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

ОТЧЕТ

по лабораторной работе № 2
по дисциплине «Построение и Анализ Алгоритмов»
Тема: «Кратчайшие пути в графе: коммивояжёр»
Вариант 2

Студент гр. 3343 Жучков О.Д.

Преподаватель _____ Жангиров Т.Р.
_____

Санкт-Петербург
2025

Цель работы

Целью работы является изучение задачи коммивояжёра, точных и приближенных методов её решения и реализация алгоритма Литтла с модификацией и алгоритма ближайшего соседа.

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

Вариант 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.

Выполнение работы

Алгоритм Литтла работает так: он начинает с матрицы стоимостей путей между городами, выполняет её редукцию для получения нижней границы стоимости решения. Затем выбирается ребро (путь между городами) с максимальным штрафом за неиспользование. Создаются две ветви: одна включает это ребро в маршрут, другая - исключает. Для каждой ветви матрица стоимостей корректируется (запрещаются петли, исключаются ребра), редуцируется, и вычисляется нижняя граница стоимости решения. Ветви с наименьшей нижней границей продолжают исследоваться, пока не будет найден полный маршрут, который становится текущим лучшим решением. Другие ветви отсекаются, если их нижняя граница превышает стоимость текущего лучшего решения.

Функция редукции матрицы проходит по матрице, вычитая минимальный элемент из каждой строки и столбца. Функция подсчитывает и возвращает общую сумму вычтенных минимумов, а редукция уменьшает значения матрицы, не меняя решения задачи, при этом помогает находить более оптимальное решение.

Для вычисления максимального штрафа за отказ от ребра используется метод, который находит ячейку с нулевым значением в матрице, которая имеет максимальный "штраф". Штраф для ячейки вычисляется как сумма минимального элемента в строке (исключая элемент в столбце) и минимального элемента в столбце (исключая элемент в строке). Эта функция нужна для выбора ветвления в алгоритме ветвей и границ: ячейка с максимальным штрафом указывает на наиболее перспективное направление для дальнейшего поиска решения.

Кроме того, к нижней оценке веса решения добавляется нижняя оценка суммарного веса остатка пути на основе МОД. МОД строится от стянутых вершин, то есть если в текущем решении есть путь соединяющий две и более вершин, они стягиваются в одну.

Оценка сложности алгоритма.

По времени: Поиск нулевого элемента с наибольшим “штрафом” – $O(n^2)$, редукция матрицы – $O(n^2)$; в худшем случае происходит проход по всему бинарному дереву, количество элементов в нем равно 2^n-1. Так, примерная сложность алгоритма – $O(n^2 * 2^n)$, по памяти аналогично $O(n^2*2^n)$, для каждой вершины нужно хранить матрицу n*n.

Также реализован алгоритм АБС (Аппроксимация Ближайшего Соседа) — это приближённый алгоритм решения задачи коммивояжера. Он основан на жадном подходе, который позволяет найти решение задачи с ограниченной точностью за приемлемое время. Этот жадный алгоритм начинает с произвольного города и поочередно выбирает следующий город, который является ближайшим к текущему, пока не будут посещены все города. Выполняется всего за $O(n^2)$ и дает приближение не более чем в 2 раза больше стоимости оптимального пути.

Тестирование

Таблица 1 – Тестирование алгоритма

№ п/п	Входные данные	Выходные данные	Комментарии
1	5 -1 32 25 67 24 5 -1 39 84 8 52 33 -1 5 37 96 51 77 -1 26 48 77 84 51 -1	1) 0 2 3 1 4 137.0 2) 0 4 3 1 2 217.0	Проверка на произвольной матрице полного графа (1 – алг. Литтла, 2 – АБС)
2	5 -1 34 22 41 75 34 -1 99 98 18 22 99 -1 79 30 41 98 79 -1 94 75 18 30 94 -1	1) 0 3 2 4 1 202.0 2) 0 2 4 1 3 209.0	Проверка на симметричной матрице

3	8 -1 14 61 63 63 66 72 60 11 -1 63 76 35 2 37 83 56 37 -1 55 24 35 81 35 38 52 66 -1 68 77 100 68 80 35 69 19 -1 91 65 42 85 63 51 93 13 -1 87 12 59 27 44 57 16 68 -1 24 70 78 49 58 8 94 45 -1	1) 0 1 5 7 6 2 4 3 198.0 2) 0 1 5 7 4 3 2 6 261.0	Проверка на матрице большего размера

