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: Коммивояжер. Вариант 3.

Студентка гр. 3343 _____ Ермолаева В. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Реализовать алгоритм для нахождения пути коммивояжера и его стоимости двумя методами.

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, 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.

Вариант 3. МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — поиск в глубину с учётом веса добавляемой дуги и нижней оценки веса остатка пути. Приближённый алгоритм: АМР. Замечание к варианту 3 Начинать МВиГ со стартовой вершины.

Выполнение работы

Для решения задачи был применен метод ветвей и границ с использованием эвристик. На каждом шаге выбирается ребро, добавление которого в итоге даст минимальную сумму пути. Для оценки оставшегося пути используются две эвристики:

1) Минимальное остовное дерево (МОД). Оценивает минимальную стоимость соединения оставшихся городов.

2) Полусумма легчайших рёбер. Для каждого куска вычисляется полусумма минимальных входящих и исходящих рёбер, что даёт нижнюю оценку стоимости.

Второй метод, реализованный для решения задачи - алгоритм модификации решения АМР. Он заключается в поиске решения через локальные модификации. Город переставляется в другое место, если вычисленная новая стоимость маршрута оказывается меньше текущей.

Временная сложность МВиГ в худшем случае будет экспоненциальной из-за полного перебора O(n!). Затраты по памяти будут составлять O(n) для хранения текущего маршрута. Для АМР временная сложность - $O(n^3)$, т. к. для каждой модификации перебирается n^2 вариантов, а пространственная - O(n).

Описание реализованных функций и структур:

- def generate_random_matrix(n): Генерирует квадратную матрицу стоимостей размером n x n.
- def save_matrix_to_file(matrix, filename): Сохраняет матрицу стоимостей в файл.
- load_matrix_from_file(filename): Загружает матрицу стоимостей из файла.
- calculate_total_cost(path, cost_matrix): Вычисляет общую стоимость маршрута.
- amr_algorithm(initial_path, cost_matrix): Реализует алгоритм модификации решения (АМР) для поиска оптимального пути.
- get_allowed_edges(path, remaining_cities): Возвращает список допустимых рёбер для продолжения пути.
- calculate_mst(cost_matrix, path, remaining_cities): Вычисляет минимальное остовное дерево (МОД) для оставшихся городов.
- calculate_half_sum(cost_matrix, path, remaining_cities): Вычисляет полусумму весов двух легчайших рёбер для каждого куска.
- calculate_lower_bound(path, current_cost, remaining_cities, cost_matrix): Вычисляет нижнюю оценку стоимости оставшегося пути.
- branch_and_bound(cost_matrix): Реализует алгоритм ветвей и границ для поиска оптимального пути.
- print_matrix(matrix): Выводит матрицу стоимостей на экран.
- print_path(path): Преобразует путь в строку для вывода.

Исходный код программы смотреть в приложении А.

Тестирование

Результаты тестирования представлены в таблице 1.

Табл. 1. – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	4 -1 8 1 11 1 -1 7 7 12 3 -1 6 2 15 2 -1	0 2 1 3 13.0	Результат соответствует ожиданиям.
2.	5 -1 12 14 24 87 29 -1 70 12 92 36 98 -1 6 17 12 55 47 -1 62 146 80 55 77 -1	0 2 4 1 3 135.0	Результат соответствует ожиданиям.
3.	6 -1 40 39 2 16 35 77 -1 84 2 47 59 7 19 -1 54 6 17 68 80 83 -1 90 23 11 82 88 81 -1 90 39 11 35 89 65 -1	0 1 3 5 2 4 117.0	Результат соответствует ожиданиям.

Выводы

В ходе выполнения лабораторной работы был реализован метод ветвей и границ для решения задачи коммивояжера. Для оптимизации алгоритма были применены эвристики для оценки оставшегося пути и выбора лучшего варианта. Также был написан и применен алгоритм модификации решения для улучшения найденного решения.

