

Содержание

[**Реферат** 5](#_Toc59020615)

[**1.** **Введение** 6](#_Toc59020616)

[**2.** **Постановка задачи** 7](#_Toc59020617)

[**3.** **Теоретическая часть задания** 8](#_Toc59020618)

[**4.** **Описание алгоритма программы** 9](#_Toc59020619)

[**5.** **Описание программы** 11](#_Toc59020620)

[**6.** **Тестирование** 15](#_Toc59020621)

[**7.** **Ручной расчёт задачи** 21](#_Toc59020622)

[**Заключение** 24](#_Toc59020623)

[**Список литературы** 25](#_Toc59020624)

[**Приложение А. Листинг программы.** 26](#_Toc59020625)

**Алгоритм Флойда**.

Цель исследования – разработка программы, способная находить кратчайшие пути в каждую из всех вершин в графе. Получить навыки в разработке консольного приложения с пользовательским интерфейсом. Сохранить результаты работы программы в файл.

# **Введение**

Алгоритм Флойда предназначен для нахождения кратчайшихпутеймежду всемивершинами в графе. По сути он представляет собой простой перебор всех путей и выбор из них наименьшего. Перебор осуществляется по так называемой **«матрице смежности»** размера NxN, где N — количество вершин графа.

В качестве среды разработки мною была выбрана среда MicrosoftVisualStudio2019, язык программирования – Си/C++.

Целью данной курсовой работы является разработка программы на языке Си/C++, который является широко используемым. Именно с его помощью в данном курсовом проекте реализуется алгоритм Флойда-Уоршелла.

# **Постановка задачи**

Требуется разработать программу, которая выводила исходную матрицу смежности и обработанную с кратчайшими путями. Сначала необходимо, чтобы пользователь задал ввод матрицы: автоматический (случайная генерация) или ручной(в самой программе или из файла). Исходный граф в программе должен задаваться матрицей смежности, причём при генерации данных должны быть предусмотрены граничные условия. Программа должна работать так, чтобы пользователь вводил количество вершин для генерации матрицы смежности. После обработки этих данных на экран должна выводиться матрица смежности графа с кратчайшими путями и сохранение результатов в файл. Устройство ввода – клавиатура и мышь.

Таблица 1 – Описание поведения программы при тестировании

|  |  |  |  |
| --- | --- | --- | --- |
| Описание теста | Как достичь | Ожидаемый результат | Полученный результат |
| Запуск программы | Открыть exe - файл | Вывод сообщения о выборе:  сгенерировать матрицу или  вывести матрицу из памяти или ввести её самому. | Верно |
| Выбор генерации матрицы | Ввести в программе номер нужной команды | Вывод сообщения о  количестве вершин в графе  и выбора его вида. | Верно |
| Вывод матрицы | Открыть созданный программой файл в папке Debug | Сохранение результатов работы программы в файл | Верно |
| Ввод неверной команды или элемента | Ввести в программе или файле неверные значения | Вывод сообщения с просьбой изменить введённое значение. | Верно |

# **Теоретическая часть задания**

Граф G (рисунок 1) задается множеством вершин X1, X2, ..., Xn. и множеством ребер , соединяющих между собой определенные вершины. Ребра из множества А ориентированы, что показывается стрелкой, которая указывает достижимость данной вершины, граф с такими ребрами называется ориентированным графом.

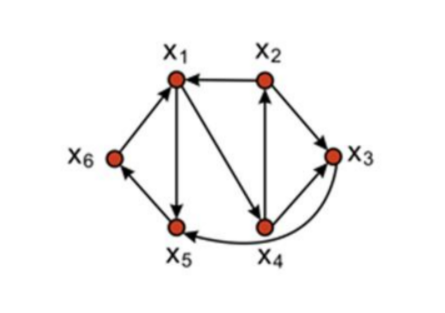


Рисунок 1 - Пример графа

При представлении графа матрицей смежности информация о ребрах графа хранится в квадратной матрице, где присутствие пути из одной вершины в другую обозначается единицей, иначе нулем. Существует много алгоритмов на графах, для нахождения кратчайших путей. Остановимся на алгоритме Флойда. Сформируем матрицу смежности **D[][]** графа **G=(V, E)**, в котором каждая вершина пронумерована от **1** до **|V|**. Эта матрица имеет размер**|V|´|V|**, и каждому ее элементу **D[i][j]** присвоен вес ребра, идущего из вершины **i** в вершину **j**. По мере выполнения алгоритма, данная матрица будет перезаписываться: в каждую из ее ячеек внесется значение, определяющее оптимальную длину пути из вершины **i** в вершину **j**.

