является строкой.
* *\_\_findPathCost(self)* – метод находит стоимость пути *\_\_path*. Аргументов не принимает. Возвращает стоимость пути.
* *\_\_solve(self)* – метод запускает АВБГ (описано в разделе “описание алгоритмов”). Метод ничего не принимает. Метод ничего не возвращает.
* *\_\_call\_\_(self)* – вызывает метод *\_\_solve(self)*. Аргументов не принимает. Возвращает цепочку вершин (решение задачи коммивояжёра), и вес этого пути.

**Функции двух нижних оценок (для МВиГ)**

* *minWeightEdges(matrix)* – функция находит полусумму двух легчайших ребер в графе. Аргументы функции: *matrix* – двумерный список (матрица весов графа). Функция возвращает найденное значение.
* *primFindMST(matrix, startVertex=0)* – функция строит минимальное остовное дерево алгоритмом Прима, из заданной вершины. Аргументы функции: *matrix* – двумерный список (матрица весов графа), *startVertex* – вершина, из которой будет строится дерево. Функция возвращает кортеж из минимального остовного дерева и его веса.

**Оценка сложности алгоритмов.**

Метод Ветвей и Границ:

По операциям в среднем (при «случайных» матрицах стоимостей) – О(), где n – кол-во вершин, а

По памяти – О(, где n – количество вершин

Алгоритм Включения Ближайшего Города:

По операциям – O(), где n – количество вершин.

По памяти – О(, где n – количество вершин

**Тестирование.**

Таблица 1 – тестирование метода ВиГ.

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
| 1 | 3  inf 27 43 16 30 26 7 inf 16 1 30 25 20 13 inf 35 5 0 21 16 25 inf 18 18 12 46 27 48 inf 5 23 5 5 9 5 inf | Цепочка:  3-5-6-2-4-1-3  Стоимость её прохождения:  80 |
| 2 | 1  inf 10 11 4 4 8 inf 6 6 6 6 10 inf 4 6 9 4 6 inf 6 11 4 5 10 inf | Цепочка:  1-4-2-5-3-1  Стоимость её прохождения:  25 |
| 3 | 1  inf 9 9 11 9 12 8 9 inf 7 4 4 1 11 9 7 inf 11 4 11 2 11 4 11 inf 3 1 11 9 4 4 3 inf 10 11 12 1 11 1 10 inf 10 8 11 2 11 11 10 inf | Цепочка:  1-2-6-4-5-3-7-1  Стоимость её прохождения:  28 |

Таблица 2 – тестирование АВБГ.

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
| 1 | 3  inf 27 43 16 30 26 7 inf 16 1 30 25 20 13 inf 35 5 0 21 16 25 inf 18 18 12 46 27 48 inf 5 23 5 5 9 5 inf | Цепочка:  3-6-2-1-4-5-3  Стоимость её прохождения:  73 |
| 2 | 1  inf 10 11 4 4 8 inf 6 6 6 6 10 inf 4 6 9 4 6 inf 6 11 4 5 10 inf | Цепочка:  1-4-5-3-2-1  Стоимость её прохождения:  33 |
| 3 | 1  inf 9 9 11 9 12 8 9 inf 7 4 4 1 11 9 7 inf 11 4 11 2 11 4 11 inf 3 1 11 9 4 4 3 inf 10 11 12 1 11 1 10 inf 10 8 11 2 11 11 10 inf | Цепочка:  1-7-3-5-4-6-2-1  Стоимость её прохождения:  28 |

Пример вывода дополнительной информации для алгоритмов на примере теста 1, представлен на рисунках 1-2 соответственно.

