Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації та управління

ЗВІТ

про виконання лабораторного практикуму №4(2)

Виконав:

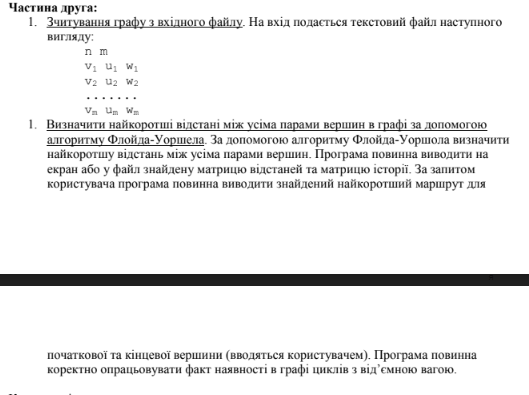
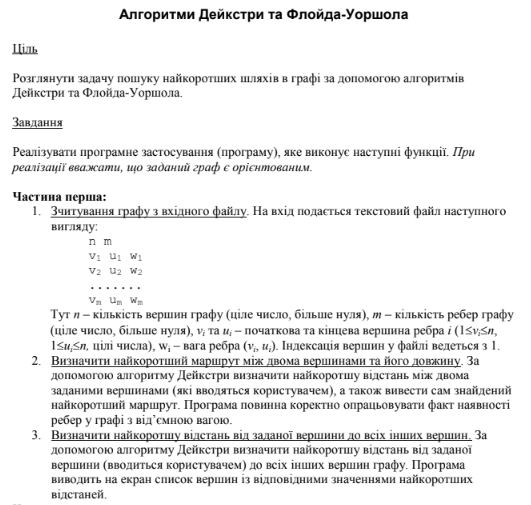
студент 1- го курсу ФІОТ

групи *ІП-91*

*Кінчур Вадим Вікторович*

Київ 2020

2. Умова лабораторної роботи



3. Програмний код (С++):

#include <string>

#include <ostream>

#include <list>

#include <queue>

#include <vector>

#include <iostream>

#include <iomanip>

#include <ostream>

#include <fstream>

const int INF = 999999999;

struct edge {

int destination;

int weight;

edge(int destination, int weight) {

this->destination = destination;

this->weight = weight;

}

};

class Graph

{

private:

std::list<edge>\* adj\_list;

int vertex\_number;

int\*\* adj\_matrix;

void fill\_adj\_matrix();

public:

Graph(int);

Graph(Graph&);

~Graph();

void add\_edge(int, int, int);

int\*\* get\_adj\_matrix();

bool contains\_negative\_edges();

friend class Algorithm;

};

struct dijkstra\_result {

int\* path;

int\* dist;

};

struct floyd\_result {

int\*\* dist\_matrix;

int\*\* history\_matrix;

};

class Algorithm {

private:

static std::ostream\* out;

static dijkstra\_result& dijkstra(Graph&, int);

static floyd\_result floyd(Graph&);

static void print\_way\_between(dijkstra\_result, int, int);

static bool contains\_negative\_cycles(int, floyd\_result);

static void show\_way\_floyd(floyd\_result, int, int);

static void print\_way(Graph&, int, int);

static void print\_all\_ways(Graph&, int);

static void print\_floyd(Graph&);

static void show\_menu();

static void do\_work(Graph& graph);

template<typename T>

static void print\_matrix(T\*\*, int, int);

public:

static void start\_menu(Graph&);

};

int main() {

std::ifstream file\_handler("input.txt");

int n, m;

file\_handler >> n >> m;

Graph graph(n);

for (auto i = 0; i < m; ++i) {

int start, end, weight;

file\_handler >> start >> end >> weight;

graph.add\_edge(start, end, weight);

}

file\_handler.close();

Algorithm::start\_menu(graph);

}

Graph::Graph(int vertex\_number) {

this->vertex\_number = vertex\_number;

fill\_adj\_matrix();

adj\_list = new std::list<edge>[vertex\_number];

}

Graph::Graph(Graph& graph) {

vertex\_number = graph.vertex\_number;

fill\_adj\_matrix();

adj\_list = new std::list<edge>[vertex\_number];

}

Graph::~Graph() {

delete[] adj\_list;

for (auto i = 0; i < vertex\_number; ++i)

delete[] adj\_matrix[i];

delete[] adj\_matrix;

