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: Алгоритмы для решения задачи коммивояжёра

Студентка гр. 3343 Гельман П.Е.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Цель данной лабораторной работы состоит в изучении модифицированного алгоритма Литтла и алгоритма ближайшего соседа для нахождения решения задачи коммивояжера.

Задание.

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны заклятием 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.

Sample Input 1:

3

-1 1 3

3 -1 1

1 2 -1

Sample Output 1:

0 1 2

3.0

Sample Input 2:

4

-1 3 4 1

1 -1 3 4

9 2 -1 4

8 9 2 -1

Sample Output 2:

0 3 2 1

6.0

Задание варианта:

Вариант 2. ·

2 МВиГ: Алгоритм Литтла с модификацией: после приведения матрицы, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. Приближённый алгоритм: АБС. Замечание к варианту 2 Начинать АБС со стартовой вершины.

Примечания для варианта:

Независимо от варианта, при сдаче работы должна быть возможность генерировать матрицу весов (произвольную или симметричную; для варианта 6 —симметричную), сохранять её в файл и использовать в качестве входных данных.

Выполнение работы.

Реализованный функционал:

1. Класс Node – класс узла бинарного дерева поиска алгоритма МВиГ. Каждый узел содержит поле матрицы, нижней границы, текущего маршрута. def get_cell_with_max_penalty(self) – метод, который находит в матрице весов клетку с минимальным штрафом за ее удаление (клетку с 0, у которой сумма минимальных элементов в строке и столбце максимальна). def make_children(self) – метод, которые создает двух потомков текущего узла – левого и правого. Левые узлы отвечают за исключение текущего ребра, правые – за включение ребра в маршрут. Для левого потомка в матрице запрещается текущее ребро (=inf), для правого запрещается обратное ребро, а также столбец и строка соответствующие пункту прибытия и пункту отправления. Также запрещаются все ребра, которые могут создать подциклы в текущем маршруте и досрочно завершить путь.
2. def clone_matrix(matrix) – клонирование матриц
3. def row_mins(matrix) – нахождение минимумов в строках матрицы
4. def column_mins(matrix) – нахождение минимумов в столбцах
5. def sum_finites(arr) – суммирует минимумы в переданном массиве
6. def reduce_rows(matrix, mins) – редуцирование матрицы по строкам
7. def reduce_columns(matrix, mins) – редуцирование матрицы по столбцам
8. def reduce(matrix, route) – полная редукция матрицы с добавлением к нижней оценке веса решения нижней оценки суммарного веса остатка пути на основе МОД.
9. def find_next_start_city(edges, start_city) – нахождение следующего номера ребра, которое ведет в заданный город.
10. def find_next_end_city(edges, end_city) – нахождение следующего номера ребра, которое выходит из заданного города.

11. def get_close_edges(route) – нахождение ребер, которые могут досрочно завершать поиск пути. Сначала ищутся ребра, продолжающие цепочку в начало, затем которые продолжают в конец.
12. def little(matrix) – функция алгоритма Литтла. Инициализируется корень дерева, ищется минимальная начальная граница стоимости маршрута. Создается очередь с приоритетом для хранения узлов, из нее извлекается узел с минимальной нижней границей, далее идет проверка, если осталось одно ребро до окончания маршрута, то добавлчем его, проверяем текущее решение на минимальность и завершаем программу, иначе находим правого и левого потомков, и добавляем их в очередь.
13. def print_matrix(matrix) – печать матрицы
14. def generate_matrix(n, symmetric=False, min_weight=1, max_weight=100, infinity_prob=0.05) – генерация матрицы, n –размер, symmetric – симметричная/произвольная, min/max_weight – диапазон значений матрицы, infinity_prob – вероятность, что какой-то элемент не на главной диагонали будет бесконечностью.
15. def save_matrix_to_file(matrix, filename) – сохранение матрицы в файл
16. def load_matrix_from_file(filename) – выгрузка матрицы из файла
17. def prim_mst(matrix, route) – модифицированный алгоритм Прима с учетом кусков. В функции находим куски текущего маршрута, если все вершины – один кусок, то завершаем, иначе строим новую урезанную матрицу стоимостей допустимых ребер между кусками, к этой матрице применяем обычный алгоритм поиска МОД.
18. def find_chunks(route, matrix) – поиск кусков в текущем маршруте.
19. def build_chunk_matrix(matrix, chunks) – построение матрицы весов из найденных кусков.
20. def standard_prim(chunk_matrix) – обычный алгоритм Прима для нахождения МОД.

