Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Домашнее задание № 3

ПО КУРСУ

«Алгоритмические языки»

на тему «Объектно ориентированное программирование в языке Си++. Алгоритмы на графах»

Студент ИУ8-23 В. С. Ажгирей

(Группа) (И. О. Фамилия)

М. Ю. Григорьева

Преподаватель: М. В. Малахов

(И.О. Фамилия)

2024

Введение

Вариант 2. Разработайте приложение для поиска кратчайшего пути из одной вершины в другую в ориентированном взвешенном и невзвешенном (в рамках данной задачи будем считать, что вес всех ребер в невзвешенном графе равен единице) графах. Графы читаются из файла, в приложении должен быть функционал по изменению считанных графов (удалению/добавлению вершин и рёбер, изменению веса ребер для взвешенного графа).

В разработанных классах должны быть конструкторы и операторы перемещения и присваивания, а также как минимум одна переопределенная операция, помимо присваивания.

В приложениях должен присутствовать код для обработки возможных исключительных ситуаций (например, ошибка при чтении данных из файла из-за некорректности данных).

Цели и задачи:

- изучить поставленную задачу
- изучить основы теории графов
- подобрать необходимый алгоритм на графах для текущей задачи
- написать код программы на языке C++;
- разработать тестовые примеры и отладить программу;
- рассмотреть исключительные ситуации и разработать методы их обработки
- подготовить отчет по лабораторной работе.

Основная часть

Исходный текст программы:

Файл заголовка mainwindow.h:

```
#pragma once
#include <vector>
#include <string>
#include <algorithm>
#include <fstream>
#include <iostream>


class Graph {
private:
	std::vector<std::vector<int>> graph;
	size_t size;
	bool oriented;

	std::vector<bool> visited;
	std::vector<int> distances;
	size_t start;

public:
	Graph();
	Graph(std::vector<std::vector<int>>);
	Graph(size_t, bool, std::vector<std::vector<int>>);
	Graph(const Graph&);
	Graph(Graph&&);

	void set_size(size_t);
	void set_oriented(bool);
	void set_graph(std::vector<std::vector<int>>);
	void set(size_t, bool, std::vector<std::vector<int>>);

	size_t get_size();
	bool get_oriented();
	std::vector<std::vector<int>> get_graph();

	bool is_valid();
	void add_vertex();
	void delete_vertex(size_t);
	void add_edge(size_t, size_t, int);
	void delete_edge(size_t, size_t);

	void dijkstra();
	void prepare_dijkstra(size_t);
	int get_min_distance(size_t, size_t);

	friend void read_data(Graph&, std::string);
	friend std::ostream& operator << (std::ostream&, const Graph&);
	friend std::istream& operator >> (std::istream&, Graph&);
};

std::vector<int> split(const std::string&);

void read_data(Graph&, std::string);

template <typename T>
std::ostream& operator << (std::ostream&, const std::vector<T>);
```

```
std::ostream& operator << (std::ostream&, const Graph&);

std::istream& operator >> (std::istream&, Graph&);
```