# **Описание алгоритма программы**

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на **фазы**.

Перед k-ой фазой (k = 1 \ldots n) считается, что в матрице расстояний d[][] сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества \{ 1, 2, \ldots, k-1 \} (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой фазой величина d[i][j] равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний d[][] записать матрицу смежности графа: d[i][j] = g[i][j] — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" \infty. Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим **пересчитать** матрицу d[][] таким образом, чтобы она соответствовала требованиям уже для k+1-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

* Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины \{ 1, 2, \ldots, k \}, **совпадает** с кратчайшим путём, которому разрешено проходить через вершины множества \{ 1, 2, \ldots, k-1 \}.

В этом случае величина d[i][j] не изменится при переходе с k-ой на k+1-ую фазу.

* "Новый" кратчайший путь стал **лучше** "старого" пути.

Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая i \Rightarrow k, а другая — k \Rightarrow j), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на k-1-ой фазе или ещё раньше, и нам достаточно взять просто сумму d[i][k] + d[k][j], она и даст длину "нового" кратчайшего пути.

**Объединяя** эти два случая, получаем, что на k-ой фазе требуется пересчитать длины кратчайших путей между всеми парами вершин i и j следующим образом:

new\_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний d[i][j] будет записана длина кратчайшего пути между i и j, либо \infty, если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно **не создавать отдельную матрицу** \rm new\_d[][] для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице d[][]. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

**Асимптотика** алгоритма, очевидно, составляет O (n^3).

Если ребра между вершинами не будет или он окажется слишком большим, то мы в матрице выводим INF. В программе по умолчанию за INF мы принимаем значение 50.

Псевдокод алгоритма Флойда:

G – матрица смежности графа

1. Для k = 1 до n выполнять

2. Для i = 1 до n выполнять

3. Для j = 1 до n выполнять

4. Если G(i, k) + G(k, j) < G(I, j)

5. ТО

6. {

7. G(i, j) = G(i, k) + G(k, j)

8. }

# **Описание программы**

Данная программа является многомодульной, поскольку состоит из

нескольких функций: main, printMatrix для вывода матрицы, originalFloyd для рассчёта кратчайших путей.

Работа программы начинается с запроса генерации матрицы. Если

пользователь выбрал сгенерировать матрицу смежности, то на экран выводится запрос на количество вершин в графе, а затем выбор ориентированного графа или нет. Выбор осуществляется с помощью переменной numberEnter.

Рассмотрим часть программы с выбором случайной генерации матрицы:

if (numberEnter == 1) {

cout << "Введите количество вершин в матрице: ";

scanf\_s("%d", &n);

cout << endl;

cout << "Выберите вид графа: " << endl << "1. Ориентированный" << endl << "2. Неориентированный" << endl;

Добавим srand(time(NULL)), чтобы при каждом запуске генерировалась новая случайная матрица.

srand(time(NULL));

int\*\* matrix;

matrix = (int\*\*)malloc(sizeof(int) \* n); //выделяем память для

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

matrix[i] = (int\*)malloc(sizeof(int) \* n); // заполнения матрицы

}

С помощью переменной proverka и цикла while проверяем правильность введённого значения.

proverka = false;

while (!proverka)

{

cin >> numberVec;

if (numberVec == 1 || numberVec == 2) {

proverka = true;

}

else

{

printf("Команда введена неверно или её не существует. Попробуйте ещё раз.\n");

}

}

При вводе верной команды будет выполнена генерация матрицы

// Случайная генерация матрицы

if (numberEnter == 1 || numberEnter == 2) {

for (int i = 0; i < n; i++) //заполнение массива

{

for (int j = 0; j < n; j++)

{

Генерируем в матрице главную диагональ состоящую из нулей

if (i == j)

{

matrix[i][j] = 0;

}

Далее генерируем элементы в матрице. С помощью переменной ver мы будем создавать связь между вершинами с вероятностью 50%.

else

{

ver = rand() % 2;