21. def nearest_neighbor(matrix, st_city=0) – приближенный алгоритм ближайшего соседа. Он последовательно строит маршрут, на каждом шаге выбирая ближайший непосещенный город, пока не обойдет все города, после чего возвращается в начальный пункт. Начинается со стартового города.

Сложность алгоритма.

АБС:

- По памяти O(n), так как храним множество непосещенных городов unvisited и посещенных route.
- По времени $O(n^2)$. На каждом из n шагов алгоритм перебирает всех непосещённых соседей (в худшем случае n-1, n-2, ..., 1), что даёт сумму арифметической прогрессии.

Алгоритм Литтла с модификацией:

По времени

Сложность точного алгоритма ЗК (методом ВиГ) в среднем (при «случайных» матрицах стоимостей)

$$\approx C^n, \text{ где } C \cong 1.26$$

Однако, можно учесть еще сложность алгоритма Прима для кусков

- Алгоритм Прима: O(m log(n)), m – ребра, n - вершины
- Поиск кусков: O(n) анализ ребер маршрута
- Построение матрицы кусков: O(k^2) – k – число кусков (в худ. случае n)

По памяти

- O(n^2) для матрицы весов
- Память на узел O(n^2) на матрицу, O(n) на маршрут, общая память на дерево O(k * n^2 + k*n), k – число активных узлов дерева.
- Приоритетная очередь O(k) хранит все активные узлы

- Итого ~ O(k * n^2)

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Коммен тарии
1.	1 2 3 --- 1 | -1 33 44 2 | 10 -1 91 3 | 78 13 -1	0 2 1 67.0	Верно
2.	1 2 3 4 --- 1 | -1 6 24 44 2 | 23 -1 46 7 3 | 51 72 -1 83 4 | 78 96 75 -1	0 1 3 2 139.0	Верно
3.	1 2 3 4 5 --- 1 | -1 15 95 61 27 2 | 15 -1 64 65 53 3 | 95 64 -1 30 7 4 | 61 65 30 -1 2 5 | 27 53 7 2 -1	0 1 2 3 4 138.0	Верно
4.	1 2 3 4 --- 1 | -1 6 85 87 2 | -1 -1 72 19 3 | 69 86 -1 24 4 | 33 73 53 -1 Стартовый город 1	РЕЗУЛЬТАТ: Оптимальный маршрут: 1 → 2 → 4 → 3 → 1 Общая длина пути: 147	Верно
5.	1 2 3 --- 1 | -1 49 47 2 | 49 -1 72 3 | 47 72 -1 2	РЕЗУЛЬТАТ: Оптимальный маршрут: 2 → 1 → 3 → 2 Общая длина пути: 168	Верно
6.	1 2 3 4 5 6 --- 1 | -1 -1 71 7 55 10	Результат алгоритма Литтла: 0 5 3 1 2 4 155.0	Верно

	2 | 43 -1 51 88 86 46 3 | 80 61 -1 74 24 58 4 | 2 8 27 -1 97 70 5 | 47 80 71 67 -1 81 6 | -1 74 83 15 83 -1	РЕЗУЛЬТАТ: Оптимальный маршрут: 1 → 4 → 2 → 6 → 3 → 5 → 1 Общая длина пути: 215	

Выводы.

В ходе лабораторной работы был реализован алгоритм Литтла с модификацией и алгоритм ближайшего соседа, проанализирована их временная сложность и сложность по памяти.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import heapq
import math
import random


def print_matrix(matrix):
    n = len(matrix)
    header = " " * 5 + " ".join(f"{i + 1:4}" for i in range(n))
    print(header)
    print(" " * 5 + "-" * len(header[5:]))
    for i in range(n):
        row_str = f"{i + 1:2} | "
        row_elements = []
        for j in range(n):
            if matrix[i][j] == math.inf:
                row_elements.append("  -1")
            else:
                row_elements.append(f"{int(matrix[i][j]):4}")
        row_str += " ".join(row_elements)
        print(row_str)


def generate_matrix(n, symmetric=False, min_weight=1, max_weight=100,
infinity_prob=0.05):
    """Генерация матрицы весов (симметричной или произвольной)"""
    matrix = [[math.inf if i == j else 0 for j in range(n)] for i in
range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                continue  # диагональ уже заполнена бесконечностями

            if symmetric and j < i:
                # для симметричной матрицы используем уже заданные значения
                matrix[i][j] = matrix[j][i]
            else:
                # генерируем новое значение
                if random.random() < infinity_prob:  # вероятность
генерирования инф
                    weight = math.inf
                else:
                    weight = random.randint(min_weight, max_weight)
                matrix[i][j] = weight

