

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине Построение и анализ алгоритмов
Тема: «Кратчайшие пути в графе: коммивояжёр»

Студент гр. 3343

Преподаватель

Пивоев Н. М.

Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Изучить принцип работы алгоритмов Литтла и АДО МОД и реализовать их на практике.

Задание.

Вариант 1

Метод Ветвей и Границ: Алгоритм Литтла. Приближённый алгоритм: 2-приближение по Мид (Алгоритм двойного обхода минимального остовного дерева). Замечание к варианту 1 АДО МОД является 2-приближением только для евклидовой матрицы. Начинать обход МОД со стартовой вершины.

Независимо от варианта, при сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную), сохранять её в файл и использовать в качестве входных данных.

Алгоритм Литтла

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны закланием процветания. Для этого ему необходимо посетить каждый город ровно один раз, создавая тропу благополучия, и вернуться обратно в столицу, используя минимум своих чародейских сил. Вашей задачей является помощь в прокладывании маршрута с помощью древнего и могущественного алгоритма ветвей и границ.

Карта дорог Алгоритмии перед Гамильтоном представляет собой полный граф, где каждый город соединён магическими порталами с каждым другим. Стоимость использования портала из города в город занимает определённое количество маны, и Гамильтон стремится минимизировать общее потребление магической энергии для закрепления проклятия.

Входные данные:

Первая строка содержит одно целое число N (N — количество городов). Города нумеруются последовательными числами от 0 до $N-1$. Следующие N строк содержат по N чисел каждая, разделённых пробелами, формируя таким образом матрицу стоимостей M . Каждый элемент $M_{i,j}$ этой матрицы представляет собой затраты маны на перемещение из города i в город j .

Выходные данные:

Первая строка: Список из N целых чисел, разделённых пробелами, обозначающих оптимальный порядок городов в магическом маршруте Гамильтона. В начале идёт город 0, с которого начинается маршрут, затем последующие города до тех пор, пока все они не будут посещены. Вторая строка: Число, указывающее на суммарное количество израсходованной маны для завершения пути.

АДО МОД

Разработайте программу, которая решает задачу коммивояжера при помощи 2-приближенного алгоритма. В данной постановке задачи нужно вернуться в исходную вершину после прохождения всех остальных вершин. При обходе остовного дерева (MST) необходимо идти по минимальному допустимому ребру из текущего. Каждая вершина в графе обозначается неотрицательным числом, начиная с 0, каждое ребро имеет неотрицательный вес. В графе **нет рёбер из вершины в саму себя**, в матрице весов на месте таких *отсутствующих рёбер* стоит значение **-1**.

В первой строке указывается начальная вершина. Далее идёт матрица весов.

В качестве выходных данных необходимо представить длину пути, полученного при помощи алгоритма. Следующей строкой необходимо представить путь, в котором перечислены вершины, по которым необходимо пройти от начальной вершины.

Выполнение работы

Описание алгоритма Литтла

Алгоритм Литтла – алгоритм решения задачи коммивояжёра, основанный на методе ветвей и границ. Он позволяет находить наиболее оптимальный путь по матрице расстояний. Алгоритм состоит из следующих шагов:

1) Сначала происходит вычитание минимального значения строки из всех элементов этой строки. То же самое повторяется для всех строк, а затем для столбцов.

2) Затем идёт поиск нулевых элементов матрицы, среди них выбирается тот, в строке и ряде которого находятся минимальные элементы.

3) Далее идёт ветвление на два разных случая. В первом удаляется строка и столбик, содержащие выбранный нулевой элемент, а также запрещаются рёбра, которые могут привести к недопустимым решениям и алгоритм начинается заново для новой матрицы. Во втором рассматриваемый нулевой элемент заменяется на бесконечность и алгоритм начинается заново.

Каждый раз полученная стоимость сравнивается с рекордной. Если полученная стоимость больше минимальной, то текущая ветвь решения не рассматривается.

