**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: Задача Коммивояжера

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Песчатский С. Д. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## Задание №1

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 Mi,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.

Пример входных данных:

3

-1 1 3

3 -1 1

1 2 -1

Пример выходных данных:

0 1 2

3.0

## Индивидуальный вариант

Вариант 4. МВиГ: последовательный рост пути + использование для отсечения двух нижних оценок ве́са оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — не в глубину, а по антиприоритету (S+L)/(0,5N+k). Приближённый алгоритм: АВБГ "улучшенный". Замечание к варианту 4. И МВиГ, и АВБГ "улучшенный" начинать со стартовой вершины.

## Описание алгоритма

Задача коммивояжёра — это классическая проблема дискретной оптимизации. Её начали исследовать ещё в XIX столетии, и сегодня она остаётся ключевой в теории алгоритмов и операционном анализе. Суть задачи — найти минимальный замкнутый маршрут, который проходит через каждый город ровно один раз и возвращается в начальный пункт.

Задача имеет несколько возможных методов решения:

* **Точные методы:** Полный перебор; Метод ветвей и границ (метод Литтла).
* **Приближенные алгоритмы**: Жадный метод; Муравьиный метод; Метод с использованием минимального остовного дерева

В текущей реализации рассмотрим метод ветвей и границ, а также приближенный алгоритм включения ближайшего города (АВБГ).

Работа алгоритма решающего ЗК по методу ветвей и границ (Задание №1):

1) Инициализация: программа считывает номер стартовой вершины и матрицу стоимостей. Все значения -1 в матрице заменяются на INF (бесконечность), обозначая отсутствие связи между городами. Начальное значение best\_cost устанавливается равным INF, что означает отсутствие найденного решения на старте алгоритма.

2) При количестве вершин равном 2 решение проверяется напрямую: если между вершинами существует двусторонняя связь (стоимости не равны INF), оптимальный маршрут строится как [0, 1, 0] с суммарной стоимостью graph[0][1] + graph[1][0]. В противном случае алгоритм переходит к стандартной процедуре: создается начальный узел с путем из стартовой вершины, нулевой границей и приоритетом, вычисленным по эвристической формуле. Очередь с приоритетом инициализируется этим узлом для последующей обработки.

3) Алгоритм ветвей и границ использует итеративный подход с приоритетной очередью. На каждом шаге:

Матрица сокращается методом reduce\_matrix

Текущая граница (current\_bound) увеличивается на сумму вычтенных минимумов

Если current\_bound превышает best\_cost - ветка отбрасывается

Для выбора ребра ветвления (find\_new\_branch):

* + Для каждого нуля в матрице вычисляется штраф (сумма минимальных элементов строки и столбца без этого нуля)
  + Выбирается ребро с максимальным штрафом

Создаются две новые ветки:

* + Ветка включения ребра: обновляются ограничения и путь
  + Ветка исключения ребра: добавляется запрет на выбранное ребро

Обе ветки добавляются в приоритетную очередь для дальнейшей обработки

4) Завершение обработки ветки происходит когда путь включает все города. Проверяется возможность завершения цикла (возврат в стартовый город). Если стоимость полученного маршрута меньше best\_cost - он становится новым оптимальным решением. При равенстве стоимостей маршрут добавляется в список лучших вариантов. Алгоритм продолжает обработку оставшихся в очереди веток до их исчерпания.

5) Завершение алгоритма: сортируются все найденные оптимальные пути, затем выводится первый путь и его стоимость

Работа алгоритма решающего ЗК по приближенному алгоритм включения ближайшего города (АВБГ): (Задание №2)

1) Инициализация: считывается стартовая вершина и матрица стоимостей. Значения -1 заменяются на INF (нет пути). Создается пустой маршрут и массив посещенных вершин.

2) АВБГ: создается вектор tour для хранения пути и вектор visited для хранения посещённых вершин. Начиная с первого города, ищем ближайший к нему город, а также ближайший к найденному. Найдя необходимые города, проверяем подходят ли они по формуле I+O-R, где I — вес дуги в новый город, O — вес дуги из нового города, R — вес исключаемой дуги. Если ближайший город улучшает ситуацию, то добавляем его в путь.

4) Завершение: алгоритм завершится после того как длина пути станет равно количеству.

