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: Кратчайшие пути в графе: коммивояжёр.

Студентка гр. 1383 _____ Федорова О.В.

Преподаватель _____ Токарев А. П.

Санкт-Петербург
2023

Цель работы.

Изучить принцип работы алгоритмов нахождения пути коммивояжера на графах. Решить с их помощью задачу.

Задание.

Решить задачу коммивояжера

Дано: матрица весов графа, все веса неотрицательны; стартовая вершина.

Найти: путь коммивояжёра (последовательность вершин) и его стоимость.

Выполнение работы.

Была реализована структура State, для хранения пути коммивояжера, стартовой вершины, информации о посвященных вершинах и вектор оценки стоимости.

Работа программы начинается с вызова функции test, с передачей потока для вывода и названия файла, с которого необходимо считать данные.

С файла считываются число n - размер графа, n*n вершин - элементы матрицы смежности графа, число - номер стартовой вершины.

После считывания, созданный двумерный массив - граф, стартовая вершина и поток вывода передаются в функцию get_best_solution.

В этой функции создается состояние State solve - первое полученное решение, необходимое для получения нижней оценки для работы МВГ. Данное состояние инициализируется при передаче в функцию SolveTSP в которой происходит вызов функции get_lower_bound, в которой ищется первый путь коммивояжера, записывается в переданное состояние, после чего инициализируется массив с нижними оценками для каждой вершины. Принцип работы поиска первого решения задачи : обходится граф, если на текущей вершине некуда идти, то вершина помечается не посещенной, удаляется из пути и в матрице смежности путь из предыдущей вершины до

данной инициализируется бесконечностью, чтобы не совершить снова переход в нее, если же есть куда идти - вершина помещается в путь, помечается посещенной и цикл поиска пути продолжается до тех пор, пока размер пути коммивояжера не равен n + 1 (замкнутый). Для обеспечения замкнутости пути, в матрицу смежности последним столбцом скопирован стартовый столбец и вставлена строка с бесконечностями в конец. То есть возвращение в стартовую вершину будет обязательно, так как алгоритм будет работать до тех пор, пока все вершины не будут посещены, при этом при переходе происходит дополнительная проверка : если переход возможен только в последнюю вершину(по факту это стартовая), но при этом еще не все вершины посещены, то переход невозможен.

После получения первого решения, создается двумерный массив для хранения решений ans, который передается в функцию curr_branch, который обходит граф, переходя из стартовой вершины во все возможные, но переход происходит лишь при условии выгодности - если уже найденный первый путь на данном шаге был более выгодный. Для получения оценки i-го шага следует обратиться к полю state.lower_bounds[path[i]], path[i] вернет символ на i-м шаге, а в массиве lower_bounds в порядке вершин лежат стоимости всего пути, которая была в оценочном пути.

Если в curr_branch размер пути текущего состояния уже достиг размера графа, то происходит проверка перехода из текущей вершины в стартовую, если такой есть, то состояние дополняется данным путем и весь путь записывается в ответ, если нет, то ветвь заканчивает свое существование.

Для тестирования был написан файл test.cpp ,в котором ответ, выданный на двух файлах input.txt и input2.txt сравнивается с заранее заготовленным ответом в файле check.txt. Если содержимое файла check.txt равняется содержимому файла answer.txt, то заготовленные тесты пройдены успешно и выводится Passed.

Выводы.

Был изучен принцип работы алгоритмов нахождения пути коммивояжера на графах. С помощью МВГ решена задача. Программа была протестирована, результаты тестов корректны.

