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: Коммивояжёр: ветви и границы. Вариант 3 .

Студентка гр. 3343 _____ Синицкая Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Целью лабораторной работы изучение и реализация алгоритма коммивояжера метода ветвей и границ (МВиГ) для решения задачи коммивояжёра с использованием стратегии последовательного роста пути.

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 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.

Основные шаги алгоритма:

1. Инициализация данных. Создается матрица расстояний, может быть как симметричной, так и несимметричной.

2. Инициализация начального пути. Выбирается начальный путь, который может быть сгенерирован случайным образом или сформирован на основе эвристики. Путь будет служить отправной точкой для дальнейших улучшений.

3. Определение стоимости пути. Рассчитывается стоимость текущего пути, которая определяется как сумма расстояний между последовательными объектами в пути, используя заранее подготовленную матрицу расстояний.

4. Поиск улучшений (поиск по соседям). Алгоритм начинает исследовать возможные соседние решения, полученные путем небольших изменений текущего пути. Для каждого варианта пути вычисляется его стоимость.

5. Выбор лучшего пути. Из всех возможных соседних путей выбирается тот, который имеет минимальную стоимость. Если найден путь с меньшей стоимостью, он становится новым текущим решением.

6. Проверка условия остановки. Алгоритм продолжает искать улучшения до тех пор, пока не будет достигнут критерий остановки. Это может быть задано как ограничение на количество итераций, отсутствие улучшений в течение нескольких шагов или достижение предельного времени работы.

7. Вывод решения. После завершения работы алгоритма выводится лучший найденный путь, а также его стоимость, являющаяся решением задачи.

Описание функций и методов:

1. *vector<vector<int>> generate_symmetric_matrix(int N, int min_w = 1, int max_w = 100)* — функция генерирует случайную симметричную матрицу расстояний между городами.

Параметры:

int N — количество городов.

int min_w — минимальное значение веса (по умолчанию 1).

int max_w — максимальное значение веса (по умолчанию 100).

Возвращает:

vector<vector<int>> — симметричная матрица расстояний между городами.

2. *vector<vector<int>> generate_matrix(int N, int min_w = 1, int max_w = 100)* — функция генерирует случайную матрицу расстояний между городами без симметрии.

Параметры:

int N — количество городов.

int min_w — минимальное значение веса (по умолчанию 1).

int max_w — максимальное значение веса (по умолчанию 100).

Возвращает:

vector<vector<int>> — матрица расстояний между городами.

3. *void save_matrix(const vector<vector<int>>& dist, const string& filename)* — метод сохраняет матрицу расстояний в файл.

Параметры:

const vector<vector<int>>& dist — матрица расстояний.

const string& filename — имя файла для сохранения.

4. *vector<vector<int>> load_matrix(const string& filename)* — функция загружает матрицу расстояний из файла.

Параметры:

const string& filename — имя файла для загрузки.

Возвращает:

vector<vector<int>> — загруженная матрица расстояний.

5. *string print_path(const vector<int>& path)* — функция формирует строковое представление пути.

Параметры:

const vector<int>& path — путь, состоящий из номеров городов.

Возвращает:

string — строковое представление пути.

6. *double calculate_total_cost(const vector<int>& path, const vector<vector<int>>& cost_matrix)* — функция вычисляет общую стоимость пути по матрице расстояний.

Параметры:

const vector<int>& path — путь.

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

Возвращает:

double — общая стоимость пути.

7. *pair<vector<int>, double> amr_algorithm(const vector<vector<int>>& cost_matrix)* — функция реализует алгоритм минимального разностного приближения (AMR) для поиска оптимального пути.

Параметры:

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

Возвращает:

pair<vector<int>, double> — оптимальный путь и его стоимость.

8. *vector<pair<int, int>> get_allowed_edges(const vector<int>& path, const set<int>& remaining_cities)* — функция находит все допустимые рёбра для расширения пути.