}

void Graph::add\_edge(int start, int end, int weight) {

adj\_list[start - 1].push\_back(edge(end - 1, weight));

adj\_matrix[start - 1][end - 1] = weight;

}

int\*\* Graph::get\_adj\_matrix() {

return adj\_matrix;

}

bool Graph::contains\_negative\_edges()

{

for (auto i = 0; i < vertex\_number; ++i) {

for (auto el : adj\_list[i]) {

if (el.weight < 0)

return true;

}

}

return false;

}

void Graph::fill\_adj\_matrix() {

adj\_matrix = new int\* [vertex\_number];

for (auto i = 0; i < vertex\_number; ++i)

adj\_matrix[i] = new int[vertex\_number];

for (auto i = 0; i < vertex\_number; ++i) {

for (auto j = 0; j < vertex\_number; ++j) {

adj\_matrix[i][j] = INF;

}

}

}

struct compare {

bool operator()(edge& a, edge& b) const {

return a.weight > b.weight;

}

};

std::ostream\* Algorithm::out;

void Algorithm::start\_menu(Graph& graph) {

std::cout << "Do you want to work with file or console?(file/console)\n";

std::string answer;

std::cin >> answer;

std::ofstream file;

if (answer == "file") {

std::cout << "Type the name of the file :\n";

std::string file\_name;

std::cin >> file\_name;

file.open(file\_name, std::ios::out);

out = &file;

}

else out = &std::cout;

while (answer != "exit") {

show\_menu();

do\_work(graph);

std::cout << "If you to exit print \'exit\' else press any key ...\n";

std::cin >> answer;

}

file.close();

}

dijkstra\_result& Algorithm::dijkstra(Graph& graph, int start)

{

dijkstra\_result result;

int\* dist = new int[graph.vertex\_number];

int\* path = new int[graph.vertex\_number];

std::priority\_queue<edge, std::vector<edge>, compare> que;

for (auto i = 0; i < graph.vertex\_number; ++i) {

dist[i] = INF;

path[i] = -1;

}

que.push(edge(start, 0));

dist[start] = 0;

while (!que.empty()) {

auto u = que.top().destination; que.pop();

for (auto& el : graph.adj\_list[u]) {

if (dist[el.destination] > dist[u] + el.weight) {

dist[el.destination] = dist[u] + el.weight;

path[el.destination] = u;

que.push(el);

}

}

}

result.dist = dist;

result.path = path;

return result;

}

template<typename T>

T\*\* fill\_matrix(int n, int m, T filler) {

T\*\* matrix = new T \* [n];

for (auto i = 0; i < n; ++i)

matrix[i] = new T[m];

for (auto i = 0; i < n; ++i) {

for (auto j = 0; j < m; ++j) {

matrix[i][j] = filler;

}

}

return matrix;

}

floyd\_result Algorithm::floyd(Graph& graph)

{

auto n = graph.vertex\_number;

auto\*\* dist = fill\_matrix(n, n, INF);

auto\*\* history = fill\_matrix(n, n, -1);

auto\*\* matrix = graph.get\_adj\_matrix();

floyd\_result result;

for (auto i = 0; i < n; ++i) {

for (auto j = 0; j < n; ++j) {

if (i == j) {

dist[i][j] = 0;

history[i][i] = 0;

}

else {

dist[i][j] = matrix[i][j];

history[i][j] = i + 1;

}

}

}

for (auto k = 0; k < n; ++k) {

for (auto i = 0; i < n; ++i) {

for (auto j = 0; j < n; ++j) {

if (dist[k][k] < 0) {

dist[k][k] = -INF;

result.dist\_matrix = dist;

result.history\_matrix = history;

return result;

}

if (dist[i][j] > dist[i][k] + dist[k][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

history[i][j] = history[k][j];

}

}

}

}

result.dist\_matrix = dist;

result.history\_matrix = history;

return result;

}

void Algorithm::print\_way\_between(dijkstra\_result info, int start, int end) {

std::vector<int> path;

auto current = end - 1;

while (current != -1) {

path.push\_back(current + 1);

current = info.path[current];

}

std::reverse(path.begin(), path.end());

for (auto i = 0; i < path.size(); ++i) {

if (i == path.size() - 1)

\* out << path[i];

else

\*out << path[i] << "->";

}

}

bool Algorithm::contains\_negative\_cycles(int n, floyd\_result info) {

for (auto i = 0; i < n; ++i)

if (info.dist\_matrix[i][i] == -INF)

return true;

return false;

