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 $v_{i(1)} - v_{i(2)} - \ldots - v_{i(k-1)} - v_{i(k)}$, то следующим выбирается город v_j, ближайший к этой цепочке, т.е. имеющий минимальную из стоимостей $C_{i(q),j}$ (для $q = 1, \ldots, k$), и этот город вставляется в текущий маршрут вслед за городом $v_{i(q)}$. Тогда цепочку будет иметь вид: $v_{i(1)} - v_{i(2)} - \ldots - v_{i(q)} - v_j - v_{i(q+1)} - \ldots - v_{i(k-1)} - v_{i(k)}$

Выполнение работы.

Весь код программ представлен в приложении А.

Описание алгоритмов.

Метод ветвей и границ:

Инициализируем рекорд как бесконечность. Добавляем начальную вершину текущий путь, затем рекурсивно начинаем строить маршрут. Вычисляются оценки для оставшегося пути частичного решения (две оценки: полусумма двух легчайших ребер и вес МОД для вершин, ещё не включенных в текущее решение). Если сумма из стоимости текущего маршрута + веса минимального ребра из текущего частичного решения к любой из оставшихся вершин + максимальная из оценок становится больше, чем рекорд, то решение отбрасывается. Если все вершины были добавлены в текущее решение, то его сравниваем с рекордом, если оно меньше него, то рекорд меняем на текущее решение. 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 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.

Методы класса:

- *__findMinCostEdge(self, vertex, remainVertices)* – метод находит минимальное ребро из вершины текущего пути к оставшимся вершинам ещё не добавленным в текущее частичное решение. Аргументы: *vertex* – вершина, из которой ищут ребро, *remainVertices* – списко вершин ещё не добавленных в текущее частичное решение. Метод возвращает вес минимального ребра.
- *__solve(self, currentPath, currentWeight)* – отвечает за поиск решения методом ВиГ. 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 запускает МВиГ путем вызова *__solve*. 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, remainVertex)* – функция находит полусумму двух легчайших ребер в оставшихся вершинах ещё не включенных в путь. Аргументы функции: *matrix* – двумерный список (матрица весов графа), *remainVetex* – список

вершин, ещё не добавленных в путь. Функция возвращает найденное значение.

- *primFindMST(matrix, remainVertex)* – функция строит минимальное остовное дерево алгоритмом Прима, из оставшихся вершин ещё не включенных в путь. Аргументы функции: *matrix* – двумерный список (матрица весов графа), *remainVetex* – список вершин, ещё не добавленных в путь. 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) – O(C^n), где n – кол-во вершин, а $C \approx 1{,}26$

По памяти – O(n^2), где n – количество вершин

Алгоритм Включения Ближайшего Города:

По операциям – O(n^2), где n – количество вершин.

По памяти – O(n), 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: 3-5-6-2-1-4-3 Стоимость её прохождения: 63
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.

```
Введите название файла матрицы: matrix.txt
Введите стартовую вершину (нумерация начинается с 1): 3
Рассматривается путь: 3, его вес 0
Оценки оставшегося пути:
    По полусумме двух легчайших ребер: 3.0
    По весу МОД: 55
    Минимальное ребро из текущего пути к оставшимся вершинам: 0
Ещё не рассмотренные вершины: ['1', '2', '4', '5', '6']
Добавляем к пути вершину (1) путь: 3-1
Рассматривается путь: 3-1, его вес 20
Оценки оставшегося пути:
    По полусумме двух легчайших ребер: 3.0
    По весу МОД: 24
    Минимальное ребро из текущего пути к оставшимся вершинам: 16
Ещё не рассмотренные вершины: ['2', '4', '5', '6']
Добавляем к пути вершину (2) путь: 3-1-2
Рассматривается путь: 3-1-2, его вес 47
Оценки оставшегося пути:
    По полусумме двух легчайших ребер: 5.0
    По весу МОД: 23
    Минимальное ребро из текущего пути к оставшимся вершинам: 1
Ещё не рассмотренные вершины: ['4', '5', '6']
Добавляем к пути вершину (4) путь: 3-1-2-4
Рассматривается путь: 3-1-2-4, его вес 48
Оценки оставшегося пути:
    По полусумме двух легчайших ребер: 5.0
    По весу МОД: 5
    Минимальное ребро из текущего пути к оставшимся вершинам: 18
Ещё не рассмотренные вершины: ['5', '6']
Добавляем к пути вершину (5) путь: 3-1-2-4-5
Рассматривается путь: 3-1-2-4-5, его вес 66
Оценки оставшегося пути:
    По полусумме двух легчайших ребер: 0
    По весу МОД: 0
    Минимальное ребро из текущего пути к оставшимся вершинам: 5
Ещё не рассмотренные вершины: ['6']
Добавляем к пути вершину (6) путь: 3-1-2-4-5-6
```

