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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Коммивояжер (TSP)

Студент гр. 1304 Поршнев Р.А.

Преподаватель Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Написать программу, которая решает задачу коммивояжера.

Задание.

Дана карта городов в виде ассиметричного, неполного графа $G = (V, E)$, где V ($|V| = n$) – это вершины графа, соответствующие городам; E ($|E| = m$) – это ребра между вершинами графа, соответствующие путям сообщения между этими городами. Каждому ребру m_{ij} (переезд из города i в город j) можно сопоставить критерий выгодности маршрута (вес ребра) равный w_i (натуральное число [1, 1000]), $m_{ij} = inf$, если $i = j$.

Если маршрут включает в себя ребро m_{ij}, то $x_{ij} = 1$, иначе $x_{ij} = 0$.

Требуется найти минимальный маршрут (минимальный гамильтонов цикл):

$$\min W = \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} w_{ij}$$

Основные теоретические положения.

Для решения задачи коммивояжера применён алгоритм Беллмана-Хелда-Карпа. Данный алгоритм реализует принцип динамического программирования, который заключается в том, что для того, чтобы решить сложную задачу, её нужно разбить на несколько более простых задач, причём для решения каждой такой задачи используется решение более простой задачи. Асимптотика реализованного алгоритма: $O(n^3 2^n \log n)$. Данная асимптотика отличается от оригинальной, равной $O(n^2 2^n)$, из-за того, что в качестве хранения стоимости пути из стартовой вершины в конечную для каждого подмножества используется не двумерный, а ассоциативный массив std::map. Для реализации алгоритма Беллмана-Хелда-Карпа был выбран язык C++ ввиду его быстродействия, ведь при количестве узлов, равном 20, программа должна работать не более трёх минут в среднем.

Выполнение работы.

Для считывания и получения графа был реализован класс *Graph*. Данный класс содержит следующие методы:

- *void ReadGraph()* – данный метод реализует считывание графа из файла и его запись в матрицу смежности;
- *AdjacencyMatrix GetGraphStorage()* – данный метод предназначен для получения графа в виде матрицы смежности.

Решение поставленной задачи реализовано в классе *TravellingSalesmanProblem*. Для данного класса реализован пользовательский конструктор, который принимает на вход матрицу смежности графа и производит инициализацию данных определёнными значениями. Входные данные: матрица смежности графа. Данный класс содержит следующие методы:

- *Run()* – данный метод предназначен для запуска отсчёта времени работы алгоритма, метода генерации всех подмножеств узлов графа, метода решения задачи коммивояжера и метода для вывода ответа на исходную задачу;
- *void GenerateSubsetsOfNodes()* – данный метод предназначен для генерации всех подмножеств узлов графа, что происходит следующим образом: перебираются все числа от 1 до $2^{n-1}-1$, где n – число узлов в графе. Затем каждое такое число переводится в двоичную систему счисления и каждое такое битовое представление подмножества имеет длину $n-1$ бит. Начиная с 0 и двигаясь слева направо, можно получить информацию о вхождении (i + 2)-го узла в данное подмножество: если бит установлен в 1, то (i + 2)-ой узел входит в данное подмножество, иначе – не входит;
- *void FindCheapestHamiltonianCycle()* – данный метод запускает алгоритм Белламана-Хелда-Карпа: инициализирует информацию единичных подмножествах и перебирает возможные размеры подмножеств;
- *void IteratingThroughSubsetWithoutNode(Set subset)* – данный метод предназначен для независимого выкалывания каждого узла из рассматриваемого подмножества. Входные данные: подмножество узлов в виде вектора;
- *void FindMinCostOfPathInSubset(Set &subset, int nodeName)* – данный метод предназначен для поиска самого дешёвого пути от стартового узла до выколотого при условии, что путь проходит только через данное подмножество.

Входные данные: подмножество узлов в виде вектора и имя узла, который выколот;

- *Set GetCleanSubset(SetInBitForm subsetInBitForm)* – данный метод предназначен для очистки подмножества узлов в виде строки от лишних нулей, которые появились в результате применения *std::bitset*, а также для перевода уже очищенной от нулей строки к виду вектора узлов. Выходные данные: подмножество узлов в виде строки из нулей и единиц. Выходные данные: подмножество узлов в виде вектора.
- *void PrintAnswer()* – данный метод предназначен для вывода ответа на задачу в консоль и фиксации времени работы алгоритма.