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
from math import inf, isinf
from random import randint


def mst(matrix, route):
    n = len(matrix)
    if n < 2:
        return 0
    print("Оценок остатка пути через МОД")
    chunks = find_chunks(route, matrix)
    print(f"\tНайдены куски {chunks}")
    if len(chunks) == 1:
        return 0
    chunk_matrix = build_chunk_matrix(matrix, chunks)
    print("\tДопустимые ребра между кусками:")
    print_matrix(chunk_matrix)
    return mst_prim(chunk_matrix)


def find_chunks(route, matrix):
    if not route:
        return []
    edges = {source: dest for source, dest in route}
    visited = set()
    chunks = []
    starts = set(edges.keys()) - set(edges.values())
    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)

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


def build_chunk_matrix(matrix, chunks):
    n_chunks = len(chunks)
    chunk_matrix = [[inf] * n_chunks for _ in range(n_chunks)]
```

```
        for i in range(n_chunks):
            for j in range(n_chunks):
                if i == j:
                    continue
                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 mst_prim(chunk_matrix):
        n = len(chunk_matrix)
        if n < 2:
            return 0

        used = [False] * n
        mst_weight = 0
        heap = []
        heapq.heappush(heap, (0, 0))
        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] != inf:
                    heapq.heappush(heap, (chunk_matrix[v][u], u))
        return mst_weight


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

        def get_priority_cell(self):
            print("Ищем клетку с максимальным штрафом за ее исключение")
            max_penalty = -inf
            max_cell = None
            n = len(self.matrix)
            for row in range(n):
                for col in range(n):
                    if self.matrix[row][col] == 0:
                        row_min = inf
                        for i in range(n):
                            if i == col or isinf(self.matrix[row][i]):
                                continue
                            if self.matrix[row][i] < row_min:
                                row_min = self.matrix[row][i]
                        print(f"\tНашли минимальный элемент в строке
{row} = {row_min}")
                        col_min = inf
                        for i in range(n):
                            if i == row or isinf(self.matrix[i][col]):
                                continue
                            if self.matrix[i][col] < col_min:
```

```
                                col_min = self.matrix[i][col]
                        print(f"\tНашли минимальный элемент в столбце
{col} = {col_min}")
                        penalty = row_min + col_min
                        print(f"\tШтраф за исключение этой клетки
[{row},{col}] = {penalty}")
                        if penalty > max_penalty:
                            max_penalty = penalty
                            max_cell = (row, col, max_penalty)
                            print(f"Текущий штраф [{row},{col}] - макси-
мальный = {max_penalty}")
            return max_cell

        def make_children(self):
            print("Создание двух потомков для узла")
            cell = self.get_priority_cell()
            if cell is None:
                return None, None
            row, col, left_penalty = cell
            print(f"Нашли клетку [{row}, {col}] с максимальным штрафом =
{left_penalty}")

            left_matrix = clone_matrix(self.matrix)
            print("Создаем левого потомка")
            print("Матрица до запрещения:")
            print_matrix(left_matrix)
            left_matrix[row][col] = inf
            print("Матрица после запрещения:")
            print_matrix(left_matrix)
            print(f"Запрещаем ребро текущее left_matrix[{row}][{col}] :
{left_matrix[row][col]}")
            left_bound = self.bound + left_penalty
            print(f"Обновляем нижнюю границу для левого потомка:
{left_bound}")
            left_route = self.route.copy()
            print(f"Копируем маршрут: {left_route}")
            left_child = Node(left_matrix, left_bound, left_route)

            right_matrix = clone_matrix(self.matrix)
            print(f"Созадем правого потомка")
            print("Матрица до запрещения:")
            print_matrix(right_matrix)
            right_matrix[col][row] = inf
            print(f"Запрещаем обратное ребро right_matrix[{col}][{row}] :
{right_matrix[col][row]}")
            print("Матрица после запрещения:")
            print_matrix(right_matrix)
            print(f"Запрещаем все ребра из строки {row} и столбца {col}
у правого потомка")
            for i in range(len(right_matrix)):
                right_matrix[row][i] = inf
                right_matrix[i][col] = inf
            print("Матрица после запрещения:")
            print_matrix(right_matrix)
            right_route = self.route.copy()
            print(f"Копируем маршрут {right_route}")
            right_route.append((row, col))
            print(f"Включаем текущее ребро в маршрут : {right_route}")
```