// вершины связаны

if (ver == 1) {

matrix[i][j] = rand() % 100;

}

// вершины не связаны

else {

matrix[i][j] = INF;

}

}

}

}

Если был выбран неориентированный граф выполняется следующее условие:

if (numberVec == 2) {

for (int i = 0; i < n; i++) //отзеркаливание матрицы

{

for (int j = 0; j < n; j++)

{

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

}

}

}

}

Далее мы выводим исходную матрицу на экран с помощью функции printMatrix.

printMatrix(matrix, n, fout);

Функция с алгоритмом Флойда

originalFloyd(matrix, n);

Далее снова вызываем функцию для вывода матрицы на экран, но в этот раз обработанную.

printMatrix(matrix, n, fout, a);

cout << endl << "Результат сохранён в файл newMatrix.txt" << endl;

}

Функция originalFloyd

Функция принимает матрицу matrix и количество вершин n

void originalFloyd(int\*\* matrix, int n) {

//Пробегаемся по всем вершинам и ищем более короткий путь

//через вершину k

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

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

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

//Новое значение ребра равно минимальному между старым

//и суммой ребер i <-> k + k <-> j (если через k пройти быстрее)

matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);

}

}

}

return;

}

Функция printMatrix

Функция принимает матрицу matrix, количество вершин n и файл fout для сохранения результатов в файл.

void printMatrix(int\*\* matrix, int n, ofstream &fout) {

cout << "Исходная матрица: " << endl;

fout << "Исходная матрица: " << endl;

fout << "Количество вершин: " << n << endl;

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

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

if (matrix[i][j] >= INF) {

cout << "INF" << " ";

fout << "INF" << " ";

}

else {

cout << matrix[i][j] << " ";

fout << matrix[i][j] << " ";

}

}

cout << endl;

fout << endl;

}

cout << endl;

fout << endl;

}

Пользователь может также может ввести матрицу в самой программе или из файла. Полный код см. в приложении А.

Ниже можно увидеть оформление начального запроса и дальнейшие

действия с ним.

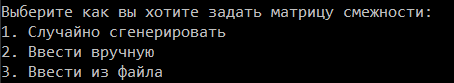


Рисунок 2- Выбор ввода

Ввод количества вершин.



Рисунок 3 - Ввод количества вершин

Ввод вида графа.

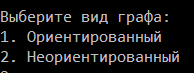


Рисунок 4 - Ввод вида графа

Имеем:



Рисунок 5 - конечный результат

# **Тестирование**

Среда разработки MicrosoftVisualStudio 2010 предоставляет все

средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось в рабочем порядке, в процессе разработки,

после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций.

Ниже продемонстрированы результаты тестирования программы при

всех возможных вариантах ввода.