Также была реализована структура *PathInfo*, которая имеет следующие поля соответственно: имя конечного узла пути, стоимость пути и путь к конечной в вершине в виде вектора.

В самом худшем случае, а это когда на вход подаётся полный граф на 20 вершин, программа работает около 120-ти секунд.

Код программы находится в Приложении А.

Тестирование.

Тестирование программы для графов с 20-ю вершинами приведено в Таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	graph1.txt	[1, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1], 20000, 130464 ms	Ответ для полного графа с 20-ю узлами и одинаковыми весами дуг
2.	graph2.txt	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1], 210, 130776 ms	Ответ для графа-кольца
3.	graph3.txt	[1, 19, 8, 13, 5, 14, 15, 3, 20, 10, 9, 17, 2, 18, 6, 7, 16, 4, 12, 11, 1], 1507, 131922 ms	Ответ для полного графа с 20-ю вершинами и случайными весами дуг

Продолжение Таблицы 1

4.	graph4.txt	Path doesn't exists. 133395ms	Ответ для графа, у которого из одной из вершин не выходит ни одной дуги
5.	graph5.txt	Path doesn't exists. 8699ms	Ответ для графа, у которого все узлы изолированы

Вывод.

В ходе выполнения данной работы изучены принципы динамического программирования и изучено их приложение в виде алгоритма Беллмана-Хелда-Карпа, также повторены асимптотические оценки алгоритмов.

Разработана и протестирована программа на языке C++, считывающая граф из текстового файла и решающая задачу коммивояжера с помощью алгоритма Беллмана-Хелда-Карпа. Данный алгоритм использует принцип динамического программирования: для решения некоторой сложной задачи следует разбить её на более простые подзадачи и решить сначала их. Достоинством данного алгоритма является его скорость: полный перебор и базовый метод ветвей и границ уступают асимптотике алгоритма Беллмана-Хелда-Карпа, равной в данной реализации $O(n^3 2^n \log n)$. В самом худшем случае, когда на вход программа получает полный граф на 20 вершин, программа работает около двух минут, что удовлетворяет ограничению в три минуты.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <sstream>
#include <map>
#include <cmath>
#include <bitset>
#include <numeric>
#include <chrono>
#define MAX_SIZE_OF_SUBSET 19
#define MAX_COST_OF_WAY 100000

struct PathInfo;

using AdjacencyMatrix = std::vector<std::vector<int>>;
using SetInBitForm = std::string;
using Set = std::vector<int>;
using PathsInfo = std::vector<PathInfo>;
using Path = std::vector<int>;

/*
* Данная структура хранит номер узла (endNodeName),
* на котором заканчивается данное подмножетсво узлов
* и стоимость пути, проходящего через данное подмножество узлов
* и заканчивающегося в endNodeName.
*/
struct PathInfo {
    int endNodeName;
    int costOfPath;
    Path path;
};
```

```
class Graph {
public:
    /*
    * Данный метод предназначен для считывания графа из файла.
    * В файле на самом деле хранится матрица смежности,
    * которая в ходе алгоритма переписывается в матрицу
    * смежности, но которая теперь хранится в программе в виде
    * структуры данных
    */
    void ReadGraph() {
        std::string edgeWeight, lineOfInputFile;
        std::vector<int> edgesWeights;
        std::stringstream lineOfInputFileStream;
        std::ifstream inputFile("graph1.txt");
        if (inputFile.is_open())
        {
            while (getline(inputFile, lineOfInputFile))
            {
                lineOfInputFileStream.str(lineOfInputFile);
                while (getline(lineOfInputFileStream, edgeWeight, '
')) {
                    edgesWeights.push_back(std::stoi(edgeWeight));
                }
                graphStorage_.push_back(edgesWeights);
                edgesWeights.clear();
                lineOfInputFileStream.clear();
            }
        }
        inputFile.close();
    }
    /*
    * Данный метод предназначен для получения
    * считанной из файла матрицы смежности.
    * Выходные данные: матрица смежности графа.
    */
    AdjacencyMatrix GetGraphStorage() {
        return graphStorage_;
    }
private:
    AdjacencyMatrix graphStorage_;
};