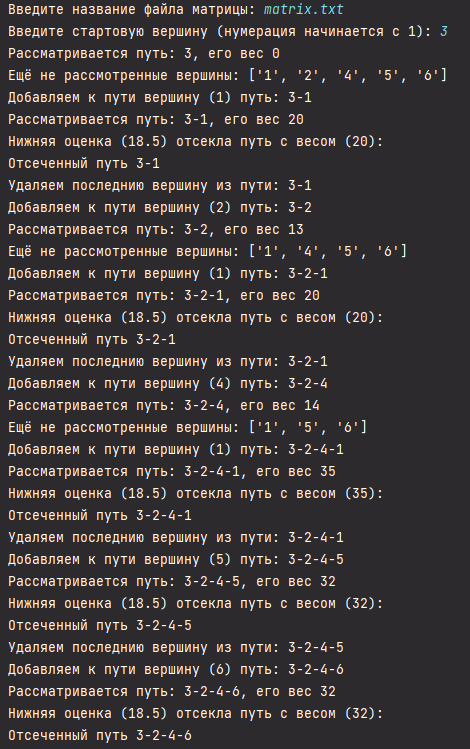
****

Рисунок 1 - Отрывок вывода дополнительной информации на первом тесте для МВиГ

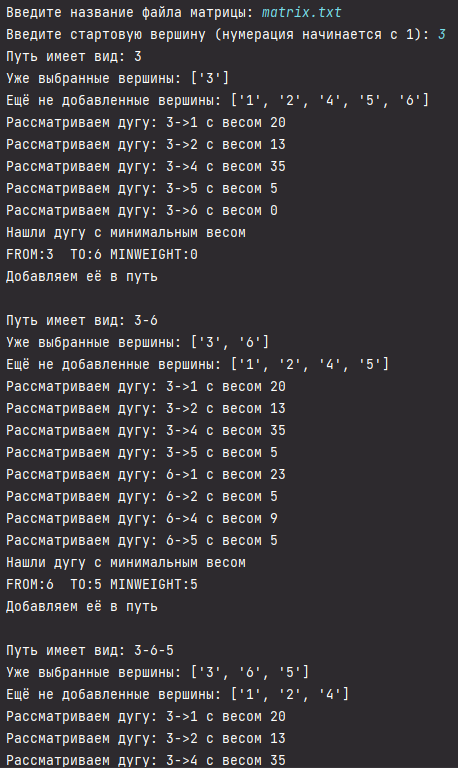


Рисунок 2 – Отрывок вывода дополнительной информации на первом тесте для АВБГ

**Вывод.**

В ходе лабораторной работы была изучена задача Коммивояжера, а также два метода ее решения: метод ветвей и границ и алгоритм включения ближайшего города, реализованы соответствующие алгоритмы**.**

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММ**

Название файла: algrorithmIncludeNearestCity.py

import math

from readFuncs import readMatrix

class AlgorithmIncludeNearestCity:

def \_\_init\_\_(self, matrix, startVertex, DEBUG=False):

# инициализация полей класса

self.\_\_matrix = matrix

self.\_\_startVertex = startVertex

self.\_\_DEBUG = DEBUG

self.\_\_path = [self.\_\_startVertex]

def \_\_nextCity(self):

"""

Функция ищет минимальное ребро в ещё не добавленные город

:return: ничего не возвразает

"""

# создаем список ещё не пройденных городов

notIncluded = [i for i in range(len(self.\_\_matrix)) if i not in self.\_\_path]

# заводим начальные значения

minWeight = math.inf

indexFrom = 0

indexTo = 0

if self.\_\_DEBUG:

print(f"Путь имеет вид: {'-'.join(self.\_\_strVertexArray(self.\_\_path))}")

print(f"Уже выбранные вершины: {self.\_\_strVertexArray(self.\_\_path)}")

print(f"Ещё не добавленные вершины: {self.\_\_strVertexArray(notIncluded)}")

# рассматрицаем дуги из уже просмотренных вершин в ещё не просмотренные

for vertex in self.\_\_path:

for nextVertex in notIncluded:

if self.\_\_DEBUG:

print(

f"Рассматриваем дугу: {vertex + 1}->{nextVertex + 1} с весом {self.\_\_matrix[vertex][nextVertex]}")

if self.\_\_matrix[vertex][nextVertex] <= minWeight:

# если нашли ребро с меньшим весов, чем текущее

# то обнавляем значения

indexFrom = vertex

indexTo = nextVertex

minWeight = self.\_\_matrix[vertex][nextVertex]

if minWeight != math.inf:

# добавляем к решение, если ребро в графе есть

if self.\_\_DEBUG:

print("Нашли дугу с минимальным весом")

print(f"FROM:{indexFrom + 1} TO:{indexTo + 1} MINWEIGHT:{minWeight}")

print("Добавляем её в путь\n")

self.\_\_path.insert(self.\_\_path.index(indexFrom) + 1, indexTo)

else:

# если нет, то граф состоит из нескольких не связаных частей

raise RuntimeError("Граф не целый")

@staticmethod

def \_\_strVertexArray(array):

"""

Выполянет преобразование элементов массива в массив из строк

:param array: массив для преобразования

:return: массив, где каждый элемент строка

"""

return [str(vertex + 1) for vertex in array]

def \_\_findPathCost(self):

"""

Функция находит стоимость пути

:return: возвращает стоимость

"""

cost = 0

for i in range(1, len(self.\_\_path)):

# увеличивем переменную суммы на значения ребра в пути

cost += self.\_\_matrix[self.\_\_path[i - 1]][self.\_\_path[i]]

return cost

def \_\_solve(self):

"""

Функция отвечает за алгоритм АВБГ

:return: ничего не возвращает

"""

# пока не добавили все вершины

while len(self.\_\_path) < len(self.\_\_matrix):

# пытаемся добавить минимальное ребро

try:

self.\_\_nextCity()

except RuntimeError as e:

print(e)

break

# нашли путь по всем вершниам, теперь проверяем, что можно зациклить

if self.\_\_matrix[self.\_\_path[-1]][self.\_\_startVertex] != math.inf:

if self.\_\_DEBUG:

print(f"Образуем цикл добавлением дуги {self.\_\_path[-1] + 1}->{self.\_\_startVertex + 1}")

self.\_\_path.append(self.\_\_startVertex)

else:

raise RuntimeError("Нельзя построить цепочку")

def \_\_call\_\_(self):

"""

Функция запускает АВБГ

:return:возвращает полученный путь и его вес

"""

# вызываем метод решения

self.\_\_solve()

# возврщаем полученные значения

return self.\_\_path, self.\_\_findPathCost()

def main():

filename = input("Введите название файла матрицы: ")

startVertex = int(input("Введите стартовую вершину (нумерация начинается с 1): ")) - 1

matrix = readMatrix(filename)

alg = AlgorithmIncludeNearestCity(matrix, startVertex, True)

path, cost = alg()

print("Цепочка:")

print("-".join([str(elem + 1) for elem in path]))

print("Стоимость её прохождения:")

print(cost)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Название файла: methodBB.py

import math

from copy import deepcopy

from readFuncs import readMatrix

from primFindMST import primFindMST

from minWeightEdges import minWeightEdges

class methodBB:

def \_\_init\_\_(self, matrix, startVertex, DEBUG=False):

# инициализация полей класса

self.\_\_matrix = matrix

self.\_\_startVertex = startVertex

self.\_\_recordPath = []

self.\_\_recordWeight = math.inf

self.\_\_DEBUG = DEBUG

def \_\_solve(self, currentPath, currentWeight):

"""

Рекурсивный метод перебирающий все возможные решение, используется МВиГ

:param currentPath: текущий путь

:param currentWeight: вес текущего пути

:return: ничего не возвращает

"""

if self.\_\_DEBUG:

print(f"Рассматривается путь: {'-'.join(self.\_\_strVertexArray(currentPath))}, его вес {currentWeight}")

# нашли путь по все вершинам

if len(currentPath) == len(self.\_\_matrix):

# проверям, что он лучше рекорда

if currentWeight + self.\_\_matrix[currentPath[-1]][self.\_\_startVertex] < self.\_\_recordWeight:

# если лучше обновляем рекорд

if self.\_\_DEBUG:

print(

f"СТАРОЕ оптимальное решиение: cost:{self.\_\_recordWeight}\npath:{'-'.join(self.\_\_strVertexArray(self.\_\_recordPath))}")

self.\_\_recordWeight = currentWeight + self.\_\_matrix[currentPath[-1]][self.\_\_startVertex]

self.\_\_recordPath = currentPath + [self.\_\_startVertex]

if self.\_\_DEBUG:

print(

f"НОВОЕ оптимальное решиение: cost:{self.\_\_recordWeight}\npath:{'-'.join(self.\_\_strVertexArray(self.\_\_recordPath))}")

else:

# если хуже то возращаемся

if self.\_\_DEBUG:

print(

f"Нашли цепочку c ценой({currentWeight + self.\_\_matrix[currentPath[-1]][self.\_\_startVertex]}):"

f"\n{'-'.join(self.\_\_strVertexArray(currentPath + [self.\_\_startVertex]))} - она не оптимальная!")

return

# отсечение ветвей хуже нижней оценки

if currentWeight > self.\_\_estimation:

if self.\_\_DEBUG:

print(f"Нижняя оценка ({self.\_\_estimation}) отсекла путь с весом ({currentWeight}):")

print(f"Отсеченный путь {'-'.join(self.\_\_strVertexArray(currentPath))}")

return

# достаем последнюю вершину пути

lastVertex = currentPath[-1]

# создаем список ещё не просмотренных вершин

notViewed = [i for i in range(len(self.\_\_matrix)) if i not in currentPath]

if self.\_\_DEBUG:

print(f"Ещё не рассмотренные вершины: {[str(vertex + 1) for vertex in notViewed]}")

for vertex in notViewed:

# поочередно добавляем вершины в путь, если в них он есть из последней вершины текущего пути

if self.\_\_matrix[lastVertex][vertex] != math.inf:

# добавляем вершину

currentPath.append(vertex)

if self.\_\_DEBUG:

print(

f"Добавляем к пути вершину ({vertex + 1}) путь: {'-'.join(self.\_\_strVertexArray(currentPath))}")

self.\_\_solve(currentPath, currentWeight + self.\_\_matrix[lastVertex][vertex])

if self.\_\_DEBUG:

print(f"Удаляем последнию вершину из пути: {'-'.join(self.\_\_strVertexArray(currentPath))}")

currentPath.pop()

else:

return

@staticmethod

def \_\_strVertexArray(array):

"""

Выполянет преобразование элементов массива в массив из строк

:param array: массив для преобразования

:return: массив, где каждый элемент строка

"""

return [str(vertex + 1) for vertex in array]

def \_\_call\_\_(self):

"""

Функция находит нижнюю границу, после чего запускает МВиГ

:return:возвращает полученный путь и его вес

"""

try:

# пытаем получить нижнюю границу

self.\_\_estimation = primFindMST(deepcopy(self.\_\_matrix), self.\_\_startVertex)[1] + minWeightEdges(

deepcopy(self.\_\_matrix))

except RuntimeError as e:

print(e)

# запускаем МВиГ

self.\_\_solve([self.\_\_startVertex], 0)

return self.\_\_recordPath, self.\_\_recordWeight

def main():

filename = input("Введите название файла матрицы: ")

startVerext = int(input("Введите стартовую вершину (нумерация начинается с 1): ")) - 1

matrix = readMatrix(filename)

MBBsolver = methodBB(matrix, startVerext, True)

path, weight = MBBsolver()

print("Цепочка:")

print("-".join([str(elem + 1) for elem in path]))

print("Стоимость её прохождения:")

print(weight)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Название файла: minWeightEdges.py

from readFuncs import readMatrix

import math

def minWeightEdges(matrix):

"""

Функция находит полусумму двух минимальных ребер

:param matrix: матрица весов

:return: возвращает полусумму двух минимальных ребер

"""

# инициализируем значения как бесконечность

first, second = math.inf, math.inf

# проходимся по все матрице

for i in range(len(matrix)):

for j in range(len(matrix)):

# если вес легче первого минимального, то это значение становится первым минимальным,

# а изначальное уходит второму

if matrix[i][j] < first:

second = first

first = matrix[i][j]

elif first < matrix[i][j] < second: # если найденное меньше только второгу, то перезаписываем второе

second = matrix[i][j]

# возвращаем полусумму

return (first + second) / 2

def main():

print(minWeightEdges(readMatrix("matrix.txt")))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: primFindMST.py

import math

from readFuncs import readMatrix

def primFindMST(matrix, startVertex=0):