ПРИЛОЖЕНИЕ А

```
import random

def generate_matrix(n):
    matrix = [[-1] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i != j:
                matrix[i][j] = random.randint(1, 100)
    return matrix

def save_matrix_to_file(matrix, filename):
    with open(filename, 'w') as file:
        file.write(f"{len(matrix)}\n")
        for row in matrix:
            file.write(" ".join(map(str, row)) + "\n")

def load_matrix_from_file(filename):
    with open(filename, 'r') as file:
        n = int(file.readline())
        matrix = []
        for _ in range(n):
            row = list(map(int, file.readline().split()))
            matrix.append(row)
    return matrix

def calculate_total_cost(path, cost_matrix):
    total_cost = 0.0
    n = len(path)
    for i in range(n):
        total_cost += cost_matrix[path[i]][path[(i + 1) % n]]
    return total_cost

def amr_algorithm(cost_matrix):
    n = len(cost_matrix)
    initial_path = list(range(n))
    best_path = initial_path.copy()
    best_cost = calculate_total_cost(initial_path, cost_matrix)

    m = True
    iterations = 0
    F = n

    print(f"Начальный путь: {print_path(best_path)} со стоимостью:
{best_cost}.")

    while m and iterations < F:
        m = False

        for i in range(1, n):
            for j in range(1, n):
                new_path = best_path[:]
                new_path[i], new_path[j] = new_path[j], new_path[i]
                new_cost = calculate_total_cost(new_path, cost_matrix)

                if new_cost < best_cost:
                    print(f"\tМеняем местами города {best_path[i]} и
{best_path[j]}.")
                    print(f"Было обнаружено лучшее решение
{print_path(new_path)} со стоимостью {new_cost} (улучшение на {best_cost -
new_cost}).")
```

```
                        best_path = new_path
                        best_cost = new_cost
                        m = True

                        iterations += 1
                        break

        print(f"Все города были посещены в порядке: {print_path(best_path)}.
Стоимость найденного пути = {best_cost}.")
        return best_path, best_cost

    def get_allowed_edges(path, remaining_cities):
        allowed_edges = []
        last_city = path[-1]
        for city in remaining_cities:
            allowed_edges.append((last_city, city))
        return allowed_edges

    def calculate_mst(cost_matrix, path, remaining_cities):
        chunks = [path] + [[city] for city in remaining_cities]

        print("\tОценим оставшийся путь с помощью МОД для оставшихся кусков:")
        print(f"\t{" | ".join([", ".join(map(str, chunk)) for chunk in
chunks])}.")
        print(f"\tВсе доступные ребра:")

        edges = []
        for i in range(len(chunks)):
            for j in range(len(chunks)):
                if i != j:
                    start = chunks[i][-1]
                    end = chunks[j][0]
                    cost = cost_matrix[start][end]
                    if cost != -1:
                        print(f"\t\t{start} -> {end}, стоимость = {cost}")
                        edges.append((cost, start, end))

        edges.sort()
        parent = {city: city for chunk in chunks for city in chunk}

        def find(u):
            while parent[u] != u:
                parent[u] = parent[parent[u]]
                u = parent[u]
            return u

        mst_weight = 0
        for cost, u, v in edges:
            root_u = find(u)
            root_v = find(v)
            if root_u != root_v:
                print(f"\tДобавляем к каркасу ребро {u} -> {v} со стоимостью
{cost}.")
                mst_weight += cost
                parent[root_v] = root_u

        return mst_weight

    def calculate_half_sum(cost_matrix, path, remaining_cities):
        chunks = [path] + [[city] for city in remaining_cities]
        half_sum = 0
```

```
          print("\n\tОценим оставшийся путь с помощью полусуммы весов двух
легчайших рёбер по всем кускам:")
          print(f"\t{" | ".join([", ".join(map(str, chunk)) for chunk in
chunks])}.")

          for chunk in chunks:
              incoming_edges = []
              for other_chunk in chunks:
                  if other_chunk != chunk:
                      start = other_chunk[-1]
                      end = chunk[0]