**Оценки сложности алгоритмов:**

## Алгоритм ветвей и границ (метод ветвей и границ с приоритетной очередью):

## Время - O(n² \* 2ⁿ) в худшем случае

## Обработка узла: O(n²) для оценки границы

## Число узлов: до O(2ⁿ)

## Память - O(n \* 2ⁿ) для хранения очереди

## Алгоритм ближайшего соседа (АВБГ):

## Время - O(n²)

## Выбор следующей вершины: O(n) операций

## Всего n итераций

## Память - O(n²) для матрицы смежности

## Описание функций и структур данных

1) Класс Node — класс узла, хранящий информацию о пути, границе и приоритете.

Поля:

1) vector<int> path — текущий путь.

2) double bound— граница для узла.

3) double priority — приоритет узла.

2) Функция calculate\_bound(path, min\_edges)

- Вычисляет границу веса пути.

Параметры:

1) vector<double> path — вектор текущего пути.

2) vector<pair<double, double>>& min\_edges – массив двух минимальных ребер для каждой вершины.

Используемые структуры данных:

3) solve\_tsp() - функция ветвления. Считает минимальные ребра для каждой вершины, и методом ветвей и границ решает задачу коммивояжера. Возвращает пару из лучшего пути и его длины.

4) nearest\_neighbor\_tsp() – функция решающая задачу коммивояжера, методом АВБГ. Возвращает найденный путь.

5) calculate\_path\_cost(path) – функция высчитывающая длину пути

Параметр: const vector<int>& path – путь, длину которого необходимо вычислить

6) main() – головная функция, где происходит считывание вершины начала и матрицы весов.

**Тестирование**

Результаты тестирования задания №1 представлены в табл. 1.

Результаты тестирования задания №2 представлены в табл. 2.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 3  -1 1 3  3 -1 1  1 2 -1 | 0 1 2  3.0 | Верный вывод |
|  | 4  -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1 | 0 3 2 1  6.0 | Верный вывод |
|  | 5  -1 18.97 22.36 19.42 3.61  18.97 -1 35.61 38.01 17.0  22.36 35.61 -1 16.28 21.19  19.42 38.01 16.28 -1 21.02  3.61 17.0 21.19 21.02 -1 | 0 4 1 2 3  91.9 | Верный вывод |
|  | 2  -1 10  4 -1 | 0 1  14.0 | Верный вывод |

Таблица 2 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 2  -1 18.97 22.36 19.42 3.61  18.97 -1 35.61 38.01 17.0  22.36 35.61 -1 16.28 21.19  19.42 38.01 16.28 -1 21.02  3.61 17.0 21.19 21.02 -1 | 78.58  2 0 4 1 2 | Верный вывод |
|  | 4  -1 18.97 22.36 19.42 3.61  18.97 -1 35.61 38.01 17.0  22.36 35.61 -1 16.28 21.19  19.42 38.01 16.28 -1 21.02  3.61 17.0 21.19 21.02 -1 | 79.38  4 0 1 2 4 | Верный вывод |
|  | 2  -1 3 4 1  1 -1 3 4  9 2 -1 4  8 9 2 -1 | 7  2 1 0 2 | Верный вывод |
|  | 0  -1 1 3  3 -1 1  1 2 -1 | 3  0 1 2 0 | Верный вывод |

# Приложение А Исходный код программы

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <limits>

#include <cmath>

#include <unordered\_set>

#include <iomanip>

using namespace std;

const double INF = numeric\_limits<double>::infinity(); // Бесконечность для представления отсутствия ребра

vector<vector<double>> graph; // Матрица смежности графа

int n; // Количество вершин в графе

int start; // Стартовая вершина

vector<pair<vector<int>, double>> allPaths; // Хранит все найденные пути и их стоимость

// Структура для узла в алгоритме ветвей и границ

struct Node {

    vector<int> path; // Текущий путь

    double bound; // Оценка нижней границы для этого пути

    double priority; // Приоритет для очереди с приоритетом

    // Перегрузка оператора < для min-heap (чем меньше priority, тем выше приоритет)

    bool operator<(const Node& other) const {

        return priority > other.priority; // Инвертируем сравнение для min-heap

    }

};