Файл описания graph.cpp:

```
#include "graph.hpp"

Graph::Graph()
{
	size = 0;
	oriented = false;
}

Graph::Graph(std::vector<std::vector<int>> graph)
{
	this->graph = graph;
	size = graph.size();
	oriented = false;
}

Graph::Graph(size_t size, bool oriented, std::vector<std::vector<int>> graph)
{
	this->size = size;
	this->oriented = oriented;
	this->graph = graph;
}

Graph::Graph(const Graph& oth)
{
	size = oth.size;
	oriented = oth.oriented;
	graph = oth.graph;
}

Graph::Graph(Graph&& oth)
{
	size = oth.size;
	oth.size = 0;
	oriented = oth.oriented;
	oth.oriented = false;
	graph = oth.graph;
	oth.graph.clear();
}

void Graph::set_size(size_t size)
{
	this->size = size;
}

void Graph::set_oriented(bool oriented)
{
	this->oriented = oriented;
}

void Graph::set_graph(std::vector<std::vector<int>> graph)
{
	this->graph = graph;
}
```

```cpp
void Graph::set(size_t size, bool oriented, std::vector<std::vector<int>> graph)
{
	this->size = size;
	this->oriented = oriented;
	this->graph = graph;
}

size_t Graph::get_size()
{
	return size;
}

bool Graph::get_oriented()
{
	return oriented;
}

std::vector<std::vector<int>> Graph::get_graph()
{
	return graph;
}

bool Graph::is_valid()
{
	if (graph.size() != size) {
		std::cout << graph.size();
		throw std::runtime_error("Invalid size " + std::to_string(size));
	}
	if (!std::all_of(graph.begin(), graph.end(), [this](std::vector<int> line)
		{
		return size == line.size();
		}))
		throw std::runtime_error("Invalid size");
	if (!oriented)
	{
		for (size_t i = 0; i < size; i++)
			for (size_t j = 0; j < size; j++) {
				if (graph[i][j] < 0)
					throw std::runtime_error("Invalid distance");
				if (graph[i][j] != graph[j][i])
					throw std::runtime_error("Invalid matrix");
			}
	}
	return true;
}

void Graph::add_vertex()
{
	size++;
	graph.push_back(std::vector<int>(size, 0));
	for (size_t i = 0; i < size - 1; i++)
		graph[i].resize(size);
}

void Graph::delete_vertex(size_t index)
{
	for (size_t i = 0; i < size; i++)
		graph[i].erase(graph[i].begin() + index);
	graph.erase(graph.begin() + index);
	size--;
}

void Graph::add_edge(size_t start, size_t end, int weight = 1)
{
```

```
	graph[start][end] = weight;
	if (!oriented)
		graph[end][start] = weight;
}

void Graph::delete_edge(size_t start, size_t end)
{
	graph[start][end] = 0;
	if (!oriented)
		graph[end][start] = 0;
}

void Graph::dijkstra()
{
	size_t vertex = start;
	int min_distance = 0;
	distances[start] = 0;
	while (min_distance < INT32_MAX)
	{
		size_t i = vertex;
		min_distance = INT32_MAX;
		visited[i] = true;
		for (size_t j = 0; j < size; j++)
		{
			if (graph[i][j] && !visited[j] && distances[j] > distances[i] +
graph[i][j])
				distances[j] = distances[i] + graph[i][j];
			if (!visited[j] && distances[j] < min_distance)
			{
				min_distance = distances[j];
				vertex = j;
			}
		}
	}
}

void Graph::prepare_dijkstra(size_t start)
{
	visited = std::vector<bool>(size, false);
	distances = std::vector<int>(size, INT32_MAX);
	this->start = start;
}

int Graph::get_min_distance(size_t start, size_t end)
{
	prepare_dijkstra(start);
	dijkstra();
	return distances[end];
}

std::vector<int> split(const std::string& line)
{
	std::vector<int> res;
	size_t index = 0;
	while (line.find(" ", index) != std::string::npos)
	{
		size_t new_index = line.find(" ", index);
		std::string word = line.substr(index, new_index - index);
		if (word.length())
			res.push_back(std::stoi(word));
		index = new_index + 1;
	}
	if (index < line.length())
		res.push_back(std::stoi(line.substr(index)));
```

```
	return res;
}

void read_data(Graph& obj, std::string file_name)
{
	std::ifstream fin(file_name);
	if (!fin)
		throw std::runtime_error("wrong file");
	std::string line;
	std::getline(fin, line);
	size_t size = std::stoi(line);
	std::getline(fin, line);
	bool oriented = std::stoi(line);
	std::vector<std::vector<int>> graph;
	while (!fin.eof())
	{
		std::getline(fin, line);
		std::vector<int> arr = split(line);
		graph.push_back(arr);
	}
	fin.close();
	obj.set(size, oriented, graph);
	obj.is_valid();
}

std::ostream& operator<<(std::ostream& out, const Graph& graph)
{
	for (size_t i = 0; i < graph.size; i++)
	{
		for (size_t j = 0; j < graph.size; j++)
			out << graph.graph[i][j] << " ";
		out << std::endl;
	}
	return out;
}

std::istream& operator>>(std::istream& in, Graph& graph)
{
	size_t n = graph.size;
	graph.graph = std::vector<std::vector<int>>(n, std::vector<int>(n));
	for (size_t i = 0; i < n; i++)
		for (size_t j = 0; j < n; j++)
			in >> graph.graph[i][j];
	return in;
}

template<typename T>
std::ostream& operator<<(std::ostream& out, const std::vector<T> arr)
{
	for (T item : arr)
		out << item << " ";
	return out;
}
```

