ГУАП

КАФЕДРА № 14

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

ассистент		П. В. Шпигун
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 5

НАХОЖДЕНИЕ КРАТЧАЙШЕГО ПУТИ В ОРИЕНТИРОВАННОМ ВЗВЕШЕННОМ ГРАФЕ

по курсу: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №	1446	22.12.2025	А. С. Пырву
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

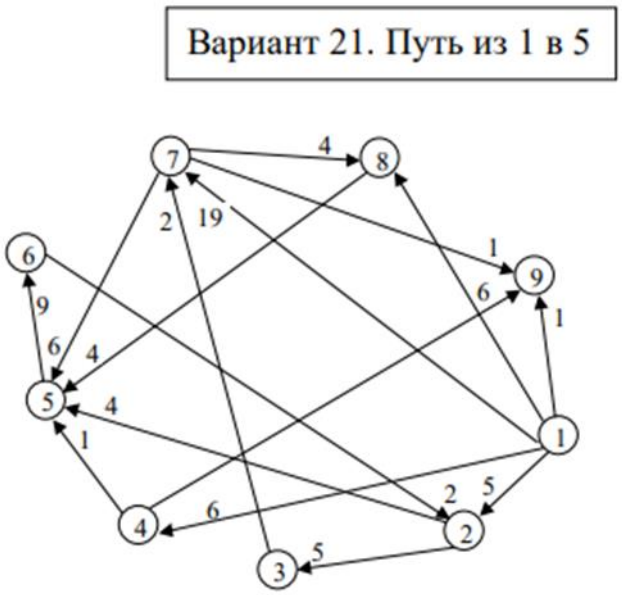
Цель работы: Изучение и реализация алгоритмов поиска кратчайших путей в ориентированных взвешенных графах: алгоритма Флойда–Уоршелла и алгоритма Беллмана–Форда.

Постановка задачи: В данной лабораторной работе необходимо найти кратчайший путь в ориентированном взвешенном графе:

1. Вручную, так как размер графа небольшой и решить поставленную задачу можно без программирования. В отчете привести сам граф с отмеченными другим цветом или толщиной линий найденное решение.
2. Решить ту же задачу программным образом, используя любые два из трех известных алгоритмов:

a) Алгоритм Флойда (алгоритм Флойда–Уоршелла)
b) Алгоритм Форда-Беллмана
c) Алгоритм Дейкстры

Вариант:



Формализация:

Константы:

MAX = INT_MAX — обозначение бесконечности (отсутствия пути).
NO_EDGE = -1 — специальное значение для отсутствия ребра в матрице смежности.

Структуры данных:

graph_create — создание и инициализация графа.
graph_free — освобождение памяти, занятой графом.
create_matrix, free_matrix — работа с вспомогательными матрицами.
create_flojd_distances, create_flojd_paths — подготовка матриц для алгоритма Флойда.
flojd — реализация алгоритма Флойда–Уоршелла.
create_bellman_distances, create_bellman_paths — инициализация массивов для алгоритма Беллмана.
bellman — реализация алгоритма Беллмана–Форда.

Использованные библиотеки:
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

Листинг программы:

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX INT_MAX
#define NO_EDGE -1  // специальное значение для отсутствия ребра

typedef struct
{
    int** matrix;
    int vertices;
} graph_t;

void graph_create(graph_t* graph, int vertices)
{
    graph->vertices = vertices;
    graph->matrix = malloc(vertices * sizeof(int*));
    for (int i = 0; i < vertices; i++)
    {
        graph->matrix[i] = malloc(vertices * sizeof(int));
        for (int j = 0; j < vertices; j++)
        {
            graph->matrix[i][j] = NO_EDGE;
        }
    }
}

void graph_free(graph_t* graph)
{
    for (int i = 0; i < graph->vertices; i++)
        free(graph->matrix[i]);
    free(graph->matrix);
}

int** create_matrix(int n)
{
    int** matrix = malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++)
        matrix[i] = malloc(n * sizeof(int));
    return matrix;
}

void free_matrix(int** matrix, int n)
{
    for (int i = 0; i < n; i++)
        free(matrix[i]);
    free(matrix);
}

void create_flojd_distances(int** d, graph_t* graph)
{
    int n = graph->vertices;
    for (int i = 0; i < n; i++)
```

```
    {
        for (int j = 0; j < n; j++)
        {
            if (i == j)
                d[i][j] = 0;
            else if (graph->matrix[i][j] != NO_EDGE)
                d[i][j] = graph->matrix[i][j];
            else
                d[i][j] = MAX;
        }
    }
}

void create_flojd_paths(int** p, graph_t* graph)
{
    int n = graph->vertices;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (graph->matrix[i][j] != NO_EDGE && i != j)
                p[i][j] = j;
            else
                p[i][j] = -1;
        }
    }
}

void flojd(graph_t* graph, int start, int end)
{
    int n = graph->vertices;
    int** d = create_matrix(n);
    int** p = create_matrix(n);
    create_flojd_distances(d, graph);
    create_flojd_paths(p, graph);

    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (d[i][k] != MAX && d[k][j] != MAX)
                {
                    if (d[i][k] + d[k][j] < d[i][j])
                    {
                        d[i][j] = d[i][k] + d[k][j];
                        p[i][j] = p[i][k];
                    }
                }
            }
        }
    }

    printf("\nFlojd algo:\n");
    if (d[start][end] == MAX)
        printf("no path\n");
    else
    {
        printf("len: %d\n", d[start][end]);
        printf("path: ");
        int v = start;
        printf("%d", v + 1);
        while (v != end)
        {
```