// Функция вычисления стоимости пути

double calculate\_path\_cost(const vector<int>& path) {

    double cost = 0;

    for (size\_t i = 0; i < path.size() - 1; ++i) {

        cost += graph[path[i]][path[i+1]]; // Суммируем веса всех ребер пути

    }

    return cost;

}

// Функция вычисления нижней границы для частичного пути

double calculate\_bound(const vector<int>& path,

                      const vector<pair<double, double>>& min\_edges) {

    double bound = 0;

    // 1. Добавляем вес уже пройденного пути

    bound += calculate\_path\_cost(path);

    // 2. Добавляем полусумму двух минимальных ребер для оставшихся вершин

    unordered\_set<int> visited(path.begin(), path.end());

    for (int node = 0; node < n; ++node) {

        if (visited.find(node) == visited.end()) { // Для непосещенных вершин

            if (min\_edges[node].second == INF) {

                if (min\_edges[node].first == INF) return INF; // Если нет ребер - возвращаем бесконечность

                bound += min\_edges[node].first; // Добавляем единственное доступное ребро

            } else {

                // Добавляем полусумму двух минимальных ребер

                bound += (min\_edges[node].first + min\_edges[node].second) / 2;

            }

        }

    }

    return bound;

}

// Основная функция решения задачи коммивояжера методом ветвей и границ

pair<vector<int>, double> solve\_tsp() {

    if (n == 0) return {{}, 0}; // Если граф пустой

    // Предварительные вычисления: два минимальных ребра для каждой вершины

    vector<pair<double, double>> min\_edges(n);

    for (int i = 0; i < n; ++i) {

        vector<double> edges;

        for (int j = 0; j < n; ++j) {

            if (i != j) edges.push\_back(graph[i][j]); // Собираем все ребра, кроме петли

        }

        sort(edges.begin(), edges.end()); // Сортируем ребра по весу

        min\_edges[i] = (edges.size() >= 2) ?

                       make\_pair(edges[0], edges[1]) : // Берем два минимальных

                       make\_pair(edges[0], INF); // Или одно, если ребро только одно

    }

    priority\_queue<Node> pq; // Очередь с приоритетом для узлов

    // Начальный узел: начинаем с вершины 0

    Node initial;

    initial.path = {0};

    initial.bound = calculate\_bound(initial.path, min\_edges);

    initial.priority = (1 + 0) / (0.5 \* n + 1); // Эвристика приоритета: (S+L)/(0.5N+k)

    pq.push(initial);

    vector<int> best\_path; // Лучший найденный путь

    double best\_cost = INF; // Стоимость лучшего пути

    // Специальный случай для графа из 2 вершин

    if(n==2){

        if(graph[0][1]!=INF && graph[1][0]!=INF){

            best\_cost=graph[0][1]+graph[1][0];

            best\_path={0,1,0};

            allPaths.push\_back({best\_path, best\_cost});

        }

    }

    else{

    // Основной цикл алгоритма ветвей и границ

    while (!pq.empty()) {

        Node current = pq.top();

        pq.pop();

        // Если текущая оценка хуже лучшего решения - отсекаем ветвь

        if (current.bound >= best\_cost) continue;

        // Если путь содержит все вершины - проверяем на лучшее решение

        if (current.path.size() == static\_cast<size\_t>(n)) {

            // Добавляем ребро обратно в начальную вершину

            double cost = calculate\_path\_cost(current.path) + graph[current.path.back()][current.path[0]];

            if (cost < best\_cost) {

                best\_cost = cost;

                allPaths.clear();

                best\_path = current.path;

                best\_path.push\_back(best\_path[0]); // Замыкаем цикл

                allPaths.push\_back({best\_path, best\_cost});

            }

            current.path.push\_back(current.path[0]);

            allPaths.push\_back({current.path, cost});

            continue;

        }

        // Генерируем потомков - все возможные следующие вершины

        int last = current.path.back();

        for (int next\_node = 0; next\_node < n; ++next\_node) {

            // Проверяем, что вершина еще не посещена и есть ребро

            if (find(current.path.begin(), current.path.end(), next\_node) == current.path.end() &&

                graph[last][next\_node] != INF) {

                Node child;

                child.path = current.path;

                child.path.push\_back(next\_node);

                child.bound = calculate\_bound(child.path, min\_edges);