Исполняемый файл main.cpp:

```
#include <iostream>
#include "graph.hpp"


int main() {
	std::string file_name = "test3.txt";
	Graph graph;
```

```
	try {
		read_data(graph, file_name);
		std::cout << "Minimum distance: " << graph.get_min_distance(0, 7) <<
std::endl;
	}
	catch (const std::runtime_error& ex) {
		std::cout << ex.what() << std::endl;
		return 1;
	}
	std::cout << "Graph:\n" << graph << std::endl;


	return 0;
}
```

Снимки выполнения работы программы

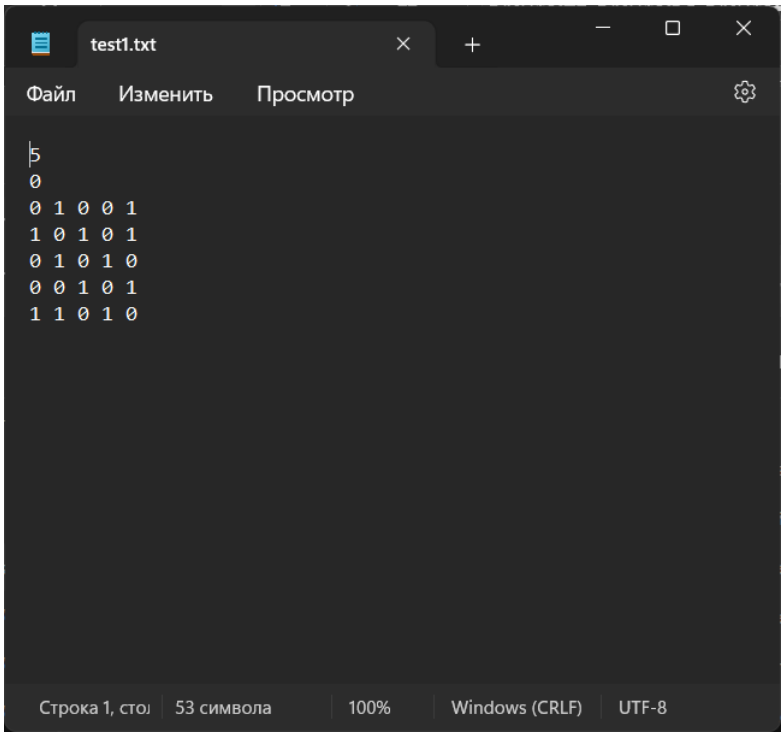

Рисунок 1 – тестовые данные test1

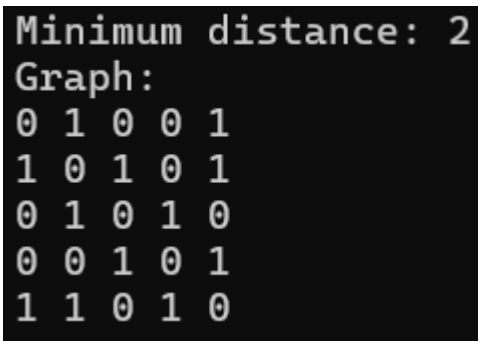

Рисунок 2 – Результат работы программы для тестовых данных test1 (поиск кратчайшего пути из вершины 1 до вершины 3)