Описание АДО МОД

Алгоритм двойного обхода минимального остовного дерева – алгоритм поиска приближённого решения задачи коммивояжёра. Сначала находится минимальное остовное дерево, с помощью алгоритма Крускала и строится новый граф, в котором каждая вершина исходного соединена с ближайшей вершиной в МОД. Затем поиском в глубину, начиная с заданной вершины, составляется гамильтонов цикл, составляющий замкнутый цикл, стоимость которого вычисляется как сумма весов рёбер в нём.

Оценка сложности алгоритма Литтла

Временная сложность:

Вычитание минимальных значений происходит за $O(n^2)$, где n – длина одной из сторон матрицы.

Обработка нулевых коэффициентов происходит за $O(n^2)$.

Ветвление происходит за $O(2^{n-1} \cdot n^2)$, поскольку в худшем случае необходимо обойти полное бинарное дерево (на каждом шаге мы можем включить или исключить ребро, а так как в конечном пути всего n вершин, то такой выбор придётся сделать максимум $n-1$ раз, потому что каждый раз добавляется вершина и ребёр на 1 меньше, чем вершин), а на каждом этапе необходимо вычитать минимальные значения и обрабатывать нулевые коэффициенты.

Временная сложность - $O(2^{n-1} \cdot n^2)$.

Пространственная сложность – тоже $O(2^{n-1} \cdot n^2)$, поскольку на каждом этапе создаётся копия матрицы, в худшем случае их $O(2^{n-1})$.

Оценка сложности АДО МОД

Временная сложность:

В методе Крускала происходит обход половины матрицы за $O(E^2)$, сортировка за $O(E \log E)$, поиск родителей вершин за $O(E^2)$, где E – количество рёбер в графе для одной вершины ($E = n-1$).

Поиск в глубину рассчитывается за $O(E + V)$, поскольку нужно обойти все вершины и рёбра, где V – все вершины.

Временная сложность - $O(E^2 + V)$.

Пространственная сложность – $O(E^2)$, поскольку составляется массив рёбер, размером E^2 , остальные массивы имеют размер E .

Код программы содержит реализацию следующих функций:

Алгоритм Литтла

- *subtract_min_from_matrix(self)* – вычитает из каждой ячейки минимальное значение для данной строки и столбца, потому что это значение и так будет заложено в минимальную стоимость.
- *coef_finder(self, row, column)* – находит минимальный элемент в выбранной строке и столбце, возвращает их сумму.
- *find_zero_coefs* – поиск нулевого элемента, у которого сумма двух элементов из той же строки и столбца минимальна.
- *find_longest_path(self, path, edge)* – находит самый длинный путь в графе основываясь на текущем ребре.
- *process_path(self, path, x_ind, y_ind)* – запрещает ребро, оканчивающее цикл в графе.
- *reduce_matrix(self, coordinate, path, x_ind, y_ind)* – удаляет строку и столбец матрицы, в которых находился выбранный нулевой элемент.
- *check_solution(self, path, current_cost, x_ind, y_ind)* – обработка матрицы размером 2x2 для нахождения более выгодной стоимости.
- *solve(self, matrix, path, lower_limit_x_ind, y_ind, iteration)* – главная функция, объединяющая все остальные для нормальной работы алгоритма. Также отвечает за ветвление матрицы.

АДО МОД

- *find(parent, node)* – используется для поиска родителя.
- *kruskal_mod(self)* – строит минимальное остовное дерево.
- *dfs(self, node, adjacent, visited, path)* – поиск в глубину для остовного дерева и построения пути
- *solve(self, start)* – главная функция, объединяющая все остальные.

Тестирование

Программа была протестирована на различных входных данных. Для удобства тестирования, создан генератор матриц, подходящий для обоих алгоритмов. Рёбра на главной диагонали – inf, а остальные – float числа. Соответственно для алгоритма Литтла float значения округляются, а для АДО МОД inf обрабатываются как -1.

Таблица 1.