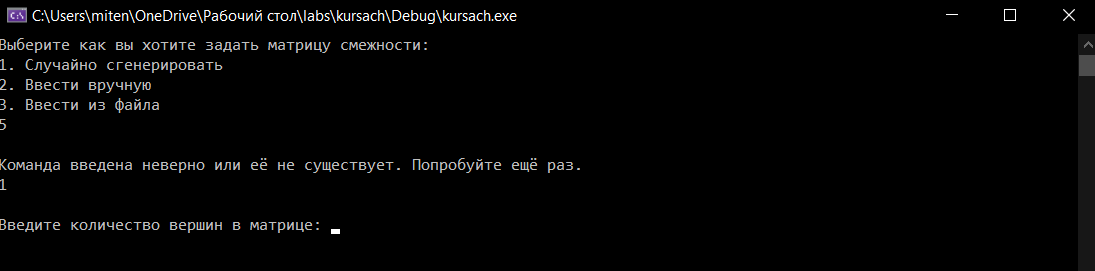


Рисунок 6 - неверный ввод



Рисунок 7 - автоматический ввод неориентированного графа

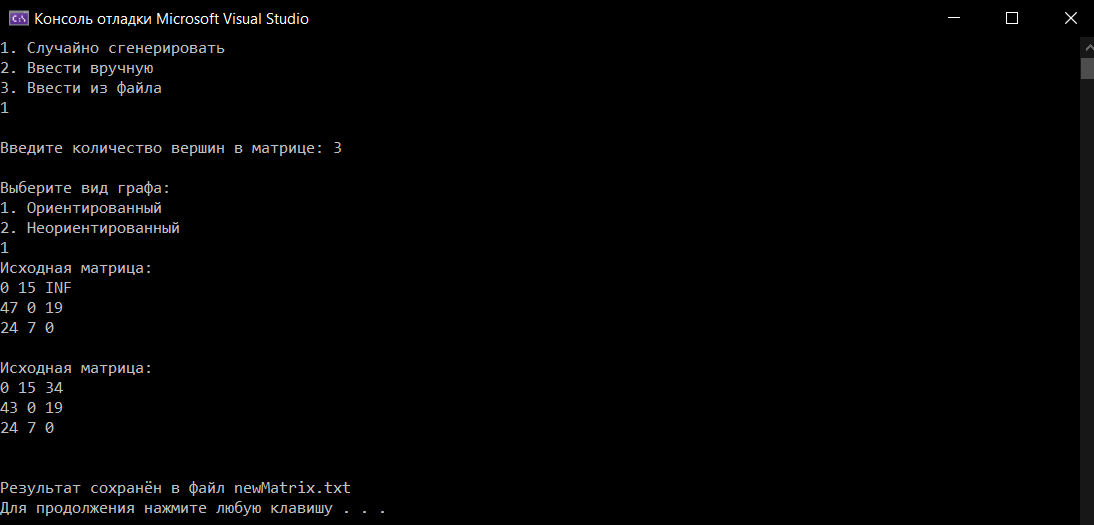


Рисунок 8 - автоматический ввод ориентированного графа

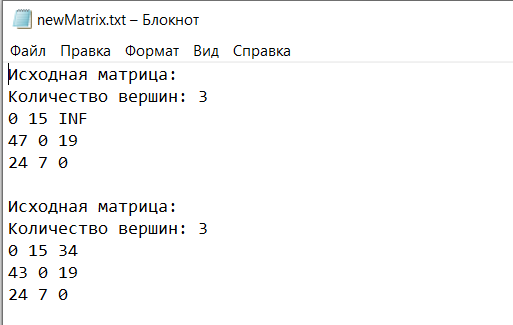


Рисунок 9 - сохранение результата в файл

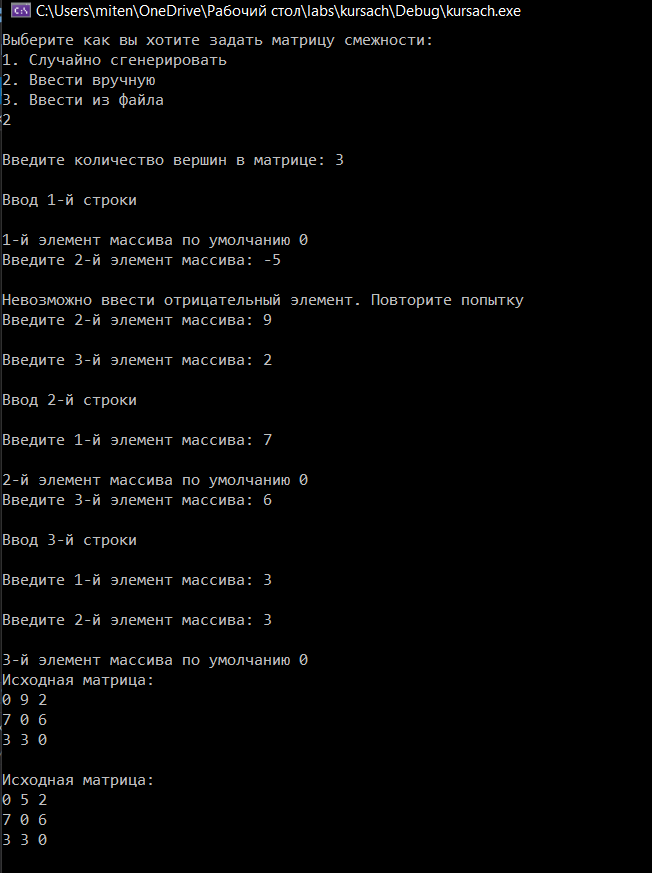


Рисунок 10 - ручной ввод с ошибкой и без

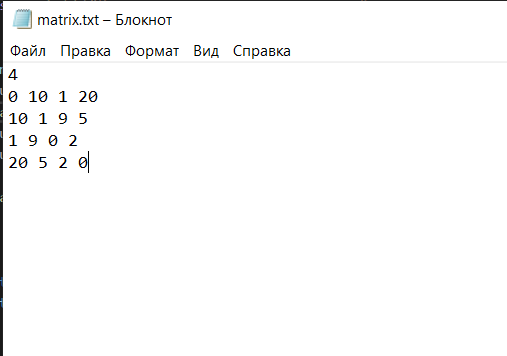


Рисунок 11 - файл с исходной матрицей с ошибкой

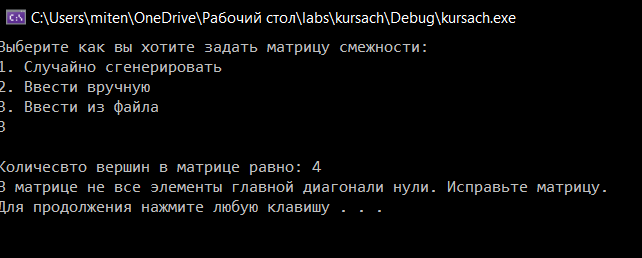


Рисунок 12 - вывод ошибки

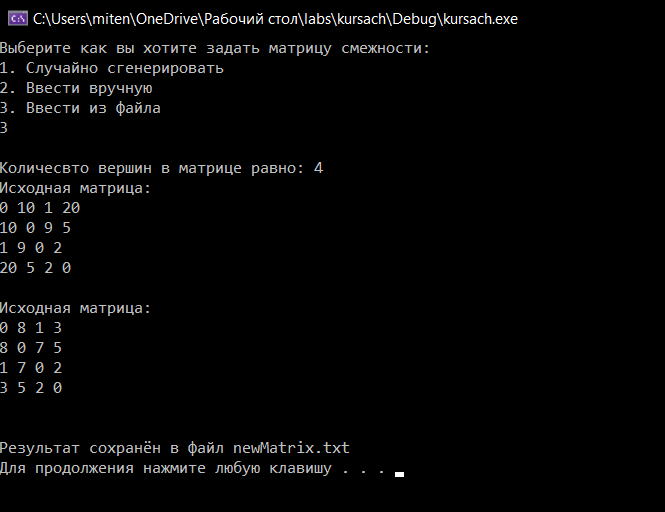


Рисунок 13 - вывод нормальной работы программы

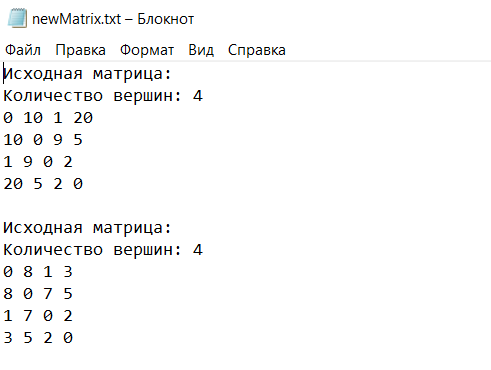


Рисунок 14 - то же сохранение результатов

В результате тестирования было выявлено, что программа успешно

проверяет данные на соответствие необходимым требованиям.