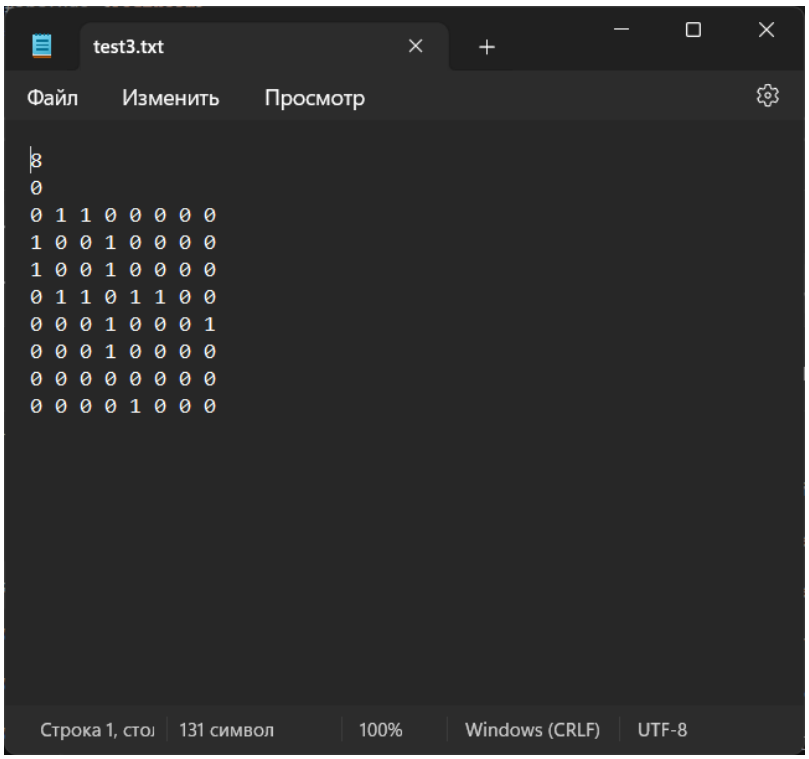

Рисунок 3 – тестовые данные test2

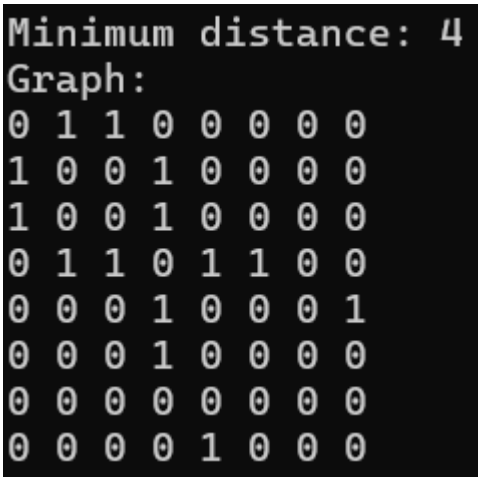

Рисунок 4 – Результат работы программы для тестовых данных test3 (поиск кратчайшего пути из вершины 1 до вершины 8)

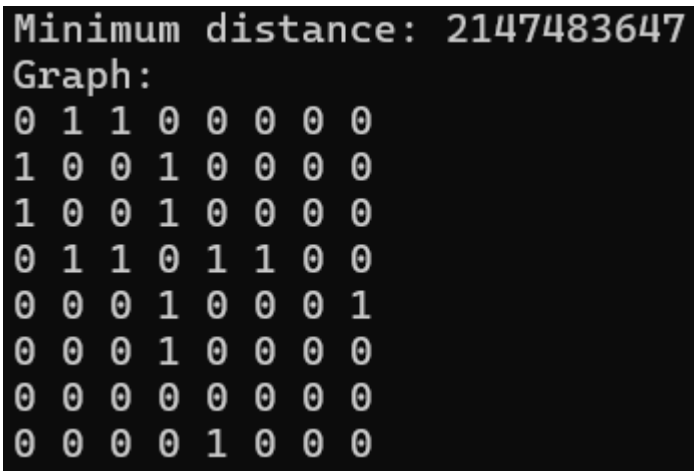

Рисунок 5 – Результат работы программы для тестовых данных test3 (поиск кратчайшего пути из вершины 1 до вершины 7) – бесконечное расстояние, т.е. отсутствие связи между вершинами

Заключение

Задачи лабораторной работы были решены, результаты проверены. Изучены на практике алгоритмы на графах и принципы объектно ориентированного программирования в языке C/C++.