                # если матрица симметричная, зеркально отражаем значение
                if symmetric and j > i:
                    matrix[j][i] = weight

    return matrix


def save_matrix_to_file(matrix, filename):
    """Сохранение матрицы в файл"""
    with open(filename, 'w') as f:
        n = len(matrix)
        f.write(f"{n}\n")
```

```
        for row in matrix:
            formatted_row = [str(int(x) if x != math.inf else -1) for x in
row]
            f.write(" ".join(formatted_row) + "\n")


def load_matrix_from_file(filename):
    """Чтение матрицы из файла"""
    try:
        with open(filename, 'r') as f:
            n = int(f.readline())
            matrix = []
            for _ in range(n):
                row = [(x if x != -1 else math.inf) for x in list(map(int,
f.readline().split()))]
                print(row)
                matrix.append(row)
        return matrix
    except FileNotFoundError:
        print(f"Ошибка: файл '{filename}' не найден")
        return None
    except Exception as e:
        print(f"Ошибка при чтении файла: {e}")
        return None


def prim_mst(matrix, route):
    """Модифицированный алгоритм Прима для оценки остатка пути через МОД с
учетом кусков"""
    n = len(matrix)
    if n < 2:  # если меньше 2 вершин, мод не существует
        return 0

    # определяем куски из текущего маршрута
    chunks = find_chunks(route, matrix)
    print(f"[PRIM_MST] Найдены куски: {chunks}")

    # если все вершины уже соединены в один кусок
    if len(chunks) == 1:
        return 0

    # строим матрицу допустимых ребер между кусками
    chunk_matrix = build_chunk_matrix(matrix, chunks)
    print("[PRIM_MST] Матрица допустимых ребер между кусками:")
    print_matrix(chunk_matrix)

    # применяем алгоритм Прима к этой матрице
    return standard_prim(chunk_matrix)


def find_chunks(route, matrix):
    """Находит все куски (связные компоненты) в текущем маршруте"""
    if not route:
        return []

    # собираем все ребра в словарь {from: to}
    edges = {src: dst for src, dst in route}

    visited = set()
    chunks = []

    # находим начальные точки кусков (те, которые не являются конечными в
других ребрах)
    starts = set(edges.keys()) - set(edges.values())
```

```
    # если все вершины в цикле, берем первую
    if not starts:
        starts = {route[0][0]}

    for start in starts:
        if start in visited:
            continue

        chunk = []
        current = start
        while current in edges:
            chunk.append(current)
            visited.add(current)
            current = edges[current]

        if current not in visited:
            chunk.append(current)
            visited.add(current)

        chunks.append(chunk)

    # добавляем изолированные вершины
    all_vertices = set(range(len(matrix)))
    isolated = all_vertices - visited
    for v in isolated:
        chunks.append([v])

    print("chunks", chunks)
    return chunks


def build_chunk_matrix(matrix, chunks):
    """Строит матрицу допустимых ребер между кусками"""
    n_chunks = len(chunks)
    chunk_matrix = [[math.inf] * n_chunks for _ in range(n_chunks)]

    for i in range(n_chunks):
        for j in range(n_chunks):
            if i == j:
                continue
            # находим минимальное ребро между последней вершиной i-го куска
            # и первой вершиной j-го куска
            last_in_i = chunks[i][-1]
            first_in_j = chunks[j][0]

            chunk_matrix[i][j] = matrix[last_in_i][first_in_j]

    return chunk_matrix


def standard_prim(chunk_matrix):
    """Стандартная реализация алгоритма Прима для полученной матрицы
кусков"""
    n = len(chunk_matrix)
    if n < 2:
        return 0

    used = [False] * n
    mst_weight = 0
    heap = []
    heapq.heappush(heap, (0, 0))  # (weight, chunk_index)

    while heap:
```

```
        weight, v = heapq.heappop(heap)

        if used[v]:
            continue

        used[v] = True
        mst_weight += weight

        for u in range(n):
            if not used[u] and chunk_matrix[v][u] != math.inf:
                heapq.heappush(heap, (chunk_matrix[v][u], u))

    return mst_weight


class Node:
    """Класс для представления узла в дереве поиска алгоритма ветвей и
границ"""