# **Ручной расчёт задачи**

Например, дан граф:

2

5

3

1

7

5

5

88

2

4

12

Тогда начальная весовая матрица будет:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | INF |
| 2 | 7 | 0 | INF | INF | 12 |
| 3 | 5 | INF | 0 | 5 | 2 |
| 4 | 8 | INF | 5 | 0 | INF |
| 5 | INF | 12 | 2 | INF | 0 |

после 1-ой итерации:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | INF |
| 2 | 7 | 0 | 12 | 15 | 12 |
| 3 | 5 | 12 | 0 | 5 | 2 |
| 4 | 8 | 15 | 5 | 0 | INF |
| 5 | INF | 12 | 2 | INF | 0 |

после 2-ой итерации:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | 19 |
| 2 | 7 | 0 | 12 | 15 | 12 |
| 3 | 5 | 12 | 0 | 5 | 2 |
| 4 | 8 | 15 | 5 | 0 | 27 |
| 5 | 19 | 12 | 2 | 27 | 0 |

после 3-ей итерации:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | 7 |
| 2 | 7 | 0 | 12 | 15 | 12 |
| 3 | 5 | 12 | 0 | 5 | 2 |
| 4 | 8 | 15 | 5 | 0 | 7 |
| 5 | 7 | 12 | 2 | 7 | 0 |