Входные данные	Выходные данные	Комментарий
3 -1 1 3 3 -1 1 1 2 -1	0 1 2 3.0	Алгоритм Литтла
4 -1 3 4 1 1 -1 3 4 9 2 -1 4 8 9 2 -1	0 3 2 1 6.0	Алгоритм Литтла
3 -1 1 1 1 -1 1 1 1 -1	0 1 2 3.0	Алгоритм Литтла
1 -1 50.1 5.45 58.91 -1 4.36 19.0 66.71 -1	73.46 1 2 0 1	АДО МОД
2 -1 18.97 22.36 19.42 3.61 18.97 -1 35.61 38.01 17.0 22.36 35.61 -1 16.28 21.19	91.92 2 3 0 4 1 2	АДО МОД

19.42 38.01 16.28 -1 21.02 3.61 17.0 21.19 21.02 -1		
-1 1 1 1 -1 1 1 1 -1	3.0 2 0 1 2	АДО МОД

```

Введи количество городов (размер матрицы)
3

[inf, 25.43, 1.5]
[18.29, inf, 51.15]
[55.05, 3.92, inf]

1 Сгенерировать обычную матрицу
2 Сгенерировать симметричную матрицу
3 Сгенерировать евклидову матрицу
4 Загрузить матрицу
5 Сохранить матрицу
6 Метод Литтла
7 Метод АДО МОД
8 Выход
6

Левая ветвь. Итерация: 0
Текущая стоимость: 22
Найденное решение: {2: 1, 3: 2, 1: 3}
Правая ветвь. Итерация 0
Текущая стоимость: 22
Затраченное время: 0.00012430000060703605
0 2 1
22.0

```

Рисунок 1 – Результат работы программы

Визуализация графа с выделенным путём обхода

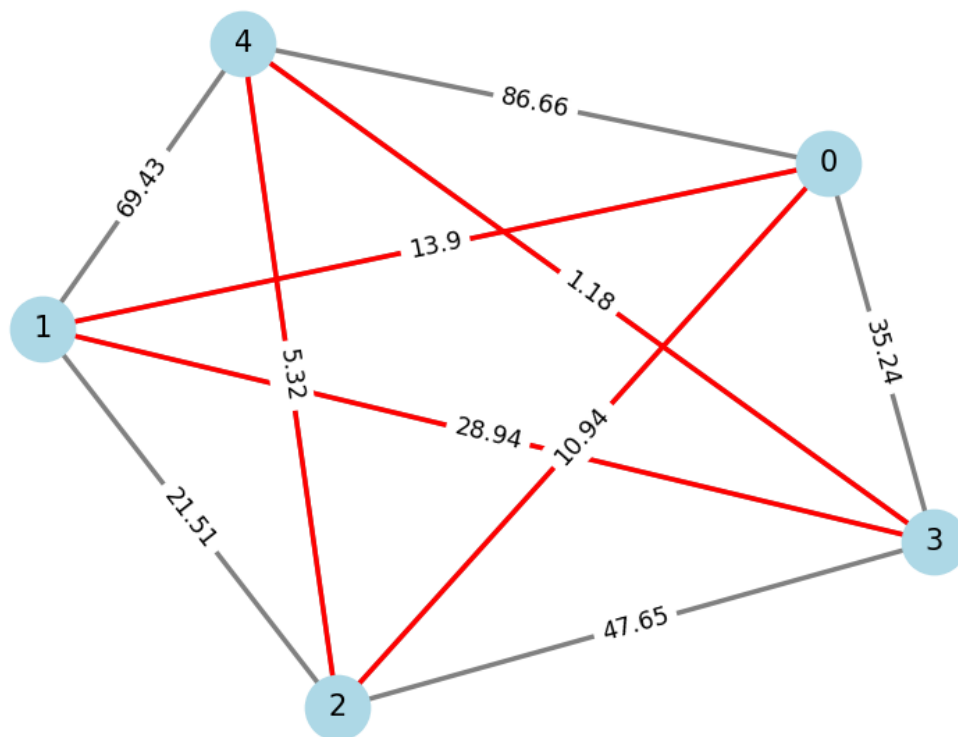


Рисунок 2 – Визуализация для АДО МОД

Исследование

Исследуем эффективность обоих алгоритмов на входных данных разного размера.

