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:

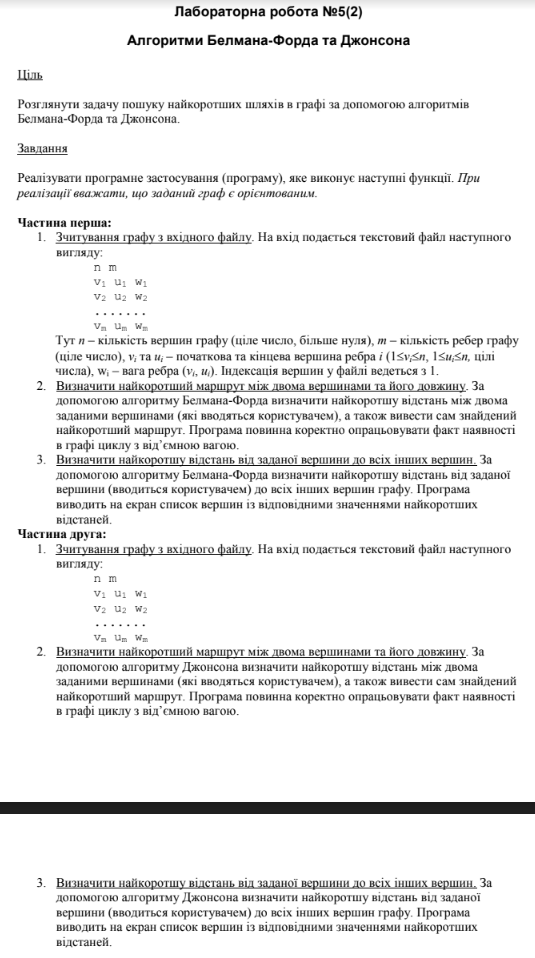
студент 1- го курсу ФІОТ

групи *ІП-91*

*Кінчур Вадим Вікторович*

Київ 2020

2. Умова лабораторної роботи



4. Програмний код (С++):

#include <ostream>

#include <iostream>

#include <fstream>

#include <list>

#include <iostream>

#include <iomanip>

#include <queue>

#include <fstream>

const int INF = INT16\_MAX;

struct dest\_edge {

int destination;

int weight;

dest\_edge(int destination, int weight) {

this->destination = destination;

this->weight = weight;

}

};

struct edge {

int start, end, weight;

edge(int start, int end, int weight) {

this->start = start;

this->end = end;

this->weight = weight;

}

};

class Graph

{

private:

std::list<dest\_edge>\* dest\_adj\_list;

std::list<edge> adj\_list;

int vertex\_number;

int\*\* adj\_matrix;

void fill\_adj\_matrix();

public:

Graph(int);

Graph(Graph&);

~Graph();

void rebuilt();

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

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

bool contains\_negative\_edges();

friend class Algorithm;

};

struct bellman\_result {

int\* dist = nullptr;

int\* path = nullptr;

};

struct johnson\_result {

int\*\* dist\_m = nullptr;

int\*\* path\_m = nullptr;

};

struct dijkstra\_result {

int\* dist = nullptr;

int\* path = nullptr;

};

class Algorithm {

private:

static std::ostream\* out;

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

static void print\_path(int\*, int, int);

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

static johnson\_result johnson(Graph&);

static void print\_path\_between(Graph&, int, int);

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

static void print\_johnson\_between(Graph&, int, int);

static void print\_all\_johnson\_paths(Graph&);

template<typename T>

static void delete\_matrix(T\*\*, const int);

template<typename T>

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

static void do\_work(Graph&);

static void show\_menu();

public:

static void start\_menu(Graph&);

};

int main() {

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

int vertex\_number, edges\_number;

file\_handler >> vertex\_number >> edges\_number;

Graph graph(vertex\_number);

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

dest\_adj\_list = new std::list<dest\_edge>[vertex\_number];

}

Graph::Graph(Graph& graph) {

vertex\_number = graph.vertex\_number;

fill\_adj\_matrix();

dest\_adj\_list = new std::list<dest\_edge>[vertex\_number];

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

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

if (graph.adj\_matrix[i][j] != INF) {

add\_edge(i + 1, j + 1, adj\_matrix[i][j]);

}

}

}

}

Graph::~Graph() {

delete[] dest\_adj\_list;

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

delete[] adj\_matrix[i];

delete[] adj\_matrix;

}

void Graph::rebuilt() {

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

dest\_adj\_list[i].clear();

}

for (auto& el : adj\_list) {

if (el.start < vertex\_number) {

dest\_adj\_list[el.start].push\_back(dest\_edge(el.end, el.weight));

adj\_matrix[el.start][el.end] = el.weight;

}

}

}

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

dest\_adj\_list[start - 1].push\_back(dest\_edge(end - 1, weight));

adj\_list.push\_back(edge(start - 1, 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 el : adj\_list) {

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;

}

}

}

std::ostream\* Algorithm::out;

template<typename T>

void Algorithm::print\_matrix(T\*\* matrix, const int row\_size, const int col\_size) {

for (auto i = 0; i < row\_size; ++i) {

for (auto j = 0; j < col\_size; ++j) {

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

\* out << std::setw(4) << matrix[i][j];

else

\*out << std::setw(4) << '?';

}

\*out << '\n';

}

}

template<typename T>

void Algorithm::delete\_matrix(T\*\* matrix, const int row\_size) {

for (auto i = 0; i < row\_size; ++i) delete[] matrix[i];

delete[] matrix;