    def __init__(self, matrix, bound, route):
        self.matrix = matrix
        self.bound = bound
        self.route = route

    def get_cell_with_max_penalty(self):
        """
        Находит клетку с максимальным штрафом за ее исключение
        (клетку с 0, у которой сумма минимальных элементов в строке и
столбце максимальна)
         """
        print("[CELL WITH MAX PENALTY] Ищем клетку с максимальным штрафом
за ее исключение")
        max_penalty = -math.inf  # изначально штраф минимальный
        cell_with_max_penalty = None
        n = len(self.matrix)

        for row in range(n):
            for col in range(n):
                if self.matrix[row][col] == 0:  # если клетка содержит 0
(после редукции)
                    # находим минимальный элемент в строке (исключая
текущий столбец)
                    row_min = math.inf
                    for i in range(n):
                        if i == col or not
math.isfinite(self.matrix[row][i]):
                            continue
                        if self.matrix[row][i] < row_min:
                            row_min = self.matrix[row][i]
                    print(f"[CELL WITH MAX PENALTY] Нашли минимальный
элемент в строке {row} = {row_min}")

                    col_min = math.inf  # находим минимальный элемент в
столбце (исключая текущую строку)
                    for i in range(n):
                        if i == row or not
math.isfinite(self.matrix[i][col]):
                            continue
                        if self.matrix[i][col] < col_min:
                            col_min = self.matrix[i][col]
                    print(f"[CELL WITH MAX PENALTY] Нашли минимальный
элемент в столбце {col} = ", col_min)
                    penalty = row_min + col_min  # штраф за исключение этой
клетки
```

```
                    print(f"[CELL WITH MAX PENALTY] Штраф за исключение
этой клетки [{row},{col}] = ", penalty)
                    if penalty > max_penalty:
                        max_penalty = penalty
                        cell_with_max_penalty = (row, col, max_penalty)
                        print(f"\n[CELL WITH MAX PENALTY] Текущий штраф
[{row},{col}] - максимальный = ", max_penalty)

        return cell_with_max_penalty

    def make_children(self):
        """Создает двух потомков для узла (ветвление)"""
        print("[MAKE_CHILDREN] Создание двух потомков для узла")
        cell = self.get_cell_with_max_penalty()  # находим клетку для
ветвления
        if cell is None:  # все клетки обработаны
            return None, None

        row, col, left_penalty = cell
        print(f"[MAKE_CHILDREN] Нашли клетку [{row}, {col}] с максимальным
штрафом = {left_penalty}")

        # левый потомок - исключаем текущее ребро
        left_matrix = clone_matrix(self.matrix)  # копируем матрицу
        print("[MAKE_CHILDREN] Копируем матрицу в левого потомка")
        print("[MAKE CHILDREN] Матрица до запрещения:")
        print_matrix(left_matrix)
        left_matrix[row][col] = math.inf  # запрещаем текущее ребро
        print("[MAKE CHILDREN] Матрица после запрещения:")
        print_matrix(left_matrix)
        print(f"[MAKE_CHILDREN] Запрещаем ребро текущее
left_matrix[{row}][{col}] : {left_matrix[row][col]}")
        left_bound = self.bound + left_penalty  # обновляем нижнюю границу
        print(f"[MAKE_CHILDREN] Обновляем нижнюю границу для левого
потомка: {left_bound}")
        left_route = self.route.copy()  # копируем маршрут
        print(f"[MAKE CHILDREN] Копируем маршрут: {left_route}")
        left_child = Node(left_matrix, left_bound, left_route)  # создаем
левого потомка
        print(f"[MAKE CHILDREN] Создали левого потомка с обновленной
матрицей, нижней границей и маршрутом")

        # правый потомок - включаем текущее ребро
        right_matrix = clone_matrix(self.matrix)  # копируем матрицу
        print("[MAKE CHILDREN] Матрица до запрещения:")
        print_matrix(right_matrix)
        print(f"\n[MAKE CHILDREN] Копируем матрицу для правого потомка")
        right_matrix[col][row] = math.inf  # запрещаем обратное ребро
        print(f"[MAKE CHILDREN] Запрещаем обратное ребро
right_matrix[{col}][{row}] : {right_matrix[col][row]}")

        print("[MAKE CHILDREN] Матрица после запрещения:")
        print_matrix(right_matrix)

        print(f"[MAKE CHILDREN] Запрещаем все ребра из строки {row} и
столбца {col} у правого потомка")
        # запрещаем все ребра из текущей строки и столбца
        for i in range(len(right_matrix)):
            right_matrix[row][i] = math.inf
            right_matrix[i][col] = math.inf

        print("[MAKE CHILDREN] Матрица после запрещения:")
        print_matrix(right_matrix)
```