```
            print(f"Ищем все подциклы")
            close_edges = get_close_edges(right_route)
            for (r, c) in close_edges:
                print(f"\tЗапрещаем ребро ({r},{c}), которое создает
подцикл")
                right_matrix[r][c] = inf

            print("Матрица после запрещения подциклов:")
            print_matrix(right_matrix)

            print("Редуцируем матрицу")
            right_penalty = reduce(right_matrix, right_route)
            print("Штраф у матрицы правого потомка после редукции: ",
right_penalty)
            right_bound = self.bound + right_penalty
            print("Обновленная нижняя граница у правого: ", right_bound)
            right_child = Node(right_matrix, right_bound, right_route)
            print("Создан правый потомок")
            return left_child, right_child


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


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


    def column_mins(matrix):
        mins = []
        print(f"Находим минимумы в столбцах матрицы")
        n = len(matrix)
        for col in range(n):
            min_val = inf
            for row in range(n):
                if matrix[row][col] < min_val:
                    min_val = matrix[row][col]
                    print(f"\tНайден минимум в столбце {col}: {min_val}")

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


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

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

        print("Матрица после редуцирования:")
        print_matrix(matrix)


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

        print("Матрица после редуцирования:")
        print_matrix(matrix)


    def reduce(matrix, route):
        print("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"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"Общая стоимость после редуцирования = {reduction_cost}")
        print(f"Находим нижнюю оценку суммарного веса остатка пути на
основе МОД")
        mst_estimate = mst(matrix, route)

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


    def find_next_start_city(edges, start_city):
        print(f"Находим индекс ребра, которое ведет в город {start_city}")
        for i, (_, dst) in enumerate(edges):
            if dst == start_city:
                print(f"Найдено ребро {i} {edges[i]}, которое ведет в
город {start_city}")
                return i
        print(f"Не найдено ребро, которое ведет в город {start_city}")
        return -1
```

```
    def find_next_end_city(edges, end_city):
        print(f"Находим индекс ребра, которое начинается в городе
{end_city}")
        for i, (src, _) in enumerate(edges):
            if src == end_city:
                print(f"Найдено ребро {i} {edges[i]}, которое начинается
в {end_city}")
                return i
        print(f"Не найдено ребро, которое начинается в {end_city}")
        return -1


    def get_close_edges(route):
        result = []
        edges = route.copy()
        print(f"Начальные ребра для анализа: {edges}")

        while edges:
            length = 1
            start_city, end_city = edges.pop(0)
            print(f"Текущее ребро: ({start_city}, {end_city})")
            index = find_next_start_city(edges, start_city)
            while index != -1:
                old_start = start_city
                start_city, _ = edges.pop(index)
                length += 1
                print(f"Найдено продолжающее ребро: ({start_city},
{old_start})")
                print(f"Обновленная цепочка: ({start_city} -> ... ->
{end_city})")
                print(f"Текущая длина цепочки: {length}")
                index = find_next_start_city(edges, start_city)
            index = find_next_end_city(edges, end_city)
            while index != -1:
                old_end = end_city
                _, end_city = edges.pop(index)
                length += 1
                print(f"Найдено продолжающее ребро: ({old_end},
{end_city})")
                print(f"Обновленная цепочка: ({start_city} -> ... ->
{end_city})")
                print(f"Текущая длина цепочки: {length}")
                index = find_next_end_city(edges, end_city)

            print(f"Итоговая длина цепочки: {length}")

            if length >= 2:
                result.append((end_city, start_city))
        print(f"Итоговый список запретных ребер: {result}")
        return result


    def little_alg(matrix):
        node = Node(matrix, 0, [])
        root_matrix = clone_matrix(matrix)
        min_bound = reduce(root_matrix, [])
        root = Node(root_matrix, min_bound, [])
```

```
        print(f"Создаем корень дерева с матрицей {root_matrix} и мин.гра-
ницей {min_bound}")
        priority_queue = []
        heapq.heappush(priority_queue, (root.bound, id(root), root))
        record = None
        while priority_queue:
            mn, _, min_node = heapq.heappop(priority_queue)
            print(f"Извлекли узел с минимальной границей {mn}")
            print(f"Текущий путь: {min_node.route}, текущая нижняя гра-
ница: {min_node.bound}")
            if record and record['length'] <= min_node.bound:
                print("Найдена запись о меньшем маршруте, завершаем про-
смотр очереди")
                break
            if len(min_node.route) == len(matrix) - 2:
                print("Добавляем последние ребра для завершения про-
смотра")
                for row in range(len(matrix)):
                    for col in range(len(matrix)):
                        if not isinf(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:
                    print("Было найдено лучшее решение, обновляем запись:
")
                    record = {'length': min_node.bound, 'route':
min_node.route}
                    print(record)
            else:
                print("Находим правого (включает ребро) и левого потомка
(исключает ребро) для текущей вершины")
                left_child, right_child = min_node.make_children()
                if left_child is not None:
                    print(f"Левый потомок найден, добавляем его в оче-
редь")
                    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 оче-
редь")
                    heapq.heappush(priority_queue, (right_child.bound,
id(right_child), right_child))
        print(record)
        return record


    def nearest_neighbor(matrix, verbose=False):
        print("Аппроксимация ближайшего соседа")
        num_cities = len(matrix)
        visited = {0}
        path = [0]
        total_cost = 0
        current_city = 0
```