Параметры:

const vector<int>& path — текущий путь.

const set<int>& remaining_cities — множество оставшихся городов.

Возвращает:

vector<pair<int, int>> — допустимые рёбра, которые можно добавить к пути.

9. *int find(int u, map<int, int>& parent)* — функция для нахождения корня в структуре непересекающихся множеств.

Параметры:

int u — узел, для которого нужно найти корень.

map<int, int>& parent — ассоциативный массив для хранения родителей узлов.

Возвращает:

int — корень множества, к которому принадлежит узел.

10. *int calculate_mst(const vector<vector<int>>& cost_matrix, const vector<int>& path, const set<int>& remaining_cities)* — функция для вычисления веса минимального остовного дерева для оставшихся городов.

Параметры:

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

const vector<int>& path — текущий путь.

const set<int>& remaining_cities — множество оставшихся городов.

Возвращает:

int — вес минимального остовного дерева.

11. *double calculate_half_sum(const vector<vector<int>>& cost_matrix, const vector<int>& path, const set<int>& remaining_cities)* — функция для вычисления полусуммы весов двух рёбер, соединяющих оставшиеся города.

Параметры:

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

const vector<int>& path — текущий путь.

const set<int>& remaining_cities — множество оставшихся городов.

Возвращает:

double — полусумма весов двух рёбер.

12. *double calculate_lower_bound(const vector<int>& path, const set<int>& remaining_cities, const vector<vector<int>>& cost_matrix)* — функция для вычисления нижней границы стоимости для оставшегося пути.

Параметры:

const vector<int>& path — текущий путь.

const set<int>& remaining_cities — множество оставшихся городов.

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

Возвращает:

double — нижняя граница стоимости для оставшегося пути.

13. *pair<vector<int>, double> branch_and_bound(const vector<vector<int>>& cost_matrix)* — функция для реализации метода ветвей и границ для решения задачи о коммивояжере.

Параметры:

const vector<vector<int>>& cost_matrix — матрица стоимости между городами.

Возвращает:

pair<vector<int>, double> — оптимальный путь и его стоимость, найденные методом ветвей и границ.

14. *main()* — это основная точка входа в программу. Выбор способа задания матрицы расстояний между городами (ввод вручную, случайная генерация или загрузка из файла). После запускается алгоритм для решения задачи.

Оценка сложности алгоритма:

Временная сложность:

1. Генерация матрицы: для функции генерации симметричной матрицы или случайной матрицы временная сложность — O(N^2), где N — количество городов (строк и столбцов матрицы).

2. Вычисление стоимости пути (calculate_total_cost): для каждого пути, содержащего N городов, нужно суммировать стоимости рёбер. Время работы функции составляет O(N), так как требуется пройти по всем рёбрам пути.

3. Алгоритм AMR (amr_algorithm): в худшем случае для каждого шага алгоритм делает два вложенных цикла по всем городам (по N городам). Таким образом, сложность алгоритма AMR составит O(N^2), при условии, что в

каждой итерации могут быть рассмотрены все возможные перестановки городов.

4. Метод ветвей и границ (branch_and_bound): использует рекурсивный обход всех возможных путей, что потенциально может рассматривать все возможные перестановки городов. В худшем случае количество таких перестановок равно N!, что приводит к экспоненциальной сложности O(N!).

Однако метод использует оценки (lower bound) для отсечения невыгодных путей, что может существенно уменьшить количество вычислений. В реальных условиях сложность может быть значительно ниже.

5. Поиск в структуре непересекающихся множеств (find): используется метод сжатия пути и имеет сложность O(α(N)), где α(N) — это обратная функция от числа, которое растет очень медленно. Для практических значений N время работы этой операции можно считать практически постоянным.

6. Основной цикл: выбор способа задания матрицы требует O(N^2) времени для ввода или генерации матрицы, а также записи её в файл.

Итог:

В худшем случае O(N!) (если эвристика неэффективна или обрезание не применяется должным образом).