class TravellingSalesmanProblem {
public:
    /*
    * Конструктор для инициализации матрицы смежности
    * в данном классе считанной матрицей смежности
    * в методе ReadGraph класса Graph.
    */
    TravellingSalesmanProblem(AdjacencyMatrix graphStorage) {
        this->graphStorage_ = graphStorage;
        this->numbOfExistingPathsInSubsets_ = 0;
        this->minCostOfWayInSubset_ = MAX_COST_OF_WAY;
        this->pathWasntFound_ = false;
    }
    /*
    * Данный метод запускает решение TSP с помощью
```

```
        * алгоритма Беллмана-Хелда-Карпа и фиксрует время
        * начала и конца работы алгоритма
        * Запускаются следующие методы: генерация подмножеств,
        * поиск минимального по стоимости гамильтонова цикла,
        * печать ответа на экран
        */
        void Run() {
            startTime_ = std::chrono::steady_clock::now();
            GenerateSubsetsOfNodes();
            FindCheapestHamiltonianCycle();
            PrintAnswer();
        }


    private:
        /*
        * Данный метод генерирует всевозможные подмножества множества
        * узлов {2, 3, .., n} с помощью битовых масок длины n - 1,
        * где если i-ый элемент подмножесва (начиная с 0)
        * равен 1, значит (i + 2)-ый узел входит в данное подмножество,
        * иначе -- не входит.
        * Например: дана маска 1011
        * |i   |0|1|2|3|
        * |mask|1|0|1|1|
        *  Следов-но, в подмножество, которое представляется в виде
вектора,
        * войдёт следующий набор узлов: {2, 4, 5}
        */
        void GenerateSubsetsOfNodes() {
            int subsetPrototype = 1;
            int maxPowerOfSubset = (int)(graphStorage_.size() - 1);
            while (subsetPrototype < (int)pow(2, maxPowerOfSubset)) {
                std::bitset<MAX_SIZE_OF_SUBSET>
dirtySubsetBitForm(subsetPrototype);

costTable_[GetCleanSubset(dirtySubsetBitForm.to_string())] = {};
                subsetPrototype++;
            }
        }
        /*
        * Данный метод представляет собой запуск алгоритма Беллмана-
Хелда-Карпа.
        * На уровне этого метода происходит инициализация информации о
единичных
        * подмножествах и перебор всех подмножеств узлов каждого
размера(мощности).
        */
        void FindCheapestHamiltonianCycle() {
            for (int i = 2; i <= graphStorage_.size(); i++) {
                costTable_[{i}].push_back({ i, graphStorage_[0][i - 1],
{1, i} });
            }
            for (int powerOfSubset = 2; powerOfSubset <
graphStorage_.size(); powerOfSubset++) {
                numbOfExistingPathsInSubsets_ = 0;
                for (auto subsetInfo : costTable_) {
                    if (subsetInfo.first.size() == powerOfSubset) {
```