```
        right_route = self.route.copy()  # копируем маршрут
        print(f"[MAKE CHILDREN] Копируем текущий маршрут в правого
потомка : {right_route}")
        right_route.append((row, col))  # включаем текущее ребро в маршрут
        print(f"[MAKE CHILDREN] Включаем текущее ребро в маршрут :
{right_route}")

        print(f"\n[MAKE CHILDREN] Ищем все подциклы")
        close_edges = get_close_edges(right_route)  # ищем все подциклы
        for (r, c) in close_edges:
            print(f"[MAKE CHILDREN] Запрещаем ребро ({r},{c}), которое
создает подцикл")
            right_matrix[r][c] = math.inf  # запрещаем все ребра, которые
создают подциклы

        print("[MAKE CHILDREN] Матрица после запрещения подциклов:")
        print_matrix(right_matrix)

        print("[MAKE CHILDREN] Редуцируем матрицу")
        right_penalty = reduce(right_matrix, right_route)  # редуцируем
матрицу
        print("[MAKE CHILDREN] Штраф у матрицы правого потомка после
редукции: ", right_penalty)
        right_bound = self.bound + right_penalty  # обновляем нижнюю
границу
        print("[MAKE CHILDREN] Обновленная нижняя граница у правого: ",
right_bound)
        right_child = Node(right_matrix, right_bound, right_route)  #
создаем правого потомка
        print("[MAKE CHILDREN] Создан правый потомок")
        return left_child, right_child


def clone_matrix(matrix):
    return [row[:] for row in matrix]


def row_mins(matrix):
    print(f"[ROW MIN] Находим минимумы в строках матрицы")
    mins = []
    for row in matrix:
        min_val = math.inf
        for val in row:
            if val < min_val:
                min_val = val
                print(f"[ROW MIN] Найден минимум в строке {row}:
{min_val}")
        mins.append(min_val if min_val != math.inf else 0)
    return mins


def column_mins(matrix):
    mins = []
    print(f"[COLUMN MIN] Находим минимумы в столбцах матрицы")

    n = len(matrix)
    for col in range(n):
        min_val = math.inf
        for row in range(n):
            if matrix[row][col] < min_val:
                min_val = matrix[row][col]
                print(f"[COLUMN MIN] Найден минимум в столбце {col}:
{min_val}")
```

```
        mins.append(min_val if min_val != math.inf else 0)
    return mins


def sum_finites(arr):
    return sum(val for val in arr if val != math.inf)


def reduce_rows(matrix, mins):
    print(f"[REDUCE ROWS] Редуцируем по строкам")
    print("[REDUCE ROWS] Матрица до редуцирования")
    print_matrix(matrix)
    n = len(matrix)
    for row in range(n):
        for col in range(n):
            if mins[row] != math.inf:
                matrix[row][col] -= mins[row]

    print("[REDUCE ROWS] Матрица после редуцирования:")
    print_matrix(matrix)


def reduce_columns(matrix, mins):
    print(f"[REDUCE COLUMNS] Редуцируем по столбцам")
    print("[REDUCE COLUMNS] Матрица до редуцирования")
    print_matrix(matrix)
    n = len(matrix)
    for col in range(n):
        for row in range(n):
            if mins[col] != math.inf:
                matrix[row][col] -= mins[col]

    print("[REDUCE COLUMNS] Матрица после редуцирования:")
    print_matrix(matrix)


def reduce(matrix, route):
    """Полная редукция матрицы с добавлением оценки МОД"""
    print(f"[REDUCE] Редукция матрицы с добавлением оценки МОД")
    row_m = row_mins(matrix)
    reduce_rows(matrix, row_m)
    column_m = column_mins(matrix)
    reduce_columns(matrix, column_m)

    print(f"[REDUCE] Сумма минимумов по строкам = {sum_finites(row_m)}, по
столбцам = {sum_finites(column_m)}")
    reduction_cost = sum_finites(row_m) + sum_finites(column_m)
    print(f"[REDUCE] Общая стоимость после редуцирования =
{reduction_cost}")
    print(f"[REDUCE] Находим нижнюю оценку суммарного веса остатка пути на
основе МОД")
    mst_estimate = prim_mst(matrix, route)

    print(f"[REDUCE] Найденная оценка по МОД: {mst_estimate}")
    print(f"[REDUCE] Итого: {reduction_cost + mst_estimate}")
    return reduction_cost + mst_estimate


def find_next_start_city(edges, start_city):
    """Находит индекс ребра, которое ведет в заданный город"""
    print(f"[FIND NEXT START CITY] Находим индекс ребра, которое ведет в
город {start_city}")
    for i, (_, dst) in enumerate(edges):  # для каждого ребра в маршруте
        if dst == start_city:  # если город назначения совпадает с искомым
```