В среднем/практическом случае O(N^2) с хорошими эвристиками и обрезанием ветвей.

Сложность по памяти:

1. Матрицы расстояний: для хранения матрицы расстояний размером N×N используется O(N^2) памяти.

2. Пути и переменные: для хранения пути используется вектор из N элементов (O(N)). Для хранения допустимых рёбер в каждой итерации может использоваться дополнительная память, но её размер также зависит от количества городов, т.е. O(N).

3. Рекурсия в методе ветвей и границ: в случае метода ветвей и границ, стек рекурсии может хранить пути длины N, что требует O(N) дополнительной памяти на каждом уровне рекурсии.

Итог: сложность по памяти составит O(N^2)

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования задания 2.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2 -1 1 1 -1	--Метод Ветвей и Границ-- Города были посещены в порядке: 0 1 Стоимость найденного пути: 2.0 --Алгоритм Минимального Разностного приближения-- Города были посещены в порядке: 0 1 Стоимость найденного пути: 2.0	Минимальное количество городов. Результат соответствует ожидаемому.
2.	4 -1 10 10 10 10 -1 10 10 10 10 -1 10 10 10 10 -1	--Метод Ветвей и Границ-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 40.0 --Алгоритм Минимального Разностного приближения-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 40.0	Все города равны по стоимости. Результат соответствует ожидаемому.
3.	4 -1 1 100 1 1 -1 1 1 100 1 -1 1 1 1 1 -1	--Метод Ветвей и Границ-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 4.0	Высокая стоимость перехода в одном направлении (из города 0 в город 2) будет игнорирована. Результат соответствует ожидаемому.

		--Алгоритм Минимального Разностного приближения-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 4.0	
4.	4 -1 10 20 30 10 -1 5 15 20 5 -1 10 30 15 10 -1	--Метод Ветвей и Границ-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 55.0 --Алгоритм Минимального Разностного приближения-- Города были посещены в порядке: 0 1 2 3 Стоимость найденного пути: 55.0	Симметричная матрица. Результат соответствует ожидаемому.
5.	4 -1 3 4 1 1 -1 3 4 9 2 -1 4 8 9 2 -1	--Метод Ветвей и Границ-- Города были посещены в порядке: 0 3 2 1 Стоимость найденного пути: 6.0 --Алгоритм Минимального Разностного приближения-- Города были посещены в порядке: 0 3 2 1 Стоимость найденного пути: 6.0	Пример со степика. Результат соответствует ожидаемому.

Выводы.