после 4-ой итерации:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | 7 |
| 2 | 7 | 0 | 12 | 15 | 12 |
| 3 | 5 | 12 | 0 | 5 | 2 |
| 4 | 8 | 15 | 5 | 0 | 7 |
| 5 | 7 | 12 | 2 | 7 | 0 |

после 5-ой итерации:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номера вершин | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 7 | 5 | 8 | 7 |
| 2 | 7 | 0 | 12 | 15 | 12 |
| 3 | 5 | 12 | 0 | 5 | 2 |
| 4 | 8 | 15 | 5 | 0 | 7 |
| 5 | 7 | 12 | 2 | 7 | 0 |

Результат работы программы:

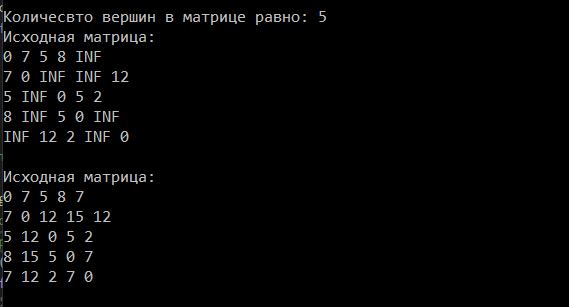


Рисунок 15 - проверка

# **Заключение**

Таким образом, в процессе создания данного проекта разработана

программа, реализующая алгоритм Флойда для нахождения кратчайших путей в графе в MicrosoftVisualStudio 2019.

При выполнении данной курсовой работы были получены навыки

разработки программ и освоены приемы создания матриц смежностей,

основанных на теории графов. Углублены знания языка программирования Cи/C++.

Недостатком разработанной программы является примитивный

пользовательский интерфейс. Потому что программа работает в консольном

режиме, не добавляющем к сложности языка сложность программного

оконного интерфейса.

Программа имеет небольшой, но достаточный для использования

функционал возможностей.

# **Список литературы**

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и

анализ - М.: МЦНМО, 2001. - 960 с.

2. Кристофидес Н. «Теория графов. Алгоритмический подход» - Мир,

1978

3. Герберт Шилдт «Полный справочник по C++» - Вильямс, 2006

4. Уилсон Р. Введение в теорию графов. Пер. с анг. 1977. 208 с.

5. Харви Дейтел, Пол Дейтел. Как программировать на C/C++. 2009 г.

# **Приложение А. Листинг программы.**

#include <iostream>

#include <algorithm>

#include <fstream>

#include <time.h>

#include <locale.h>

#define INF 999999999

using namespace std;

// Вывод матрицы

void printMatrix(int\*\* matrix, int n, ofstream &fout) {

cout << "Исходная матрица: " << endl;

fout << "Исходная матрица: " << endl;

fout << "Количество вершин: " << n << endl;

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

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

if (matrix[i][j] >= INF) {

cout << "INF" << " ";

fout << "INF" << " ";

}

else {

cout << matrix[i][j] << " ";

fout << matrix[i][j] << " ";

}

}

cout << endl;

fout << endl;

}

cout << endl;

fout << endl;

}

//Алгоритм Флойда

void originalFloyd(int\*\* matrix, int n) {

//Пробегаемся по всем вершинам и ищем более короткий путь

//через вершину k

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

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

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

//Новое значение ребра равно минимальному между старым

//и суммой ребер i <-> k + k <-> j (если через k пройти быстрее)

matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);

}

}

}

return;

}

int main(int argc, char\*\* argv) {

//matrix - матрица смежности

ofstream fout("newMatrix.txt");

setlocale(LC\_ALL, "Rus");

// n - количество вершин

// numberEnter - номер выбранного варианта ввода матрицы

// numberVec - номер ввода для вида направленности графа

int n, numberEnter, numberVec;

// proverka - переменная для проверки правильности варианта ввода

bool proverka = false;