```
            print(f"[FIND NEXT START CITY] Найдено ребро {i} {edges[i]},
которое ведет в город {start_city}")
            return i
    print(f"[FIND NEXT START CITY] Не найдено ребро, которое ведет в город
{start_city}")

    return -1


def find_next_end_city(edges, end_city):
    """Находит индекс ребра, которое начинается в заданном городе"""
    print(f"[FIND NEXT END CITY] Находим индекс ребра, которое начинается в
городе {end_city}")
    for i, (src, _) in enumerate(edges):  # для каждого ребра в маршруте
        if src == end_city:  # если город отправления совпадает с искомым
            print(f"[FIND NEXT END CITY] Найдено ребро {i} {edges[i]},
которое начинается в {end_city}")
            return i
    print(f"[FIND NEXT END CITY] Не найдено ребро, которое начинается в
{end_city}")
    return -1


def get_close_edges(route):
    """Находит ребра, которые могут образовывать подциклы"""
    print(f"\n[GET CLOSE EDGES] ИЩЕМ ВСЕ РЕБРА, КОТОРЫЕ МОГУТ ОБРАЗОВЫВАТЬ
ПОДЦИКЛЫ")
    result = []
    edges = route.copy()
    print(f"[GET CLOSE EDGES] Начальные ребра для анализа: {edges}")

    while edges:
        print(f"\n[GET CLOSE EDGES] Осталось ребер для обработки:
{len(edges)}")
        length = 1  # длина текущей цепочки
        start_city, end_city = edges.pop(0)  # берем первое ребро
        print(f"[GET CLOSE EDGES] Текущее ребро: ({start_city},
{end_city})")
        print(f"[GET CLOSE EDGES] Начинаем построение цепочки с этого
ребра")

        # ищем ребра, которые продолжают цепочку в начало
        print("\n[GET CLOSE EDGES] Поиск ребер, продолжающих цепочку В
НАЧАЛО:")
        index = find_next_start_city(edges, start_city)
        while index != -1:
            old_start = start_city
            start_city, _ = edges.pop(index)
            length += 1
            print(f"[GET CLOSE EDGES] Найдено продолжающее ребро:
({start_city}, {old_start})")
            print(f"[GET CLOSE EDGES] Обновленная цепочка: ({start_city}
-> ... -> {end_city})")
            print(f"[GET CLOSE EDGES] Текущая длина цепочки: {length}")
            index = find_next_start_city(edges, start_city)  # ищем
следующее, которое продолжает

        # ищем ребра, которые продолжают цепочку в конец
        print("\n[GET CLOSE EDGES] Поиск ребер, продолжающих цепочку В
КОНЕЦ:")
        index = find_next_end_city(edges, end_city)
        while index != -1:
            old_end = end_city
            _, end_city = edges.pop(index)
```

```
            length += 1
            print(f"[GET CLOSE EDGES] Найдено продолжающее ребро:
({old_end}, {end_city})")
            print(f"[GET CLOSE EDGES] Обновленная цепочка: ({start_city}
-> ... -> {end_city})")
            print(f"[GET CLOSE EDGES] Текущая длина цепочки: {length}")
            index = find_next_end_city(edges, end_city)

        print(f"\n[GET CLOSE EDGES] Итоговая длина цепочки: {length}")

        if length >= 2:  # если цепочка может образовать цикл
            result.append((end_city, start_city))  # добавляем в
запрещенные ребра
            print(f"[GET CLOSE EDGES] Обнаружена потенциальная
цикличность!")
            print(f"[GET CLOSE EDGES] Добавляем запретное ребро для
предотвращения цикла: ({end_city}, {start_city})")
        else:
            print("[GET CLOSE EDGES] Цепочка слишком коротка для
образования цикла - пропускаем")
    print(f"\n[GET CLOSE EDGES] Итоговый список запретных ребер: {result}")
    return result