В ходе выполнения лабораторной работы была реализована и исследована модификация метода ветвей и границ (МВиГ), основанная на стратегии последовательного роста пути. Внимание уделялось эффективному отсечению нерентабельных ветвей с использованием двух нижних оценок оставшегося пути: полусуммы весов двух легчайших рёбер в каждом из оставшихся фрагментов маршрута; оценки на основе минимального остовного дерева (МОД). Также была применена эвристика выбора дуги, основанная на стратегии поиска в глубину с приоритетом дуг, минимизирующих сумму веса добавляемой дуги и нижней оценки остатка пути. В качестве приближённого метода для сравнения был использован алгоритм минимального разностного приближения (АМР), позволяющий быстро получить разумное приближение оптимального решения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lr_2.cpp

```
#include <iostream>
#include <vector>
#include <climits>
#include <iomanip>
#include <fstream>
#include <random>
#include <algorithm>
#include <set>
#include <utility>
#include <sstream>
#include <map>

using namespace std;

using Path = vector<int>;   // определяем тип Path как вектор целых чисел для хранения пути
using Cost = double;        // определяем тип Cost как double для хранения стоимости

Path best_path;             // глобальная переменная для хранения лучшего найденного пути
Cost best_cost = INT_MAX;   // глобальная переменная для хранения лучшей найденной стоимости

// функция генерации случайной симметричной матрицы
vector<vector<int>> generate_symmetric_matrix(int N, int min_w = 1, int max_w = 100) {
    random_device rd;                                   // генератор случайных чисел
    mt19937 gen(rd());                                  // инициализация генератора
    uniform_int_distribution<> dis(min_w, max_w);   // определение распределения для целых чисел
    vector<vector<int>> matrix(N, vector<int>(N));   // создание матрицы NxN

    // проход по строкам матрицы
    for (int i = 0; i < N; ++i) {
        // проход по столбцам, начиная с диагонали
        for (int j = i + 1; j < N; ++j) {
            int w = dis(gen);                           // генерация случайного значения веса
            matrix[i][j] = matrix[j][i] = w;        // заполнение симметрично
        }
        matrix[i][i] = -1; // расстояние из себя в себя -1
    }
    return matrix;
}

// функция генерации случайной матрицы
```

```
    vector<vector<int>> generate_matrix(int N, int min_w = 1, int max_w
= 100) {
        random_device rd;                                    // генератор
случайных чисел
        mt19937 gen(rd());                                              //
инициализация генератора
        uniform_int_distribution<> dis(min_w, max_w);    // определение
распределения для целых чисел
        vector<vector<int>>  matrix(N,  vector<int>(N));    //  создание
матрицы NxN

        // проход по строкам матрицы
        for (int i = 0; i < N; ++i) {
            // проход по столбцам
            for (int j = 0; j < N; ++j) {
                // проверка на элемент на главной диагонали матрицы
                if (i == j) {
                    matrix[i][j] = -1; // расстояние из себя в себя -1
                } else  {
                    int w = dis(gen);  // генерация случайного значения
веса
                    matrix[i][j] = w;  // заполнение матрицы
                }
            }
        }
        return matrix;
    }

    // метод сохранения размера матрицы и матрицы в файл
    void  save_matrix(const  vector<vector<int>>&  dist,  const  string&
filename) {
        ofstream fout(filename);    // открытие файла для записи
        int N = dist.size();        // получение размера матрицы
        fout << N << "\n";          // запись размера в файл

        // проход по каждой строке матрицы
        for (const auto& row : dist) {
            for (int val : row) fout << val << " "; // запись значения
строки
            fout << "\n";                           // переход на новую
строку
        }
        fout.close(); // закрытие файла
    }

    // функция загрузки размера матрицы и матрицы из файла
    vector<vector<int>> load_matrix(const string& filename) {
        ifstream fin(filename);                          // открытие файла
для чтения
        int N;                                          // размер матрицы
        fin >> N;                                            // считывание
размера матрицы
        vector<vector<int>>  dist(N,  vector<int>(N));  //  инициализация
матрицы

        // проход по строкам
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j){
```

```
                fin >> dist[i][j];  // считывание элемента матрицы
            }
        }