```
            v = p[v][end];
            if (v == -1)
                break;
            printf(" -> %d", v + 1);
        }
        printf("\n");
    }

    free_matrix(d, n);
    free_matrix(p, n);
}

void create_bellman_distances(int* d, int n, int start)
{
    for (int i = 0; i < n; i++)
        d[i] = MAX;
    d[start] = 0;
}

void create_bellman_paths(int* p, int n)
{
    for (int i = 0; i < n; i++)
        p[i] = -1;
}

void bellman(graph_t* graph, int start, int end)
{
    int n = graph->vertices;
    int* d = malloc(n * sizeof(int));
    int* p = malloc(n * sizeof(int));
    create_bellman_distances(d, n, start);
    create_bellman_paths(p, n);

    for (int i = 0; i < n - 1; i++)
    {
        for (int u = 0; u < n; u++)
        {
            if (d[u] != MAX)
            {
                for (int v = 0; v < n; v++)
                {
                    if (graph->matrix[u][v] != NO_EDGE && d[u] + graph->matrix[u][v] <
d[v])
                    {
                        d[v] = d[u] + graph->matrix[u][v];
                        p[v] = u;
                    }
                }
            }
        }
    }

    printf("\nBellman algo:\n");
    if (d[end] == MAX)
        printf("no path\n");
    else
    {
        printf("len: %d\n", d[end]);
        int* path = malloc(n * sizeof(int));
        int len = 0;
        int v = end;
        while (v != -1)
        {
            path[len++] = v;
            v = p[v];
```

```
        }
        printf("path: ");
        for (int i = len - 1; i >= 0; i--)
        {
            printf("%d", path[i] + 1);
            if (i > 0)
                printf(" -> ");
        }
        printf("\n");
        free(path);
    }
    free(d);
    free(p);
}

int main()
{
    FILE* f = fopen("graph.txt", "r");
    if (!f)
    {
        printf("error\n");
        return 1;
    }

    int n, m;
    fscanf(f, "%d", &n);
    fscanf(f, "%d", &m);

    graph_t g;
    graph_create(&g, n);

    for (int i = 0; i < m; i++)
    {
        int from, to, length;
        fscanf(f, "%d %d %d", &from, &to, &length);
        g.matrix[from - 1][to - 1] = length;
    }
    fclose(f);

    int s, e;
    printf("start: ");
    scanf("%d", &s);
    printf("end: ");
    scanf("%d", &e);
    s--;
    e--;

    flojd(&g, s, e);
    bellman(&g, s, e);
    graph_free(&g);
    return 0;
}
```

Листинг 1. Код программы

Тестирование:

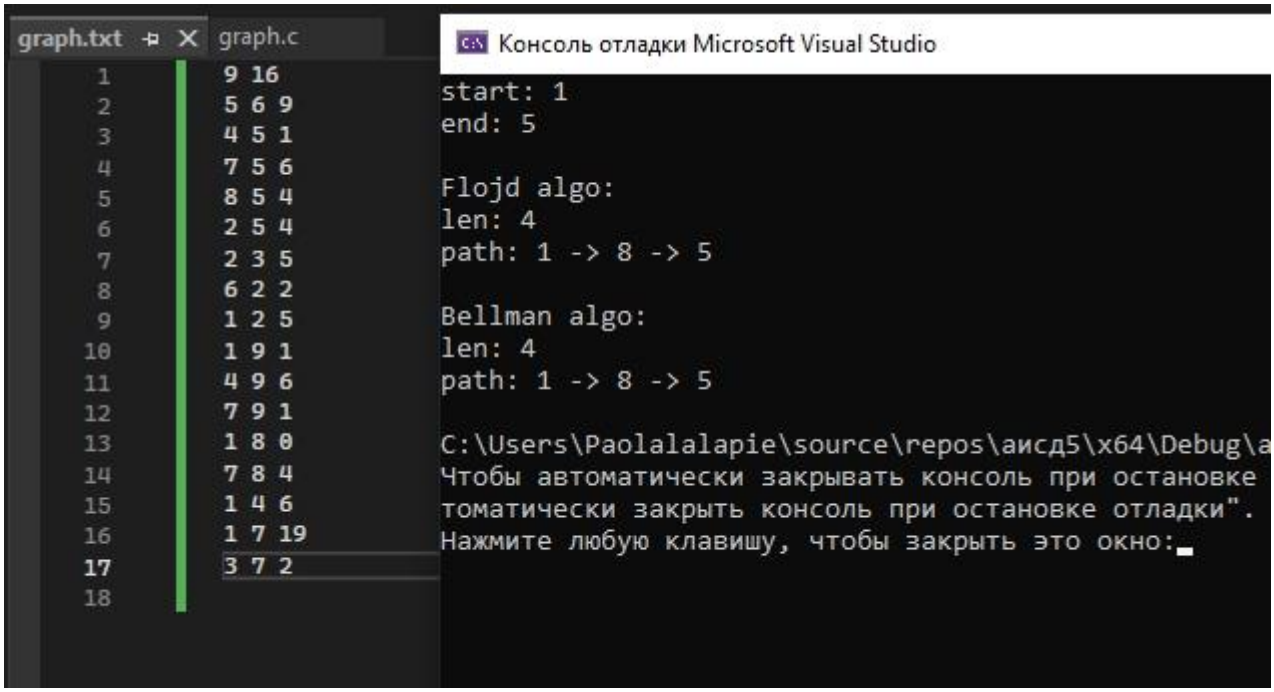


Рисунок 1. Перенесённый в файл граф и высчитанный кратчайший путь из 1 в 5 по двум из предложенных алгоритмов.