def little(matrix):
    """Алгоритм Литтла"""
    node = Node(matrix, 0, [])
    root_matrix = clone_matrix(matrix)
    min_bound = reduce(root_matrix, [])
    root = Node(root_matrix, min_bound, [])

    print(f"\nСоздаем корень дерева с матрицей {root_matrix} и мин.границей
{min_bound}")

    priority_queue = []  # Используем кучу для хранения узлов по приоритету
(нижней границе)
    heapq.heappush(priority_queue, (root.bound, id(root), root))  # (bound,
id, node)
    record = None  # лучший найденный маршрут
    print(f"Добавляем корень в очередь с приоритетом (граница, номер корня,
объект корня): {priority_queue}")
    while priority_queue:  # пока есть узлы
        print("\n\nПытаемся извлечь из очереди узел с минимальной нижней
границей")
        try:
            mn, _, min_node = heapq.heappop(priority_queue)  # Извлекаем
узел с минимальной границей
            print(f"Извлекли узел с минимальной границей {mn}")
        except IndexError:
            print("Очередь пуста, заверщаем работу")
            break  # если очередь пуста, завершаем работу

        print(f"\nТекущий путь: {min_node.route}, текущая нижняя граница:
{min_node.bound}")

        if record and record['length'] <= min_node.bound:  # если есть
запись о меньшем маршруте, завершаем
            print("Найдена запись о меньшем маршруте, завершаем просмотр
очереди")
            break

        if len(min_node.route) == len(matrix) - 2:  # если маршрут почти
полный
            print("\nМаршрут почти полный (не хватает двух городов)")
```

```
            print("Добавляем последние ребра для завершения просмотра")
            for row in range(len(matrix)):  # добавляем последние ребра для
завершения цикла
                for col in range(len(matrix)):
                    if math.isfinite(min_node.matrix[row][col]):
                        min_node.bound += min_node.matrix[row][col]
                        min_node.route.append((row, col))
                        print("Обновленная нижняя граница: ",
min_node.bound)
                        print("Обновленный путь для вершины с мин.границей:
", min_node.route)

            if record is None or record['length'] > min_node.bound:  #
обновляем запись, если нашли лучшее решение
                print("Было найдено лучшее решение, обновляем запись: ")
                record = {'length': min_node.bound, 'route':
min_node.route}
                print(record)
        else:
            print("\n Находим правого (включает ребро) и левого потомка
(исключает ребро) для текущей вершины")
            left_child, right_child = min_node.make_children()

            # добавляем потомков в очередь, если они существуют
            if left_child is not None:
                print(f"Левый потомок найден, добавляем его в очередь")
                heapq.heappush(priority_queue, (left_child.bound,
id(left_child), left_child))
            if right_child is not None:
                print(f"Правый потомок найден, добавляем его в очередь")
                heapq.heappush(priority_queue, (right_child.bound,
id(right_child), right_child))

    print(record)
    return record


def nearest_neighbor(matrix, st_city=0):
    """Алгоритм ближайшего соседа для ЗК"""
    n = len(matrix)
    if n < 2:
        return []
    print("Входная матрица:")
    print_matrix(matrix)
    route = [st_city]
    print(f"Текущий маршрут: {[x + 1 for x in route]}")
    unvisited = set(range(n)) - {st_city}  # множество непосещенных городов
    print(f"Непосещенные города: {[x + 1 for x in unvisited]}")
    total_distance = 0

    current_city = st_city

    while unvisited:
        print(f"\nТекущий город {current_city + 1}")
        # находим ближайшего непосещенного соседа
        nearest_city = None
        min_distance = math.inf

        for city in unvisited:
            distance = matrix[current_city][city]
            print(f"Дистанция до города {city + 1} от текущего города
{current_city + 1} = {distance}")
            if distance != math.inf and distance < min_distance:
```

```
                print(f"Найденная дистанция на данный момент минимальная,
текущий ближайший город - {city + 1}")
                min_distance = distance
                nearest_city = city
            else:
                print("Найденная дистанция на данный момент не минимальная,
не добавляем ее")

        # если нет доступных городов
        if nearest_city is None:
            print(f"Доступных ближайших городов не нашлось. Маршрута не
существует.")
            return None

        # добавляем город в маршрут
        route.append(nearest_city)
        print(f"Добавляем найденный город в маршрут : {[x + 1 for x in
route]}")
        unvisited.remove(nearest_city)  # удаляем из непосещенных
        print(f"Непосещенные города: {[x + 1 for x in unvisited]}")
        total_distance += min_distance  # добавляем стоимость
        print(f"Текущая стоимость {total_distance}")
        current_city = nearest_city  # текущий - теперь ближайший