//ver - переменная, отвечающая за вероятность генерации связанной вершины

int ver;

cout << "Выберите как вы хотите задать матрицу смежности: " << endl << "1. Случайно сгенерировать" << endl << "2. Ввести вручную" << endl << "3. Ввести из файла" << endl;

while (!proverka) {

scanf\_s("%d", &numberEnter);

cout << endl;

if (numberEnter == 1 || numberEnter == 2 || numberEnter == 3) proverka = true;

else printf("Команда введена неверно или её не существует. Попробуйте ещё раз.\n");

}

if (numberEnter == 1) {

cout << "Введите количество вершин в матрице: ";

scanf\_s("%d", &n);

cout << endl;

cout << "Выберите вид графа: " << endl << "1. Ориентированный" << endl << "2. Неориентированный" << endl;

srand(time(NULL));

int\*\* matrix;

matrix = (int\*\*)malloc(sizeof(int) \* n); //выделяем память для

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

matrix[i] = (int\*)malloc(sizeof(int) \* n); // заполнения матрицы

}

proverka = false;

while (!proverka)

{

cin >> numberVec;

if (numberVec == 1 || numberVec == 2) {

proverka = true;

}

else

{

printf("Команда введена неверно или её не существует. Попробуйте ещё раз.\n");

}

}

// Случайная генерация матрицы

if (numberEnter == 1 || numberEnter == 2) {

for (int i = 0; i < n; i++) //заполнение массива

{

for (int j = 0; j < n; j++)

{

if (i == j)

{

matrix[i][j] = 0;

}

else

{

ver = rand() % 2;

// вершины связаны

if (ver == 1) {

matrix[i][j] = rand() % 100;

}

// вершины не связаны

else {

matrix[i][j] = INF;

}

}

}

}

if (numberVec == 2) {

for (int i = 0; i < n; i++) //отзеркаливание матрицы

{

for (int j = 0; j < n; j++)

{

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

}

}

}

}

printMatrix(matrix, n, fout);

originalFloyd(matrix, n);

printMatrix(matrix, n, fout);

cout << endl << "Результат сохранён в файл newMatrix.txt" << endl;

}

// Ручной ввод

if (numberEnter == 2) {

cout << "Введите количество вершин в матрице: ";

scanf\_s("%d", &n);

cout << endl;

int\*\* matrix;

int numb;

matrix = (int\*\*)malloc(sizeof(int) \* n); //выделяем память для

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

matrix[i] = (int\*)malloc(sizeof(int) \* n); //заполнения матрицы

}

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

cout << "Ввод " << i + 1 << "-й строки" << endl <<endl;

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

if (i == j)

{

cout << j + 1 << "-й элемент массива по умолчанию 0" << endl;

matrix[i][j] = 0;

}

else

{

proverka = false;

while (!proverka) {

cout << "Введите " << j + 1 << "-й элемент массива: ";

cin >> numb;

cout << endl;

if (numb >= 0) {

matrix[i][j] = numb;

proverka = true;

}

else

{

cout << "Невозможно ввести отрицательный элемент. Повторите попытку" << endl;

proverka = false;

}

}

}

}

}

printMatrix(matrix, n, fout);

originalFloyd(matrix, n);

printMatrix(matrix, n, fout);

cout << endl

<< "Результат сохранён в файл newMatrix.txt" << endl;

}

// с Ввод с файла

if (numberEnter == 3) {

ifstream file("matrix.txt");

if (!file.is\_open()) { // если файл не открыт

cout << "Файл не может быть открыт!\n";

system("pause");

return 1;

}

file >> n;

cout << "Количесвто вершин в матрице равно: " << n << endl;

int\*\* matrix = (int\*\*)malloc(sizeof(int) \* n); //выделяем память для

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

matrix[i] = (int\*)malloc(sizeof(int) \* n); //заполнения матрицы

}

//Считываем матрицу

//return 1 - аварийное завершение

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

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

file >> matrix[i][j];

if (matrix[i][j] < 0) {

cout << "В матрице обнаружет отрицательный элемент. Исправьте матрицу." << endl;

system("pause");

return 1;

}

if (i == j) {

if (matrix[i][j] != 0) {

cout << "В матрице не все элементы главной диагонали нули. Исправьте матрицу." << endl;

system("pause");

return 1;

}

}

}

}

file.close(); // закрываем файл

printMatrix(matrix, n, fout);

originalFloyd(matrix, n);

printMatrix(matrix, n, fout);

cout << endl << "Результат сохранён в файл newMatrix.txt" << endl;

}

system("pause");

return 0;

}