```
        while len(visited) < num_cities:
            next_city = min((i for i in range(num_cities) if i not in
visited),
                            key=lambda i: matrix[current_city][i])
            print(f"Выбрали город {next_city} - стоимость {matrix[cur-
rent_city][next_city]}")
            if next_city is None or matrix[current_city][next_city] ==
inf:
                return {'length': inf, 'route': []}
            visited.add(next_city)
            path.append(next_city)
            total_cost += matrix[current_city][next_city]
            current_city = next_city
            if verbose:
                print(f"Текущий путь: {path}, стоимость {total_cost}")
        if matrix[current_city][0] == inf:
            return {'length': inf, 'route': []}
        path.append(0)
        total_cost += matrix[current_city][0]
        if verbose:
            print(f"Приближение пути найдено: {path}, стоимость: {to-
tal_cost}")
        return {'length': total_cost, 'route': path}


    def generate_matrix(size, symmetric=False, max_weight=100):
        matrix = [[-1] * size for _ in range(size)]
        for i in range(size):
            for j in range(size):
                if i == j:
                    matrix[i][j] = inf
                elif symmetric and j > i:
                    weight = randint(1, max_weight)
                    matrix[i][j] = weight
                    matrix[j][i] = weight
                elif not symmetric and j != i:
                    matrix[i][j] = randint(1, max_weight)
        return matrix


    def print_matrix(matrix):
        for row in matrix:
            print("\t".join(map(str, row)))


    def write_matrix_to_file(matrix, filename="matrix.txt"):
        with open(filename, 'w') as f:
            f.write(str(len(matrix)) + "\n")
            for row in matrix:
                f.write(' '.join([str(i) if not isinf(i) else "-1" for i
in row]) + '\n')


    def read_matrix_from_file(filename="matrix.txt"):
        matrix = []
        with open(filename, 'r') as f:
            n = int(f.readline().strip())
            lines = [line.strip() for line in f.readlines()]
```

```
            for line in lines:
                if line:
                    matrix.append([int(i) if i != "-1" else inf for i in
line.split()])
        for i in range(len(matrix)):
            matrix[i][i] = inf
        return matrix


    def input_matrix():
        n = int(input())
        matrix = [[int(i) if i != "-1" else inf for i in input().split()]
for row in range(n)]
        for i in range(n):
            matrix[i][i] = inf
        return matrix


    input_option = int(input("1 - ручной ввод, 2 - случайная матрица с
сохранением в файл, 3 - чтение из файла\n"))
    if input_option == 1:
        matrix = input_matrix()
    elif input_option == 2:
        opt = int(input("1 - Обычная матрица, 2 - Симметричная\n"))
        size = int(input("Размер: \n"))
        if opt == 1:
            matrix = generate_matrix(size)
            write_matrix_to_file(matrix)
        elif opt == 2:
            matrix = generate_matrix(size, True)
            write_matrix_to_file(matrix)
    elif input_option == 3:
        matrix = read_matrix_from_file()
    input_option = int(input("1 - Алгоритм Литтла, 2 - Аппроксимация
ближайшего соседа\n"))
    if input_option == 1:
        solution = little_alg([matrix[i].copy() for i in range(len(ma-
trix))])
        route = [0] * len(matrix)
        cur_city = 0
        route[0] = 0
        next_city = {}
        length = 0
        for (source, dest) in solution['route']:
            next_city[source] = dest
        for i in range(1, len(matrix)):
            length += matrix[cur_city][next_city[cur_city]]
            route[i] = next_city[cur_city]
            cur_city = route[i]
        length += matrix[cur_city][0]
        print(' '.join(map(str, route)))
        print(float(length))
        if length != solution["length"]:
            print(length, solution["length"])
    elif input_option == 2:
        solution = nearest_neighbor([matrix[i].copy() for i in
range(len(matrix))])
        print(solution)
```

```
print(' '.join(map(str, solution["route"][:-1])))
print(float(solution["length"]))
```