        fin.close(); // закрытие файла
        return dist;
    }

    // функция вывода пути
    string print_path(const vector<int>& path) {
        stringstream ss; // создание строкового потока

        // прохож по всем элементам пути кроме последнего
        for (size_t i = 0; i < path.size() - 1; ++i) {
            ss << path[i] << " "; // добавление элемента в поток
        }
        ss << path.back();  // добавление последнего элемента

        return ss.str();
    }

    // функция вычисления общей стоимости пути
    double calculate_total_cost(const std::vector<int>& path, const
std::vector<std::vector<int>>& cost_matrix) {
        double total_cost = 0.0; // стоимость
        int n = path.size();     // размер пути

        // проход по всем элементам пути
        for (int i = 0; i < n; ++i) {
            total_cost += cost_matrix[path[i]][path[(i + 1) % n]]; //
стоимость между текущим и следующим городом
        }

        return total_cost;
    }

    // функция поиска с алгоритмом минимального разностного приближения
    pair<vector<int>, double> amr_algorithm(const vector<vector<int>>&
cost_matrix) {
        int n = cost_matrix.size();                                    //
количество городов
        vector<int> initial_path(n);                                 // массив
для начального пути
        iota(initial_path.begin(),   initial_path.end(),   0);       //
заполнение для начального пути по порядку от 0 до n-1

        vector<int>         best_path         =         initial_path;
// лучший путь
        double    best_cost    =    calculate_total_cost(initial_path,
cost_matrix); // подсчёт стоимости начального пути

        bool m = true;          // флаг улучшения пути
        int iterations = 0;     // счетчик итераций
        int F = n;              // максимальное количество итераций

        cout << "Начальный путь: " << print_path(best_path) << " со
стоимостью: " << best_cost << endl;
```

```
        while (m && iterations < F) {
            m = false; // сброс флага улучшения пути

            // проход по всем городам
            for (int i = 1; i < n; ++i) {
                for (int j = 1; j < n; ++j) {
                    vector<int>       new_path       =       best_path;
// сохранение текущего лучшего пути
                    swap(new_path[i],                      new_path[j]);
// обмен местами двух городов
                    double  new_cost  =  calculate_total_cost(new_path,
cost_matrix);  // подсчет новой стоимости

                    // проверка на лучше ли новая стоимость
                    if (new_cost < best_cost) {

                        cout  <<  "\tМеняем  местами  города  "  <<
best_path[i] << " и " << best_path[j] << endl;
                        cout << "\tБыло обнаружено лучшее решение " <<
print_path(new_path) << " со стоимостью " << new_cost << " (улучшение на
" << (best_cost - new_cost) << ")" << endl;

                        best_path = new_path;    // обновление лучшего
пути
                        best_cost = new_cost;     // обновление лучшей
стоимости
                        m = true;                    // установка флага
лучшего пути
                        iterations++;             // увеличение счетчика
итераций
                        break;                   // выход из внутреннего
цикла
                    }
                }
            }
        }

        cout  <<  "Все  города  были  посещены  в  порядке:  "  <<
print_path(best_path) << " Стоимость найденного пути: " << best_cost <<
endl;

        return {best_path, best_cost};
    }

    // фуункция получения допустимых рёбер для добавления к текущему
пути
    vector<pair<int, int>> get_allowed_edges(const vector<int>& path,
const set<int>& remaining_cities) {
        vector<pair<int, int>> allowed_edges; // массив для хранения
допустимых рёбер
        int last_city = path.back();             // последний город из
пути

        // проход по всем оставшимся городам
        for (int city : remaining_cities) {
            allowed_edges.emplace_back(last_city, city); // добавление
рёбра к последнему городу
        }
```

```
        return allowed_edges;
    }

    // функция поиска корня в структуре непересекающихся множеств
    int find(int u, map<int, int>& parent) {
        // пока узел не является корнем
        while (parent[u] != u) {
            parent[u] = parent[parent[u]]; // сокращаем путь (родитель текущего узла сразу ссылается на корень)
            u = parent[u];                 // переход к родителю
        }
        return u;
    }

    // функция вычисления веса минимального остовного дерева
    int calculate_mst(const vector<vector<int>>& cost_matrix, const vector<int>& path, const set<int>& remaining_cities) {