}

void Algorithm::show\_way\_floyd(floyd\_result info, int start, int end) {

std::vector<int> path;

auto current = end;

while (current != start) {

path.push\_back(current);

current = info.history\_matrix[start - 1][current - 1];

};

path.push\_back(info.history\_matrix[start - 1][current]);

std::reverse(path.begin(), path.end());

for (auto i = 0; i < path.size(); ++i) {

if (i == path.size() - 1)

\* out << path[i];

else

\*out << path[i] << "->";

}

\*out << '\n';

}

void Algorithm::print\_way(Graph& graph, int start, int end) {

if (graph.contains\_negative\_edges()) {

\*out << "Graph containts negative edges! Using Dijkstra algorithm is impossible!\n";

return;

}

auto info = dijkstra(graph, start - 1);

if (info.dist[end - 1] != INF) {

\*out << "Distance between vertices is equal to " << info.dist[end - 1] << '\n';

print\_way\_between(info, start, end);

}

delete[] info.path;

delete[] info.dist;

}

void Algorithm::print\_all\_ways(Graph& graph, int start) {

if (graph.contains\_negative\_edges()) {

\*out << "\nGraph containts negative edges! Using Dijkstra algorithm is impossible!\n";

return;

}

auto info = dijkstra(graph, start - 1);

for (auto i = 0; i < graph.vertex\_number; ++i) {

if (i != start - 1) {

\*out << "Shortest way from " << start << " to " << i + 1 << " is equal to ";

if (info.dist[i] == INF)

\* out << "infinity\n";

else

\*out << info.dist[i] << '\n';

}

}

delete[] info.path;

delete[] info.dist;

}

template<typename T>

void Algorithm::print\_matrix(T\*\* matrix, int n, int m) {

for (auto i = 0; i < n; ++i) {

for (auto j = 0; j < m; ++j) {

if (matrix[i][j] != INF)

\* out << std::setw(5) << matrix[i][j];

else

\*out << std::setw(5) << "?";

}

\*out << '\n';

}

}

template<typename T>

void delete\_matrix(T\*\* matrix, int n) {

for (auto i = 0; i < n; ++i)

delete[] matrix[i];

delete[] matrix;

}

void Algorithm::print\_floyd(Graph& graph) {

floyd\_result info = floyd(graph);

if (contains\_negative\_cycles(graph.vertex\_number, info)) {

\*out << "\nGraph contains negative cycle!\n";

return;

}

\*out << "Distance matrix : \n";

print\_matrix(info.dist\_matrix, graph.vertex\_number, graph.vertex\_number);

\*out << "History matrix : \n";

print\_matrix(info.history\_matrix, graph.vertex\_number, graph.vertex\_number);

std::cout << "\nPrint vertices to find shortest way between for\n";

int start, end;

std::cin >> start >> end;

if (info.dist\_matrix[start - 1][end - 1] != INF) {

\*out << "\nThe distance is equal to " << info.dist\_matrix[start - 1][end - 1] << '\n';

show\_way\_floyd(info, start, end);

}

else

\*out << "\nPath between vertices doesn`t exist!\n";

delete\_matrix(info.dist\_matrix, graph.vertex\_number);

delete\_matrix(info.history\_matrix, graph.vertex\_number);

}

void Algorithm::show\_menu() {

std::cout << "Choose one of the next options : \n";

std::cout << "1 - Find shortest path between two vertices with Dijkstra algorithm\n"

<< "2 - Find shortest paths from source to other vertices with Dijkstra algorithm\n"

<< "3 - Show shortest paths between all vertices with Floyd-Warshall algorithm\n";

}

void Algorithm::do\_work(Graph& graph) {

int input = -1, start, end;

while (input < 1 || input > 3) {

std::cout << "\nType your choice : ";

std::cin >> input;

}

switch (input)

{

case 1:

std::cout << "\nType start and end vertices : \n";

std::cin >> start >> end;

print\_way(graph, start, end);

\*out << "\n\n";

break;

case 2:

std::cout << "\nType start vertex : \n";

std::cin >> start;

print\_all\_ways(graph, start);

\*out << "\n\n";

break;

case 3:

print\_floyd(graph);

\*out << "\n\n";

break;

}

}

4. Результати виконання

**Вхідні дані:**

4 8

1 2 -2

1 3 3

1 4 -3

2 3 2

3 4 5

4 1 4

4 2 5

4 3 5