Таблица 2. Исследование эффективности по времени.

Алгоритм Литтла	
N	Время в секундах
5	0.000177
10	0.004046
20	0.048956
30	0.412179
40	0.863758
АДО МОД	
N	Время в секундах
100	0.004695
250	0.020719
500	0.130701
750	0.418319
1000	0.707339

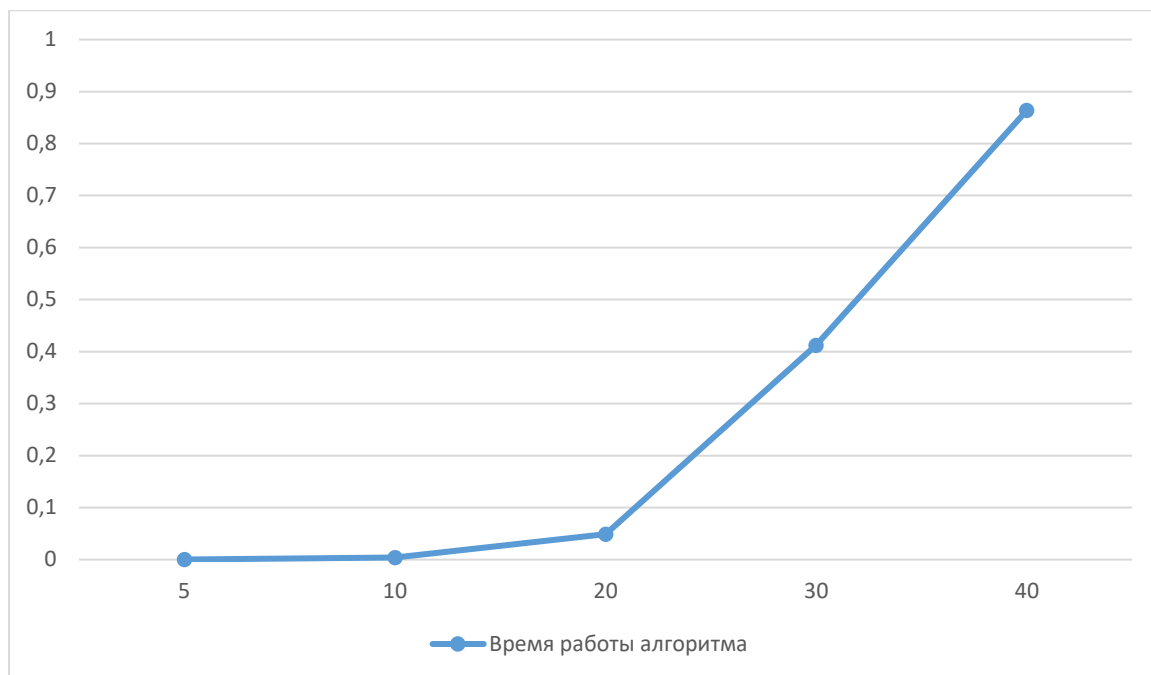


Рисунок 3 – График зависимости затраченного времени от размера матрицы для алгоритма Литтла