        vector<vector<int>> chunks;               // массив для хранения частей пути
        chunks.push_back(path);                   // добавление текущего пути как первой части
        for (int city : remaining_cities) {
            chunks.push_back({city});                    // добавление оставшиеся городов как отдельных частей
        }

        cout << "\tОценка оставшегося пути с помощью МОД для оставшихся кусков:\n\t";

        // проход по частям
        for (size_t i = 0; i < chunks.size(); ++i) {
            // проход по элементам в части
            for (size_t j = 0; j < chunks[i].size(); ++j) {
                cout << chunks[i][j] << " ";                  // вывод элементов
            }
            if (i + 1 < chunks.size()) cout << "| ";         // запись разделителя между частями
        }
        cout << endl;

        cout << "\tВсе доступные рёбра:\n";

        vector<tuple<int, int, int>> edges; // массив для хранения рёбер

        // проход по частям
        for (size_t i = 0; i < chunks.size(); ++i) {
            // проход по частям для соединения
            for (size_t j = 0; j < chunks.size(); ++j) {
                // пропуск самосоединения
                if (i != j) {
                    int start = chunks[i].back();         // последний город из первой части
                    int end = chunks[j].front();         // первый город из второй части
```

```
                    int cost = cost_matrix[start][end]; // стоимость
между городами

                    // проверка, что путь допустим
                    if (cost != -1) {

                        cout << "\t\t" << start << " -> " << end << "
стоимость = " << cost << endl;

                        edges.emplace_back(cost, start, end); //
добавление ребра в массив для хранения рёбер
                    }
                }
            }
        }

        sort(edges.begin(), edges.end()); // сортировка рёбер по весу

        // инициализация родителей для поиска
        map<int, int> parent; // ассоциативный массив для хранения
родителя каждого узла
        // проход по частям
        for (const auto& chunk : chunks) {
            // проход по городам в каждой части
            for (int city : chunk) {
                parent[city] = city; // установка себя как родителя
            }
        }

        int mst_weight = 0; // вес минимального остовного дерева
        // проход по всем рёбрам
        for (const auto& [cost, u, v] : edges) {
            int root_u = find(u, parent); // нахождение кореня для
первой точки
            int root_v = find(v, parent); // нахождение кореня для
второй точки
            // проверка, если корни разные
            if (root_u != root_v) {

                cout << "\tДобавление к каркасу ребра " << u << " -> "
<< v << " со стоимостью " << cost << endl;

                mst_weight += cost; // увеличение веса
минимального остова
                parent[root_v] = root_u; // объединение двух
множеств
            }
        }

        return mst_weight;
    }

    // функция вычисления полусуммы весов двух рёбер
    double calculate_half_sum(const vector<vector<int>>& cost_matrix,
const vector<int>& path, const set<int>& remaining_cities) {
        vector<vector<int>> chunks; // массив для хранения частей пути
        chunks.push_back(path);     // добавление текущего пути
        // добавление оставшихся городов как отдельных частей
```

```
        for (int city : remaining_cities) {
            chunks.push_back({city});
        }

        double half_sum = 0; // полусумма

        cout << "\n\tОценка оставшегося пути с помощью полусуммы весов
двух легчайших рёбер по всем кускам:\n\t";

        // проход по частям
        for (size_t i = 0; i < chunks.size(); ++i) {
            // проход по элементам в части
            for (size_t j = 0; j < chunks[i].size(); ++j) {
                cout << chunks[i][j]<< " ";           // вывод элемента
            }
            if (i + 1 < chunks.size()) cout << "| "; // разделитель
между частями
        }
        cout <<endl;

        // проход по частям
        for (const auto& chunk : chunks) {
            vector<int> incoming_edges; // массив для хранения входящих
рёбер
            // проход по всем частям для поиска входящих рёбер
            for (const auto& other_chunk : chunks) {
                if (other_chunk != chunk) {
                    int start = other_chunk.back();       // последний
город из другой части
                    int end = chunk.front();            // первый город
                    int cost = cost_matrix[start][end]; // стоимость
                    if (cost != -1) {