Приложение А

Исходный код программы

```
        #include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <fstream>

using namespace std;

const int INF = INT_MAX; // Бесконечность

// Структура для хранения информации о состоянии поиска
struct State {
    int vertex;
    vector<int> path;
    vector<bool> visited;
    vector<int> lower_bounds;
    int cost;
};
void get_lower_bounds(vector<vector<int>> graph, State& state) {
    vector <int>lst;
    for(int i = 0; i < graph.size(); i++) {
        graph[i].push_back(graph[i][state.vertex]);
        lst.push_back(INF);
    }
    graph.push_back(lst);

    int curr = state.vertex;//стартовя вершина
    while (state.path.size() < graph.size()) {
        curr = state.path[state.path.size() - 1];
        int nxt = -1;
        for(int i = 0; i < graph.size(); i++) {
            if(graph[curr][i] != INF && !state.visited[i] &&
                    (i != graph.size() - 1 || state.path.size() ==
graph.size() - 1)) {
                nxt = i;
                break;
            }
        }
        if(nxt != -1 && nxt != curr) {
            state.cost += graph[curr][nxt];
            state.path.push_back(nxt);
            //state.visited[nxt] = true;
            state.visited[curr] = true;
        } else {

            state.visited[curr] = false;
            state.path.pop_back();
            graph[state.path[state.path.size() - 1]][curr] = INF;
            //state.path.pop_back();
        }
    }
}
int lowerBound(vector<vector<int>>& graph, State& state);
```

```
// Функция для решения задачи коммивояжера с использованием метода
ветвей и границ
State solveTSP(vector<vector<int>>& graph, int start) {
    int n = graph.size(); // Количество вершин в графе
    // Инициализация начального состояния
    State initial;
    initial.vertex = start;
    initial.path.push_back(start);
    initial.visited = vector<bool>(n, false);
    initial.visited[start] = true;
    initial.lower_bounds = vector<int>(n, -1);
    initial.cost = 0;
    get_lower_bounds(graph, initial);
    //count cost
    int cst = 0;
    for(int i = 0; i < initial.path.size() - 2; i++) {
        cst+=graph[initial.path[i]][initial.path[i + 1]];
        initial.lower_bounds[initial.path[i + 1]] = cst;
    }
    cst+=graph[initial.path[initial.path.size() - 2]][initial.vertex];
    initial.lower_bounds[start] = cst;
    vector <int> ans = initial.path;
    ans.pop_back();
    ans.push_back(initial.vertex);
    ans.push_back(cst);
    initial.path = ans;
    return initial;
}

// Функция для вычисления нижней границы стоимости пути в текущем
состоянии
int lowerBound(int curr, State & state) {
    return state.lower_bounds[curr];
}
void curr_branch(vector<vector <int>> & ans, vector <vector <int>>
grapth, int curr, State curr_state, State lower_state, int cst) {
    if(cst < 0) return;
    if(curr_state.path.size() == grapth.size()) {
        if(grapth[curr][curr_state.vertex] != INF &&
grapth[curr][curr_state.vertex] + cst < lowerBound(curr_state.vertex,
lower_state)) {
            vector <int> path = curr_state.path;
            path.push_back(curr_state.vertex);
            path.push_back(cst + grapth[curr][curr_state.vertex]);
            ans.push_back( path);
            return;
        }
    }
    for(int i = 0; i < grapth[curr].size(); i++) {
        if(grapth[curr][i] + cst < lowerBound(i, lower_state) &&
!curr_state.visited[i]) {
            curr_state.visited[i] = true;
            curr_state.path.push_back(i);
            curr_branch(ans, grapth, i, curr_state, lower_state, cst +
grapth[curr][i]);
        }
    }
}
```

```
void get_best_solution(vector<vector <int>> & graph, int start,
std::ostream & out) {
    State solve = solveTSP(graph, start);
    int cst = 0;
    int curr = start;
    State initial;
    initial.vertex = start;
    initial.path.push_back(start);
    initial.visited = vector<bool>(graph.size(), false);
    initial.visited[start] = true;
    initial.cost = 0;
    vector <vector <int>> ans;
    ans.push_back(solve.path);
    curr_branch(ans, graph, start, initial, solve, 0);
    int mincst = solve.lower_bounds[start];
    int minind = 0;
    for(int i = 0; i < ans.size(); i++) {
        if(!ans[i].empty() && ans[i].back() > 0 && mincst >
ans[i].back()) {
            mincst = ans[i].back();
            minind = i;
        }
    }
    for(int i = 0; i < ans[minind].size() - 1; i++)
        out<<ans[minind][i]<<" ";
    out<<endl<<mincst<<endl;
}
void test(string file_name, std::ostream & out) {
    ifstream file_cin;
    file_cin.open(file_name);
    int n; // Количество вершин в графе
    //cout << "Введите количество вершин: ";
    file_cin >> n;

    // Создаем граф в виде матрицы смежности
    vector<vector<int>> graph(n, vector<int>(n, 0));
    //cout << "Введите матрицу смежности:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            file_cin >> graph[i][j];
            if (graph[i][j] == 0) {
                graph[i][j] = INF;
            }
        }
    }

    int start; // Начальная вершина
    //cout << "Введите начальную вершину: ";
    file_cin >> start;
    get_best_solution(graph, start, out);
    file_cin.close();
}

int main() {
    test("modules/input2.txt", cout);
    return 0;
}
```