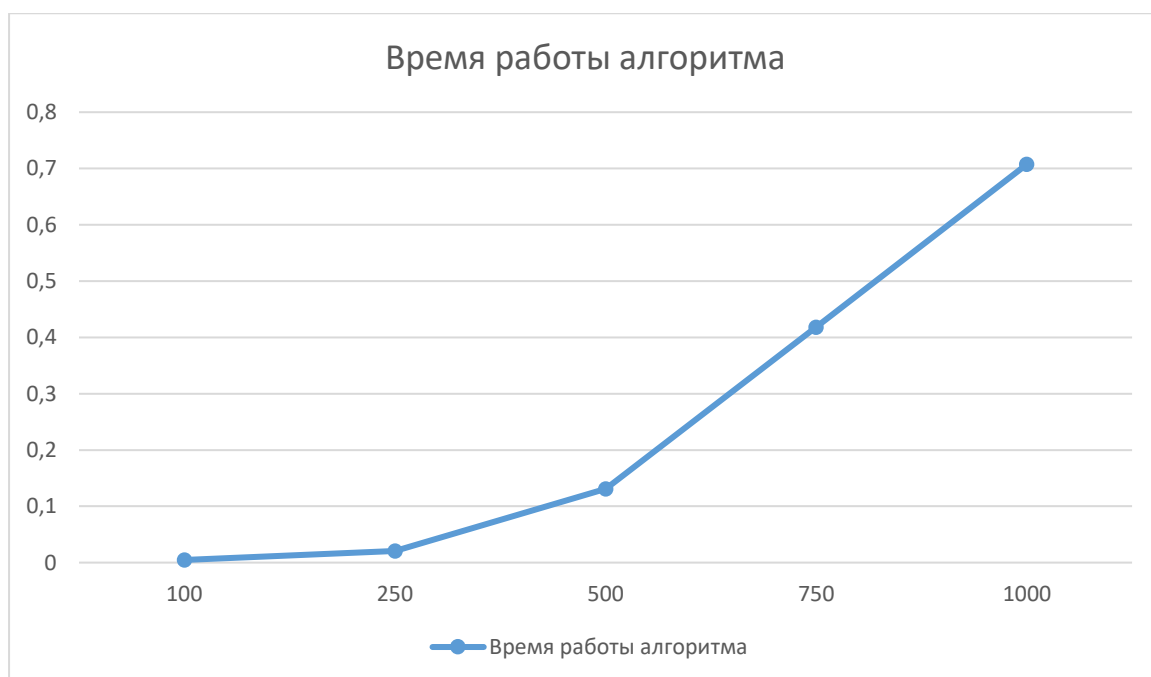


Рисунок 4 – График зависимости затраченного времени от размера матрицы для АДО МОД

Полученные данные доказывают, что время работы алгоритма Литтла возрастает крайне быстро (экспоненциально), поэтому он неэффективен даже для относительно небольших n .

Приближённый АДО МОД выполняется намного быстрее, поэтому его можно использовать для нахождения приближённого решения при больших n , а алгоритм Литтла для точного результата при малых n .

Выводы

Во время выполнения лабораторной работы, была изучена работа алгоритма Литтла и АДО МОД. Решены задачи поиска точного и приближённого расстояния коммивояжёра.

ПРИЛОЖЕНИЕ

ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.py

```
import math
import copy
import time

from matrix_generator import MatrixGenerator
from little_alg import LittleSolver
from ado_mod_alg import ADO_MOD
from visualization import visualize_mst

def print_matrix(matrix):
    print()
    for i in matrix:
        print(i)
    print()

class Handler:
    def __init__(self):
        self.matrix_generator = MatrixGenerator()
        self.matrix = []

    def generate_matrix(self, matrix_type):
        print("Введи количество городов (размер матрицы)")
        n = int(input())

        if matrix_type == "normal":
            self.matrix = self.matrix_generator.generate_normal_matrix(n)
        elif matrix_type == "symmetrical":
            self.matrix = self.matrix_generator.generate_symmetrical_matrix(n)
        elif matrix_type == "euclidean":
            self.matrix = self.matrix_generator.generate_euclidean_matrix(n)

        if n <= 15:
            print_matrix(self.matrix)
```

```

def load_matrix(self):
    file = open("text.txt", 'r')
    self.matrix = []
    for row in file:
        current = []
        for i in row.split():
            if i == "inf" or i == "math.inf":
                current.append(math.inf)
            else:
                current.append(float(i))
        self.matrix.append(current)
    if len(self.matrix) <= 15:
        print_matrix(self.matrix)

    file.close()

def save_matrix(self):
    if self.matrix is []:
        print("Матрица пустая\n")
        return

    file = open("text.txt", 'w')

    for row in self.matrix:
        file.write(' '.join(str(i) for i in row) + '\n')

    file.close()

def little_init(self):
    if self.matrix is []:
        self.generate_matrix("normal")

    copy_matrix = copy.deepcopy(self.matrix)

    for i in range(len(copy_matrix[0])):
        for j in range(len(copy_matrix[0])):
            if i != j:
                copy_matrix[i][j] = int(copy_matrix[i][j])
            else:
                copy_matrix[i][j] = math.inf

    l = LittleSolver(copy_matrix)