                        incoming_edges.push_back(cost);  //  добавление
стоимости во входящие рёбра
                    }
                }
            }
            int   min_incoming   =   incoming_edges.empty()   ?   0   :
*min_element(incoming_edges.begin(), incoming_edges.end()); // нахождение
минимального входящего ребра

            vector<int>   outgoing_edges;   //   массив   для   хранения
исходящих рёбер
            // проход по частям
            for (const auto& other_chunk : chunks) {
                if (other_chunk != chunk) {
                    int start = chunk.back();             // последний
город в текущей части
                    int end = other_chunk.front();      // первый город
в другой части
                    int cost = cost_matrix[start][end]; // стоимость
                    if (cost != -1) {
                        outgoing_edges.push_back(cost);  //  добавление
стоимости в исходящие рёбра
                    }
                }
            }
```

```
            int min_outgoing = outgoing_edges.empty() ? 0 : *min_element(outgoing_edges.begin(), outgoing_edges.end()); // нахождение минимального исходящего ребра

            cout << "\tРассматриваем кусок {";
            for (size_t i = 0; i < chunk.size(); ++i) {
                cout << chunk[i];
                if (i + 1 < chunk.size()) cout << " ";
            }
            cout << "} Минимальное входящее ребро = " << min_incoming << ", исходящее = " << min_outgoing << endl;

            half_sum += (min_incoming + min_outgoing) / 2.0; // оценка полусуммы
        }

        return half_sum;
    }

    // функция вычисления нижней границы стоимости
    double calculate_lower_bound(const vector<int>& path, const set<int>& remaining_cities, const vector<vector<int>>& cost_matrix) {
        double mst_estimate = calculate_mst(cost_matrix, path, remaining_cities);            // получение оценки минимального остовного дерева
        double half_sum_estimate = calculate_half_sum(cost_matrix, path, remaining_cities); // получение оценки полусуммы

        cout << "\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) << endl;

        return max(mst_estimate, half_sum_estimate);
    }

    // функция реализации метода ветвей и границ
    pair<vector<int>, double> branch_and_bound(const vector<vector<int>>& cost_matrix) {
        int n = cost_matrix.size();                                                  // количество городов
        vector<int> best_path;                                                       // массив для хранения лучшего пути
        double best_cost = numeric_limits<double>::infinity();                       // лучшая стоимость

        // рекурсивная функция
        function<void(vector<int>, double, set<int>)> backtrack = [&](vector<int> path, double current_cost, set<int> remaining) {
            // проверка, если все города посещены
            if (remaining.empty()) {
                double total_cost = current_cost + cost_matrix[path.back()][path[0]]; // рассчет полной стоимости цикла

                cout << "Все города были посещены в порядке: " << print_path(path) << " Стоимость: " << total_cost << "\n----------------------\n";
```

```
                // проверка, если новое решение лучше
                if (total_cost < best_cost) {
                    best_cost = total_cost; // обновление лучшей стоимости
                    best_path = path;       // обновление лучшего пути
                }
                return; // выход из функции
            }

            vector<pair<int, int>> edges = get_allowed_edges(path, remaining);  // получение допустимых рёбер для текущего пути

            // проход по каждому допустимому ребру
            for (const auto& [u, v] : edges) {
                double edge_cost = cost_matrix[u][v]; // стоимость ребра
                if (edge_cost < 0) continue;              // пропуск недопустимых рёбер

                set<int> new_remaining = remaining; // сохранение оставшихся городов
                new_remaining.erase(v);                     // удаление текущего города из оставшихся

                vector<int> new_path = path; // сохрание текущего пути
                new_path.push_back(v);           // добавление текущего города

                double lower_bound = calculate_lower_bound(new_path, new_remaining, cost_matrix); // вычисление нижней границы
                double total_estimate = current_cost + edge_cost + lower_bound;                   // оценка стоимости

                // проверка, если оценка меньше лучшей стоимости
                if (total_estimate < best_cost) {
                    backtrack(new_path, current_cost + edge_cost, new_remaining); // рекурсивный вызов