                // Если оценка лучше текущего лучшего решения - добавляем в очередь

                if (child.bound < best\_cost) {

                    double S = child.path.size();

                    double L = calculate\_path\_cost(child.path);

                    child.priority = (S + L) / (0.5 \* n + S); // Эвристика приоритета

                    pq.push(child);

                }

            }

        }

    }

    }

    // Сортируем все найденные пути для удобства

    vector<vector<int>> paths\_tmp;

    for(int i=0; i<allPaths.size(); i++){

        paths\_tmp.push\_back(allPaths[i].first);

    }

    sort(paths\_tmp.begin(), paths\_tmp.end());

    return {best\_path, best\_cost};

}

// Решение задачи коммивояжера методом ближайшего соседа

vector<int> nearest\_neighbor\_tsp() {

    if (n == 0) return {};

    vector<int> tour; // Результирующий тур

    vector<bool> visited(n, false); // Отметки о посещении вершин

    // Начинаем с стартового города

    int current\_city = start;

    tour.push\_back(current\_city);

    visited[current\_city] = true;

    // Последовательно добавляем ближайшие города

    for (int i = 1; i < n; ++i) {

        double min\_dist = INF;

        int nearest\_city = -1;

        double min3 = INF;

        int The\_city = -1;

        // Ищем ближайший непосещенный город с учетом эвристики

        for (int city = 0; city < n; ++city) {

            if (!visited[city] && graph[current\_city][city] < min\_dist) {

                min\_dist = graph[current\_city][city];

                nearest\_city = city;

                int nearest\_toThat = -1;

                double min2 = INF;

                // Ищем ближайшего соседа к найденному городу

                for(int c2 = 0; c2<n; ++c2){

                    if(!visited[c2] && graph[nearest\_city][c2] < min2){

                        min2 = graph[nearest\_city][c2];

                        nearest\_toThat = c2;

                    }

                }

                // Эвристика: выбираем город, который дает минимальную оценку

                if(!visited[city] && !visited[nearest\_toThat] && (min\_dist+min2-graph[current\_city][nearest\_toThat])<min3){

                    The\_city = nearest\_city;

                    min3 = min\_dist+min2-graph[current\_city][nearest\_toThat];

                }

            }

        }

        // Добавляем найденный город в тур

        if (The\_city != -1) {

            tour.push\_back(The\_city);

            visited[The\_city] = true;

            current\_city = The\_city;

        }

    }

    // Замыкаем тур, возвращаясь в начальный город

    tour.push\_back(tour[0]);

    return tour;

}

int main() {

    bool choice = 1; // Флаг выбора алгоритма (0 - ближайший сосед, 1 - ветвей и границ)

    cin >> start; // Читаем стартовую вершину

    cin.ignore();

    string l;

    // Чтение матрицы смежности из входных данных

    while(getline(cin,l)){

        if(l.empty()) break;

        vector<double> r;

        size\_t p = 0;

        while(p < l.length()){

            double n = stod(l.substr(p)); // Преобразуем строку в число

            r.push\_back(n == -1 ? INF : n); // -1 заменяем на бесконечность

            p = l.find(' ', p);

            if(p == string::npos) break;

            p++;

        }

        graph.push\_back(r);

    }

    n = graph.size();

    start = 0; // Всегда начинаем с 0 вершины

    if(choice){

        // Запуск алгоритма ветвей и границ

        auto res = solve\_tsp();

        // Вывод результата

        for(int a = 0; a < res.first.size()-1; a++){

            if(a < res.first.size()-2) cout << res.first[a] << " ";

            else cout << res.first[a];

        }

        cout << endl << setprecision(1) << fixed << res.second;

    }

    else{

        // Запуск алгоритма ближайшего соседа

        // Заменяем -1 на INF в матрице смежности

        for (int i = 0; i < graph.size(); ++i) {

            for (int j = 0; j < graph.size(); ++j) {

                if (graph[i][j] == -1) graph[i][j] = INF;

            }

        }

        vector<int> tour = nearest\_neighbor\_tsp();

        cout << calculate\_path\_cost(tour) << endl; // Вывод стоимости тура

        // Вывод самого тура

        for (int city : tour) {

            cout << city << " ";

        }

    }

    return 0;

}