    # после прохода по каждому городу, возвращаемся в начальный город
    return_distance = matrix[current_city][st_city]
    print(f"Стоимость возврата в город, из которого отправились:
{return_distance}")
    if return_distance == math.inf:  # нет обратного пути
        print("Обратного пути нет")
        return None

    total_distance += return_distance
    route.append(st_city)
    print(f"Добавляем стартовый город в путь: {[x + 1 for x in route]}")
    return {
        'route': route,
        'distance': total_distance
    }


def main():
    cur_matrix = None
    n = 2

    while True:
        print('\n')
        print("=" * 150)
        print("Команды:")
        print("1. Сгенерировать случайную матрицу")
        print("2. Загрузить матрицу из файла")
        print("3. Сохранить матрицу в файл")
        print("4. Решить задачу методом ближайшего соседа")
        print("5. Решить задачу методом ветвей и границ (Литтла)")
        print("6. Показать текущую матрицу")
        print("7. Ввести матрицу вручную")
        print("8. Выход")

        c = input("Выберите действие (1-8): ")
        if c == "1":
            n = int(input("Введите количество городов N: "))
            sym = input("Симметричная матрица? (y/n): ").lower() == 'y'
            cur_matrix = generate_matrix(n, sym)
            print("\nСгенерированная матрица:")
```

```
            print_matrix(cur_matrix)

        elif c == "2":
            filename = input("Введите имя файла: ")
            loaded_matrix = load_matrix_from_file(filename)
            n = len(loaded_matrix)
            if loaded_matrix is not None:
                cur_matrix = loaded_matrix
                print("\nЗагруженная матрица:")
                print_matrix([[math.inf if x == -1 else x for x in row] for
row in cur_matrix])

        elif c == "3":
            if cur_matrix is None:
                print("Нет матрицы для сохранения")
                continue
            filename = input("Введите имя файла для сохранения: ")
            save_matrix_to_file(cur_matrix, filename)
            print(f"Матрица сохранена в файл '{filename}'")
        elif c == "7":
            n = int(input("Введите количество городов N: "))
            cur_matrix = []
            print(f"\nВведите матрицу {n}x{n}:")
            cost_matrix = []
            for _ in range(n):
                row = list(map(int, input().split()))
                cost_matrix.append(row)

            for i in range(n):
                matrix_row = []
                for j in range(n):
                    if cost_matrix[i][j] == -1:
                        matrix_row.append(math.inf)
                    else:
                        matrix_row.append(cost_matrix[i][j])
                cur_matrix.append(matrix_row)
            print("\nВведенная матрица:")
            print_matrix(cur_matrix)
        elif c == "4":
            if cur_matrix is None:
                print("Сначала загрузите или создайте матрицу")
                continue
            n = len(cur_matrix)
            while True:
                try:
                    start_city = int(input(f"Введите стартовый город (1-
{len(cur_matrix)}): "))
                    if 1 <= start_city <= len(cur_matrix):
                        break
                    print("Номер города вне допустимого диапазона")
                except ValueError:
                    print("Введите целое число")

            result = nearest_neighbor(cur_matrix, start_city - 1)

            print("\n" + "=" * 50)
            print("РЕЗУЛЬТАТ:")
            if result:
                print(f"Оптимальный маршрут: {' → '.join(map(str, [x + 1
for x in result['route']]))}")
                print(f"Общая длина пути: {result['distance']}")
            else:
                print("Невозможно построить маршрут для данной матрицы")
```

```
        elif c == "5":
            if cur_matrix is not None:
                result = little(cur_matrix)
                n = len(cur_matrix)
                print("\nРезультат алгоритма Литтла:")
                if result:
                    route = [0] * n
                    cur_city = 0
                    route[0] = 0

                    next_city = {}
                    for (src, dst) in result['route']:
                        next_city[src] = dst

                    for i in range(1, n):
                        route[i] = next_city.get(cur_city, 0)
                        cur_city = route[i]

                    print(' '.join(map(str, route)))
                    print(f"{result['length']:.1f}")
                else:
                    print("Матрица не введена")

        elif c == "6":
            if cur_matrix is not None:
                print_matrix(cur_matrix)
            else:
                print("Матрица не существует")

        elif c == "8":
            print("Выход")
            exit(0)

        else:
            print("Некорректная команда")


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