}

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

{

bellman\_result result;

const int n = graph.vertex\_number;

int\* dist = new int[n];

int\* prev = new int[n];

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

dist[i] = INF;

prev[i] = -1;

}

dist[start] = 0;

for (auto i = 0; i < n - 1; ++i) {

for (auto& edge\_ : graph.adj\_list) {

int u = edge\_.start, v = edge\_.end;

if (dist[v] > dist[u] + edge\_.weight && dist[u] != INF) {

dist[v] = dist[u] + edge\_.weight;

prev[v] = u;

}

}

}

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

if (dist[el.end] != INF && dist[el.end] > dist[el.start] + el.weight) {

delete[] dist;

delete[] prev;

return result;

}

}

result.dist = dist;

result.path = prev;

return result;

}

void Algorithm::print\_path(int\* path, int start, int end) {

auto current = end - 1;

std::list<int> way;

while (current != start - 1) {

way.push\_front(current + 1);

current = path[current];

}

way.push\_front(start);

for (auto el : way) {

if (el != way.back())\* out << el << "->";

else \*out << el;

}

}

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

auto result = bellman(graph, start - 1);

if (result.dist == nullptr) {

\*out << "Graph contains negative cycles!\n";

return;

}

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

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

print\_path(result.path, start, end);

}

else {

\*out << "The path doesn`t exist!\n";

}

delete[] result.dist;

delete[] result.path;

}

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

auto result = bellman(graph, start - 1);

if (result.dist == nullptr) {

\*out << "Graph contains negative cycles!\n";

return;

}

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

if (i != start - 1) {

if (result.dist[i] == INF) {

\*out << "The distance between " << start << " and " << i + 1 << " is equal to infinity\n";

}

else {

\*out << "The distance between " << start << " and " << i + 1 << " is equal to "

<< result.dist[i] << '\n';

}

}

}

delete[] result.dist;

delete[] result.path;

}

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

{

struct compare {

bool operator()(dest\_edge& a, dest\_edge& b) const {

return a.weight > b.weight;

}

};

dijkstra\_result result;

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

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

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

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

dist[i] = INF;

path[i] = -1;

}

que.push(dest\_edge(start, 0));

dist[start] = 0;

while (!que.empty()) {

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

for (auto& el : graph.dest\_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;

}

johnson\_result Algorithm::johnson(Graph& graph) {

Graph new\_graph(graph.vertex\_number + 1);

for (auto& el : graph.adj\_list)

new\_graph.add\_edge(el.start + 1, el.end + 1, el.weight);

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

new\_graph.add\_edge(graph.vertex\_number + 1, i + 1, 0);

const int n = graph.vertex\_number;

johnson\_result res;

int\*\* dist\_m = new int\* [n];

int\*\* path\_m = new int\* [n];

auto bellman\_res = bellman(new\_graph, n);

if (bellman\_res.dist == nullptr) {

delete[] dist\_m;

delete[] path\_m;

return res;

}

auto h = bellman\_res.dist;

--new\_graph.vertex\_number;

for (auto& el : new\_graph.adj\_list) {

if (el.start != n)

{

el.weight += h[el.start] - h[el.end];

}

}

new\_graph.rebuilt();

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

auto result = dijkstra(new\_graph, i);

dist\_m[i] = result.dist;

path\_m[i] = result.path;

}

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

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

if (dist\_m[i][j] != INF) {

dist\_m[i][j] += h[j] - h[i];

}

}

}

res.dist\_m = dist\_m;

res.path\_m = path\_m;

return res;

}

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

auto result = johnson(graph);

if (result.dist\_m == nullptr) {

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

return;

}

const int n = graph.vertex\_number;

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

\*out << "The distance between vertices is equal to " << result.dist\_m[start - 1][end - 1] << '\n';

print\_path(result.path\_m[start - 1], start, end);

}

else {

\*out << "The path doesn`t exist!\n";

}

delete\_matrix(result.dist\_m, n);

delete\_matrix(result.path\_m, n);

}

void Algorithm::print\_all\_johnson\_paths(Graph& graph) {

auto result = johnson(graph);

if (result.dist\_m == nullptr) {

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

return;

}

const int n = graph.vertex\_number;

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

print\_matrix(result.dist\_m, n, n);

delete\_matrix(result.path\_m, n);

delete\_matrix(result.dist\_m, n);

}

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

int input = -1, start, end;

while (input < 1 || input > 4) {

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

std::cin >> input;

}

switch (input)

{

case 1:

std::cout << "Print vertices to find path for : ";

std::cin >> start >> end;

print\_path\_between(graph, start, end);

break;

case 2:

std::cout << "Type the start vertex : ";

std::cin >> start;

print\_all\_paths(graph, start);

break;

case 3:

std::cout << "Type vertices to find path for : ";

std::cin >> start >> end;

print\_johnson\_between(graph, start, end);

break;

case 4:

print\_all\_johnson\_paths(graph);

break;

}

}

void Algorithm::show\_menu() {

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

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

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

<< "3 - Find shorted path between vertices with Johnson algorithm\n"

<< "4 - Show shortest paths between all vertices with Johnson - Algorithm algorithm\n";

}

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);

\*out << "\n";

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

std::cin >> answer;

\*out << '\n';

}

file.close();

}

4. Результати виконання:

**Вхідні дані**

4 8

1 2 -2

1 3 3

1 4 -3

2 3 2

3 4 -3

4 1 4

4 2 5

4 3 5