```
IteratingThroughSubsetWithoutNode(subsetInfo.first);
                    }
                }
                if (!numbOfExistingPathsInSubsets_) {
                    pathWasntFound_ = true;
                    return;
                }
            }
            numbOfExistingPathsInSubsets_ = 0;
            Set subset(graphStorage_.size() - 1);
            std::iota(subset.begin(), subset.end(), 2);
            FindMinCostOfPathInSubset(subset, 1);
            if (!numbOfExistingPathsInSubsets_) {
                pathWasntFound_ = true;
            }
            else {
                minCostOfHamiltonianCycle_ = minCostOfWayInSubset_;
                nodesOfMinCostedHamiltonianCycle_                    =
nodesOfMinCostedWayInSubset_;
                nodesOfMinCostedHamiltonianCycle_.push_back(1);
            }
        }
        /*
        * Данный метод предназначен для перебора всех подмножеств
        * S\{k}, где k -- узел, принадлежащий множеству S.
        * Входные данные: подмножество в виде вектора
        */
        void IteratingThroughSubsetWithoutNode(Set subset) {
            PathInfo pathInfo{ 0, 0, {} };
            Set subsetWithoutOneNode;
            for (int i = 0; i < subset.size(); i++) {
                subsetWithoutOneNode = subset;
                subsetWithoutOneNode.erase(subsetWithoutOneNode.begin()
+ i);
                FindMinCostOfPathInSubset(subsetWithoutOneNode,
subset[i]);
                nodesOfMinCostedWayInSubset_.push_back(subset[i]);
                pathInfo.endNodeName = subset[i];
                pathInfo.costOfPath = minCostOfWayInSubset_;
                pathInfo.path = nodesOfMinCostedWayInSubset_;
                costTable_[subset].push_back(pathInfo);
            }
        }
        /*
        * Данный метод предназначен для поиска минимального по стоимости
пути
        * в данном подмножестве
        * Входные данные: подмножество в виде вектора и узел,
        * в котором заканчивается путь множества.
        */
        void FindMinCostOfPathInSubset(Set &subset, int nodeName) {
            nodesOfMinCostedWayInSubset_.clear();
            minCostOfWayInSubset_ = MAX_COST_OF_WAY;
            for (auto const& pathInfo : costTable_[subset]) {
                if                (pathInfo.costOfPath                +
graphStorage_[pathInfo.endNodeName     -     1][nodeName     -     1]     <
minCostOfWayInSubset_) {
```

```
                    minCostOfWayInSubset_ = pathInfo.costOfPath +
                        graphStorage_[pathInfo.endNodeName - 1][nodeName
- 1];
                    nodesOfMinCostedWayInSubset_ = pathInfo.path;
                }
            }
            if (minCostOfWayInSubset_ != MAX_COST_OF_WAY) {
                numbOfExistingPathsInSubsets_++;
            }
        }
        /*
        * Данный метод предназначен для очистки битового
        * представления подмножеств от лишних нулей,
        * которые появились в силу перевода в двоичную
        * систему счисления с помощью std::bitset и интерпретации
        * таких подмножеств в виде вектора.
        */
        Set GetCleanSubset(SetInBitForm subsetInBitForm) {
            SetInBitForm             cleanSubsetInBitForm             =
subsetInBitForm.substr(subsetInBitForm.size() - graphStorage_.size() + 1,
                graphStorage_.size() - 1);
            Set cleanSubset;
            for (int i = 0; i < cleanSubsetInBitForm.size(); i++) {
                if (cleanSubsetInBitForm[i] == '1') {
                    cleanSubset.push_back(i + 2);
                }
            }
            return cleanSubset;
        }
        /*
        * Данный метод предназначен для печати ответа на экран
        */
        void PrintAnswer() {
            auto endTime = std::chrono::steady_clock::now();
            auto                        spentTime                        =
std::chrono::duration_cast<std::chrono::milliseconds>(endTime           -
startTime_);
            if (pathWasntFound_) {
                std::cout  <<  "Path  doesn't  exists."  <<  "  "  <<
spentTime.count() << "ms" << std::endl;
            }
            else {
                std::cout << "[";
                int                         lenOfCycle                       =
(int)nodesOfMinCostedHamiltonianCycle_.size();
                for (int i = 0; i < lenOfCycle - 1; i++) {
                    std::cout << nodesOfMinCostedHamiltonianCycle_[i] <<
", ";
                }
                std::cout                                                  <<
nodesOfMinCostedHamiltonianCycle_[lenOfCycle - 1] << "], ";
                std::cout << minCostOfHamiltonianCycle_ << ", ";
                std::cout << spentTime.count() << " ms" << std::endl;
            }
        }

        AdjacencyMatrix graphStorage_;
        std::map<Set, PathsInfo> costTable_;
```

```
        int numbOfExistingPathsInSubsets_;
        Path                          nodesOfMinCostedHamiltonianCycle_,
nodesOfMinCostedWayInSubset_;
        int minCostOfHamiltonianCycle_, minCostOfWayInSubset_;
        bool pathWasntFound_;
        std::chrono::steady_clock::time_point startTime_;
    };
    int main() {
        Graph graph;
        graph.ReadGraph();
        TravellingSalesmanProblem tsp(graph.GetGraphStorage());
        tsp.Run();
        return 0;
    }
```