```



```

x_ind = [x for x in range(len(copy_matrix))]
y_ind = [y for y in range(len(copy_matrix))]
s = time.perf_counter()
l.solve(copy_matrix, {}, 0, x_ind, y_ind, 0)
e = time.perf_counter()
print("Затраченное время:", e - s)

nxt = l.arcs[1]
res = [0]
while nxt != 1:
    res.append(nxt - 1)
    nxt = l.arcs[nxt]
print(*res, sep=" ")

print(l.record / 1)

def ado_mod_init(self):
    if self.matrix is []:
        self.generate_matrix("normal")

    print("Введи начальную вершину")
    start = int(input())
    if start >= len(self.matrix[0]):
        print("Неправильная нумерация\n")
        return

    matrix = copy.deepcopy(self.matrix)
    s = time.perf_counter()
    path = ADO_MOD(matrix).solve(start)
    e = time.perf_counter()
    print("Затраченное время:", e - s)

    if len(path) <= 15:
        visualize_mst(self.matrix, path)

def start(self):
    print("Доступные команды:")
    while True:
        print("1 Сгенерировать обычную матрицу\n"
              "2 Сгенерировать симметричную матрицу\n"
              "3 Сгенерировать евклидову матрицу\n"
              "4 Загрузить матрицу\n"
              "5 Сохранить матрицу\n")

```

```

        "6 Метод Литтла\n"
        "7 Метод АДО МОД\n"
        "8 Выход")
choice = input()
print()

if choice == '1':
    self.generate_matrix("normal")
elif choice == '2':
    self.generate_matrix("symmetrical")
elif choice == '3':
    self.generate_matrix("euclidean")
elif choice == '4':
    self.load_matrix()
elif choice == '5':
    self.save_matrix()
elif choice == '6':
    self.little_init()
elif choice == '7':
    self.ado_mod_init()
elif choice == '8':
    return

if __name__ == "__main__":
    handler = Handler()
handler.start()

```

Имя файла: visualization.py

```

import matplotlib.pyplot as plt
import networkx as nx

def visualize_mst(weight_matrix, path):
    # Создаём граф
    G = nx.Graph()

    # Добавляем узлы (их количество = длине матрицы)
    num_nodes = len(weight_matrix)
    G.add_nodes_from(range(num_nodes))

    # Добавляем рёбра с весами

```

```

    for i in range(num_nodes):
        for j in range(i + 1, num_nodes): # Чтобы избежать дублирования
(A-B и B-A)

            weight = weight_matrix[i][j]
            if weight != 0: # Если вес не нулевой, добавляем ребро
                G.add_edge(i, j, weight=weight)

# Позиционирование узлов (можно менять layout)
pos = nx.spring_layout(G) # Альтернативы: circular_layout, shell_layout,
kamada_kawai_layout

# Рисуем граф
plt.figure(figsize=(8, 6))
nx.draw_networkx_nodes(G, pos, node_size=700, node_color='lightblue')
nx.draw_networkx_labels(G, pos, font_size=12, font_family='sans-serif')

# Рисуем все рёбра серым цветом
nx.draw_networkx_edges(G, pos, width=2, edge_color='gray')

# Выделяем рёбра из path красным цветом
path_edges = []
for i in range(len(path) - 1):
    u = path[i]
    v = path[i + 1]
    if G.has_edge(u, v):
        path_edges.append((u, v))

nx.draw_networkx_edges(G, pos, edgelist=path_edges, width=2,
edge_color='red')

# Добавляем подписи весов рёбер
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=10)

plt.title("Визуализация графа с выделенным путём обхода")
plt.axis('off') # Отключаем оси
plt.show()