"""

Функция строит минимально остовное дерево из стартовой вершины, по алгоритму Прима

:param matrix: матрица весов

:param startVertex: стартовая вершина

:return: возврщает матрицу весов остовного дерева, и вес остовного дерева

"""

size = len(matrix)

# инициализируем матрицу весов остовного дерева

mst = [[0 for \_ in range(size)] for \_ in range(size)]

for i in range(size):

mst[i][i] = math.inf

# задаем список посещенных вершин

visited = [startVertex]

weight = []

# пока не посетили все вершины

while len(visited) != size:

# ищем минимальное ребро ведущее в ещё не включенную вершину

min\_w = math.inf

i, j = 0, 0

for elem in visited:

if min\_w > min(matrix[elem]):

# найдено более выгодное ребро, сохраняем его

min\_w = min(matrix[elem])

i = elem

j = matrix[elem].index(min\_w)

elif min(matrix[elem]) == math.inf:

# если минимального нет, то граф не дерево

raise RuntimeError("The tree is not unit")

# добавляем ребро если мы ещё не посещали концевую вершину

if j not in visited:

weight.append(min\_w)

visited.append(j)

mst[i][j] = min\_w

# затираем ребро в исходной матрице, чтоб снова его не взять

matrix[i][j] = math.inf

return mst, sum(weight)

def main():

print(sum(primFindMST(readMatrix("matrix.txt"), 1)[1]))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: readFuncs.py

def readMatrix(filename) -> list:

"""

Функция считывает из файла матрицу весов

:param filename: название файла из которого считывают

:return: возврщает двумерный список (матрицу весов)

"""

file = open(filename, "r") # открываем файл на чтение

matrix = []

for row in file: # пока есть строчки в файле

line = row.split() # делим считаную строку

# выполняем привидение типа для каждого элемента строки

for index, item in enumerate(line):

try:

item = int(item)

except ValueError:

item = float(item)

line[index] = item

# преобразованую строчку добавляем в матрицу

matrix.append(line)

return matrix

def dataReader():

"""

Функция считывает из стандартного потока ввода правила генерации

:return: возвразает кортеж из кол-ва узлов, мин-веса, макс-веса, флага симметричности

"""

count = int(input("Введите количество узлов, для генерации: "))

max = int(input("Введите максимальное значение веса: "))

min = int(input("Введите минимальное значение веса: "))

symmetry = "nothing"

while symmetry not in ["yes", "not"]:

symmetry = input("Нужна ли симметричная матрица(yes or not): ")

symmetry = True if symmetry == "yes" else False

return count, min, max, symmetry

Название файла: generator.py

import numpy as np

import math

from readFuncs import dataReader

def generateMatrix() -> list:

"""

Функция запрашивает из стандартного потока ввода, правила генерации, после чего генерирует 2мерный список

:return: возвращает 2мерный список (матрицу весов)

"""

n, min, max, symmetry = dataReader() # получаем правили генерации

rng = np.random.default\_rng() # создаем объект генератора

if symmetry: # если нужна симметричная матрица весов

matrix = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

for j in range(n):

num = rng.integers(low=min, high=max)

matrix[i][j] = num

matrix[j][i] = num

else: # если полностью случайная

matrix = rng.integers(low=min, high=max, size=(n, n)).tolist()

# расстанавливаем бесконечности по главной диагонали

for i in range(n):

matrix[i][i] = math.inf

return matrix

def saveMatrix(filename, matrix):

"""

Функция сохраняет матрицу весов в файл.

:param filename: название файла, в который сохраняют

:param matrix: матрица весов

:return: функция ничего не возвращает

"""

file = open(filename, "w") # открываем на запист файл

for row in matrix:

string = " ".join([str(elem) for elem in row]) # преобразуем строку матрицы в строку

file.write(f"{string}\n") # записываем строчку в файл

file.close() # закрываем файл

def main():

filename = input("Введите название файла: ")

saveMatrix(filename, generateMatrix())

if \_\_name\_\_ == "\_\_main\_\_":

main()