                } else {
                    cout << "\t=> Отсечение ветки для города " << v << " (оценка " << total_estimate << " >= " << best_cost << ")\n";
                }
            }
        };

        // инициализация: начинаем с города 0
        set<int> initial_remaining; // множество для хранения оставшихся городов
        // добавление всех городов кроме первого
        for (int i = 1; i < n; ++i) {
            initial_remaining.insert(i);
        }

        backtrack({0}, 0.0, initial_remaining); // запуск рекурсии с первого города

        return {best_path, best_cost};
    }
```

```
int main() {
    int N;                      // количество городов
    vector<vector<int>> dist; // матрица расстояний между городами
    int choice;                 // выбор способа задания матрицы

    cout << "Выберите способ задания матрицы весов:\n";
    cout << "1 - Ввести вручную\n";
    cout << "2 - Сгенерировать случайную матрицу\n";
    cout << "3 - Сгенерировать случайную симметричную матрицу\n";
    cout << "4 - Загрузить матрицу из файла\n";
    cout << "Ваш выбор: ";
    cin >> choice;

    switch (choice) {
        case 1: { // ввод вручную
            cout << "Введите количество городов: ";
            cin >> N;
            dist.assign(N, vector<int>(N));
            cout << "Введите матрицу стоимостей (" << N << "x" << N
<< "):\n";
            for (int i = 0; i < N; ++i)
                for (int j = 0; j < N; ++j)
                    cin >> dist[i][j];
            save_matrix(dist, "manual_input_matrix.txt");
            cout       <<       "Матрица       сохранена       в
manual_input_matrix.txt\n";
            break;
        }

        case 2: { // генерация случайной матрицы
            cout << "Введите количество городов: ";
            cin >> N;
            dist = generate_matrix(N);
            save_matrix(dist, "matrix.txt");
            cout << "Произвольная матрица сгенерирована и сохранена
в matrix.txt\n";
            break;
        }

        case 3: { // генерация симметричной матрицы
            cout << "Введите количество городов: ";
            cin >> N;
            dist = generate_symmetric_matrix(N);
            save_matrix(dist, "generated_symmetric_matrix.txt");
            cout << "Симметричная матрица сгенерирована и сохранена
в generated_symmetric_matrix.txt\n";
            break;
        }

        case 4: { // загрузка матрицы из файла
            string filename;
            cout << "Введите имя файла: ";
            cin >> filename;

            ifstream fin(filename);
            if (!fin) {
                cerr << "Ошибка открытия файла.\n";
                return 1;
```

```
                }

                fin >> N;
                dist = load_matrix(filename);
                if (dist.empty()) {
                    cerr << "Ошибка загрузки файла.\n";
                    return 1;
                }
                break;
            }

            default:
                cerr << "Неверный выбор.\n";
                return 1;
        }

        vector<int> best_path_mvg; // лучший путь метода ветвей и
границ
        double best_cost_mvg;       // лучшая стоимость метода ветвей и
границ

        vector<int>   best_path_amr;   //   лучший   путь   алгоритма
минимального разностного приближения
        double best_cost_amr;          // лучшая стоимость алгоритма
минимального разностного приближения

        cout<<"Метод Ветвей и Границ:"<<endl;
        auto result_mvg = branch_and_bound(dist);    // запуск метода
ветвей и границ
        best_path_mvg = result_mvg.first;           // получение лучший
путь
        best_cost_mvg = result_mvg.second;          // получение лучшую
стоимость

        cout<<"Алгоритм Минимального Разностного приближения:"<<endl;
        auto result_amr = amr_algorithm(dist);      // запуск алгоритма
AMP
        best_path_amr = result_amr.first;           // получение лучший
путь
        best_cost_amr = result_amr.second;          // получение лучшую
стоимость


        cout << endl << "====Лучший результат====" << endl;
        cout << endl << "--Метод Ветвей и Границ--" << endl;
        cout   <<   "Города   были   посещены   в   порядке:   "   <<
print_path(best_path_mvg)<< endl;
        cout   <<   "Стоимость   найденного   пути:   "   <<   fixed   <<
setprecision(1) << best_cost_mvg <<endl;

        cout   <<   endl   <<   "--Алгоритм   Минимального   Разностного
приближения--" << endl;

        cout   <<   "Города   были   посещены   в   порядке:   "   <<
print_path(best_path_amr)<< endl;
        cout   <<   "Стоимость   найденного   пути:   "   <<   fixed   <<
setprecision(1) << best_cost_amr <<endl;
```

```
    return 0;
}
```