```

Имя файла: ado_mod_alg.py

```
import math
```

```
def find(parent, node):
```

```

while parent[node] != node:
    parent[node] = parent[parent[node]]
    node = parent[node]
return node

class ADO_MOD:
    def __init__(self, matrix):
        self.matrix = matrix

    def kruskal_mod(self):
        edges = []
        n = len(self.matrix)

        # Вес ребра, текущая вершина, соединяемая вершина
        for u in range(n):
            for v in range(u + 1, n):
                if self.matrix[u][v] != -1 or self.matrix[u][v] != math.inf:
                    edges.append((self.matrix[u][v], u, v))

        edges.sort() # Сортировка по весу
        parent = list(range(n))

        mst = [[] for _ in range(n)]
        for _, u, v in edges:
            u_root, v_root = find(parent, u), find(parent, v)
            if u_root != v_root:
                mst[u].append(v) # Удвоение рёбер
                mst[v].append(u)
                parent[v_root] = u_root

        return mst

    def dfs(self, node, adjacent, visited, path):
        visited.add(node)
        path.append(node)
        for v in sorted(adjacent[node], key=lambda x: self.matrix[node][x]):
            if v not in visited:
                self.dfs(v, adjacent, visited, path)

    def solve(self, start):
        mst = self.kruskal_mod()

        path = []
        self.dfs(start, mst, set(), path)
        path.append(start)

        print(round(sum(self.matrix[path[i]][path[i + 1]] for i in
range(len(path) - 1)), 2))
        print(*path)
        return path

```

Имя файла: little_alg.py

```

import math
import copy

MAXIMUM = math.inf

class LittleSolver:
    def __init__(self, matrix):

```

```

self.matrix = copy.deepcopy(matrix)
self.record = math.inf
self.arcs = {}

def subtract_min_from_matrix(self):
    subtracted_sum = 0

    for i in range(len(self.matrix)):
        mn = min([w for w in self.matrix[i] if w != -math.inf])
        subtracted_sum += mn
        for j in range(len(self.matrix)):
            if self.matrix[i][j] != math.inf:
                self.matrix[i][j] -= mn

    for i in range(len(self.matrix)):
        mn_column = min([row[i] for row in self.matrix])
        for row in self.matrix:
            row[i] -= mn_column
        subtracted_sum += mn_column
    return subtracted_sum

def coef_finder(self, row, column):
    mn_row = mn_column = MAXIMUM
    for i in range(len(self.matrix)):
        if i != row:
            mn_row = min(mn_row, self.matrix[i][column])
        if i != column:
            mn_column = min(mn_column, self.matrix[row][i])

    return mn_row + mn_column

def find_zero_coefs(self):
    zeros = []
    coefs = []
    mx_coef = 0

    for i in range(len(self.matrix)):
        for j in range(len(self.matrix)):
            if self.matrix[i][j] == 0:
                zeros.append([i, j])

            coefs.append(self.coef_finder(i, j))
            mx_coef = max(mx_coef, coefs[-1])

    for i in range(len(coefs)):
        if coefs[i] == mx_coef:
            return zeros[i]
    exit(1)

def find_longest_path(self, path, edge):
    start, end = edge
    end_path = []

    current = start
    while current in path.keys():
        end_path.append(path[current])
        current = path[current]

    current = start
    end_path.insert(0, current)
    while current in path.values():
        for k in path.keys():
            if path[k] == current:
                current = k