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

```
Введите название файла матрицы: matrix.txt
Введите стартовую вершину (нумерация начинается с 1): 3
Путь имеет вид: 3
Уже выбранные вершины: ['3']
Ещё не добавленные вершины: ['1', '2', '4', '5', '6']
Рассматриваем дугу: 3->1 с весом 20
Рассматриваем дугу: 3->2 с весом 13
Рассматриваем дугу: 3->4 с весом 35
Рассматриваем дугу: 3->5 с весом 5
Рассматриваем дугу: 3->6 с весом 0
Нашли дугу с минимальным весом
FROM:3  TO:6 MINWEIGHT:0
Добавляем её в путь

Путь имеет вид: 3-6
Уже выбранные вершины: ['3', '6']
Ещё не добавленные вершины: ['1', '2', '4', '5']
Рассматриваем дугу: 3->1 с весом 20
Рассматриваем дугу: 3->2 с весом 13
Рассматриваем дугу: 3->4 с весом 35
Рассматриваем дугу: 3->5 с весом 5
Рассматриваем дугу: 6->1 с весом 23
Рассматриваем дугу: 6->2 с весом 5
Рассматриваем дугу: 6->4 с весом 9
Рассматриваем дугу: 6->5 с весом 5
Нашли дугу с минимальным весом
FROM:6  TO:5 MINWEIGHT:5
Добавляем её в путь

Путь имеет вид: 3-6-5
Уже выбранные вершины: ['3', '6', '5']
Ещё не добавленные вершины: ['1', '2', '4']
Рассматриваем дугу: 3->1 с весом 20
Рассматриваем дугу: 3->2 с весом 13
Рассматриваем дугу: 3->4 с весом 35
```

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()
        # возвращаем 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 __findMinCostEdge(self, vertex, remainVertices):
        """
        Метод находит минимальное ребро из последней вершины пути к
оставшимся вершинам.
        :param vertex: последняя вершина пути
        :param remainVertices: вершины ещё не добавленные в путь
        :return: вес минимального ребра из
        """
        if len(remainVertices) == 0:
            return 0
        # инициализируем ребро как бесконечность
        minEdge = math.inf
        # проходимся по всем ребрам из вершины к оставшимся, если
нашли меньше, то меняяем
        for elem in remainVertices:
            if self.__matrix[vertex][elem] < minEdge:
                minEdge = self.__matrix[vertex][elem]
```

```
        return minEdge

    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 с ценой({currentWeight +
self.__matrix[currentPath[-1]][self.__startVertex]}) >
рекорда({self.__recordWeight}):"
                        f"\n{'-
'.join(self.__strVertexArray(currentPath + [self.__startVertex]))} -
она не оптимальная!")
                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]
        # отсечение ветвей хуже текущего рекорда
        firstEstimation = minWeightEdges(deepcopy(self.__matrix),
notViewed.copy())
        secondEstimation = primFindMST(deepcopy(self.__matrix),
notViewed.copy())[1]
        minBridge = self.__findMinCostEdge(currentPath[-1],
notViewed)
```

```
        if self.__DEBUG:
            print("Оценки оставшегося пути:")
            print(f"\tПо полусумме двух легчайших ребер:
{firstEstimation}")
            print(f"\tПо весу МОД: {secondEstimation}")
            print(f"\tМинимальное ребро из текущего пути"
                  f" к оставшимся вершинам: {minBridge}")
        if currentWeight + minBridge + max(firstEstimation,
secondEstimation) > self.__recordWeight:
            if self.__DEBUG:
                print(f"Рекорд ({self.__recordWeight}) отсек путь с
весом + оценкой"
                      f"({currentWeight + minBridge +
max(firstEstimation, secondEstimation)}):")
                print(f"Отсеченный путь {'-
'.join(self.__strVertexArray(currentPath))}")
            return
        if self.__DEBUG:
            print(f"Ещё не рассмотренные вершины:
{self.__strVertexArray(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
        """
        # 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, remainVertex):
    """
    Функция находит полусумму двух легчайших ребер в оставшихся
вершинах
    :param matrix: 2мерный список матрица весов
    :param remainVertex: список оставшихся вершин
    :return: возвращает найденное значение
    """
    if len(remainVertex) <= 1:
        return 0
    if len(remainVertex) == 2:
        return (matrix[remainVertex[0]][remainVertex[1]] +
matrix[remainVertex[1]][remainVertex[0]]) / 2
    # инициализируем значения как бесконечность
    first, second = math.inf, math.inf
    for i in remainVertex:
        for j in remainVertex:
            if i == j:
                continue
            # 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]
    return (first + second) / 2


def main():
    print(attempt(readMatrix("matrix.txt"), [0, 1]))


if __name__ == "__main__":
    main()
```

Название файла: primFindMST.py

```
import math
from readFuncs import readMatrix


def primFindMST(matrix, remainVertex):
    """
    Функция строит минимальное остовное дерево для оставшихся
вершин, по алгоритму Прима.
    :param matrix: 2мерный список матрица весов
    :param remainVertex: список оставшихся вершин
    :return: возвращает вес минимального остовного дерева
    """
    # инициализируем матрицу весов остовного дерева
    mst = [[0 for _ in range(len(matrix))] for _ in
range(len(matrix))]
    for i in range(len(matrix)):
        mst[i][i] = math.inf
    size = len(remainVertex)
    if size <= 1:
        # если одна вершина то МОД равен 0
        return mst, 0
    # задаем список посещенных вершин
    visited = [remainVertex[0]]
    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:
            for newVertex in [vertex for vertex in remainVertex if
vertex not in visited]:
                if min_w > matrix[elem][newVertex]:
                    # найдено более выгодное ребро, сохраняем его
                    min_w = matrix[elem][newVertex]
                    i = elem
                    j = newVertex
        # добавляем ребро если мы ещё не посещали концевую вершину
        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(primFindMST(readMatrix("matrix.txt"), [0, 1, 2, 4, 5]))


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()
```