                      cost = cost_matrix[start][end]
                      if cost != -1:
                          incoming_edges.append(cost)
              min_incoming = min(incoming_edges) if incoming_edges else 0

              outgoing_edges = []
              for other_chunk in chunks:
                  if other_chunk != chunk:
                      start = chunk[-1]
                      end = other_chunk[0]
                      cost = cost_matrix[start][end]
                      if cost != -1:
                          outgoing_edges.append(cost)
              min_outgoing = min(outgoing_edges) if outgoing_edges else 0

              print(f"\tРассматриваем кусок {chunk}. Легчайшее входящее ребро =
{min_incoming}, а исходящее = {min_outgoing}.")
              half_sum += (min_incoming + min_outgoing) / 2

          return half_sum

      def calculate_lower_bound(path, remaining_cities, cost_matrix):
          mst_estimate = calculate_mst(cost_matrix, path, remaining_cities)
          half_sum_estimate = calculate_half_sum(cost_matrix, path,
remaining_cities)
          print(f"\tДля оставшегося пути вес минимального каркаса =
{mst_estimate}, минимальная полусумма = {half_sum_estimate}\n\t=> Берем
максимальную из двух оценок = {max(mst_estimate, half_sum_estimate)}.\n")
          return max(mst_estimate, half_sum_estimate)

      def branch_and_bound(cost_matrix):
          n = len(cost_matrix)
          best_path = None
          best_cost = float('inf')

          def backtrack(path, current_cost, remaining_cities):
              nonlocal best_path, best_cost

              if not remaining_cities:
                  total_cost = current_cost + cost_matrix[path[-1]][path[0]]
                  print(f"Все города были посещены в порядке:
{print_path(path)}. Стоимость найденного пути = {total_cost}.
\n-------------------------")
                  if total_cost < best_cost:
                      best_cost = total_cost
                      best_path = path[:]
                  return

              for u, v in get_allowed_edges(path, remaining_cities):
                  new_cost = current_cost + cost_matrix[u][v]
                  lower_bound = calculate_lower_bound(path + [v],
remaining_cities - {v}, cost_matrix)
```

```
                    if new_cost + lower_bound < best_cost:
                        backtrack(path + [v], new_cost, remaining_cities - {v})
                    else:
                        print("\t=> Данное решение заведомо плохое и не подходит.
Обрубаем ветку.\n")

            backtrack([0], 0, set(range(1, n)))
            return best_path, best_cost

        def print_matrix(matrix):
            print("Матрица стоимости путей:")
            for i in range(len(matrix[0])):
                print("\t".join(map(str, matrix[i])))
            print()

        def print_path(path):
            p = ""
            for i in range(len(path) - 1):
                p += f"{path[i]} -> "
            return p + f"{path[-1]}"

        cost_matrix = []
        opt = int(input("Хотите ли вы:\n\t1. Сгенерировать матрицу и сохранить ее
в файл;\n\t2. Загрузить матрицу из файла;\n\t3. Ввести матрицу вручную.\n"))

        if opt == 1:
            n = int(input("Введите размер матрицы стоимости путей: "))
            cost_matrix = generate_matrix(n)
            save_matrix_to_file(cost_matrix, "matrix.txt")
        elif opt == 2:
            try:
                cost_matrix = load_matrix_from_file("matrix.txt")
            except FileNotFoundError:
                print("Файл с матрицей не существует, генерируем матрицу размера
3.")
                cost_matrix = generate_matrix(3)
                save_matrix_to_file(cost_matrix, "matrix.txt")
        else:
            n = int(input("Введите размер матрицы стоимости путей: "))
            cost_matrix = []
            for i in range(n):
                row = list(map(float, input().split()))
                cost_matrix.append(row)

        opt = int(input("Какой из методов решения использовать?\n\t1. МВиГ;\n\t2.
АМР.\n"))
        best_path, best_cost = branch_and_bound(cost_matrix) if opt == 1 else
amr_algorithm(cost_matrix)
        print(f"Лучшее решение:\nВсе города были посещены в порядке:
{print_path(best_path)}. Стоимость найденного пути = {best_cost}.")
```