```

```

        end_path.insert(0, current)
    return end_path

# Запрещает циклы (например 0 - 1 - 2 - 0)
def process_path(self, path, x_ind, y_ind):
    if len(path) < 2:
        return

    re_x = path[-1]
    re_y = path[0]

    if re_x - 1 in x_ind and re_y - 1 in y_ind:
        self.matrix[x_ind.index(re_x - 1)][y_ind.index(re_y - 1)] =
math.inf

def reduce_matrix(self, coordinate, path, x_ind, y_ind):
    re_x = x_ind[coordinate[0]]
    re_y = y_ind[coordinate[1]]

    path[re_x + 1] = re_y + 1
    longest = self.find_longest_path(path, (re_x + 1, path[re_x + 1]))
    self.process_path(longest, x_ind, y_ind)

    x_ind.pop(coordinate[0])
    y_ind.pop(coordinate[1])
    self.matrix.pop(coordinate[0])
    for row in self.matrix:
        row.pop(coordinate[1])

def check_solution(self, path, current_cost, x_ind, y_ind):
    for x in range(len(self.matrix)):
        for y in range(len(self.matrix)):
            if self.matrix[x][y] == math.inf:
                path[x_ind[(x + 1) % 2] + 1] = y_ind[y] + 1
                path[x_ind[x] + 1] = y_ind[(y + 1) % 2] + 1
                self.arcs = path
                self.record = current_cost

def solve(self, matrix, path, lower_limit, x_ind, y_ind, iteration):
    self.matrix = matrix
    diff_cost = self.subtract_min_from_matrix()

    current_cost = diff_cost + lower_limit
    if current_cost >= self.record:
        return

    if len(matrix) == 2:
        self.check_solution(path, current_cost, x_ind, y_ind)
        print("\033[32mНайденное решение:\033[0m", self.arcs)
        return

    zeros = self.find_zero_coefs()
    new_path = copy.deepcopy(path)
    m1 = copy.deepcopy(matrix)
    new_x_ind = x_ind.copy()
    new_y_ind = y_ind.copy()

    self.matrix = m1
    self.reduce_matrix(zeros, new_path, new_x_ind, new_y_ind)

    print(f"\033[31mЛевая ветвь\033[0m. Итерация: {iteration}")
    print(f"Текущая стоимость: {current_cost}")

```

```

        self.solve(m1, new_path, current_cost, new_x_ind, new_y_ind,
iteration + 1)

        matrix=zeros[0][zeros[1]] = math.inf

        print(f"\033[34mПравая ветвь\033[0m. Итерация {iteration}")
        print(f"Текущая стоимость: {current_cost}")

        self.solve(matrix, path, current_cost, x_ind.copy(), y_ind.copy(),
iteration + 1)

```

Имя файла: matrix_generator.py

```

import math
import random

class MatrixGenerator:
    def __init__(self) -> None:
        pass

    def generate_normal_matrix(self, n):
        matrix = []
        for i in range(n):
            row = [round(random.uniform(1, 100), 2) for _ in range(n)]
            row[i] = math.inf
            matrix.append(row)
        return matrix

    def generate_symmetrical_matrix(self, n):
        matrix = [[0.0 for _ in range(n)] for _ in range(n)]
        for i in range(n):
            for j in range(i, n):
                a = round(random.uniform(1, 100), 2)
                matrix[i][j] = a
                matrix[j][i] = a
            if i == j:
                matrix[i][j] = math.inf
        return matrix

    def generate_euclidean_matrix(self, n, dimensions=2):
        points = [[round(random.uniform(0, 100), 2) for _ in
range(dimensions)] for _ in range(n)]
        matrix = [[0 for _ in range(n)] for _ in range(n)]

        for i in range(n):

```

```

        for j in range(n):
            if i == j:
                matrix[i][j] = math.inf
            else:
                distance = math.sqrt(sum((points[i][k] - points[j][k])
** 2 for k in range(dimensions)))
                matrix[i][j] = int(distance)
                matrix[j][i] = int(distance)
    return matrix

```