

**Содержание**

[Реферат 5](#_Toc91099467)

[Введение 6](#_Toc91099468)

[1. Постановка задачи 7](#_Toc91099469)

[2. Теоретическая часть 8](#_Toc91099470)

[3. Описание алгоритма 9](#_Toc91099471)

[4. Описание программы 12](#_Toc91099472)

[5. Тестирование 16](#_Toc91099473)

[6. Ручной расчет задачи 18](#_Toc91099474)

[Заключение 20](#_Toc91099475)

[Список используемых источников 21](#_Toc91099476)

[Листинги программы 22](#_Toc91099477)

# **Реферат**

Отчет 27страницы, 15 рисунков.

ГРАФ, ПОИСК В ГЛУБИНУ, ТЕОРИЯ ГРАФОВ, ОБХОД ГРАФА, МАТРИЦА СМЕЖНОСТИ.

Цель исследования – разработка программы, способной находить выход из лабиринта, представленном в виде графа, используя алгоритм поиска в глубину.

В работе рассмотрены правила поиска в глубину, на основе которых находится путь, ведущий к выходу лабиринта.

# **Введение**

Алгоритм поиска в глубину (англ. Depth-FirstSearch, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Поиск в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Если алгоритм попадает в лист, то нужно возвращаться назад вдоль пройденного пути, пока не будет обнаружена вершина, у которой есть еще не посещенная смежная вершина, и затем двигаться в направлении не посещённой вершины. Эти действия повторяются до возврата в начальную вершину после посещения всех остальных.

Таким образом основная идея поиска в глубину – сначала полностью исследовать одну ветку вглубь и только потом переходить к другим веткам. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатываются несколько вершин.

При построении лабиринта придерживалась следующего алгоритма: сначала выбираем начальную клетку, затем задаем случайное направление для прокладывания пути. Если соседняя клетка в этом направлении выходит за границы поля, прокопать клетку в единственно возможном направлении и переходим к следующей клетке. Повторяем эти действия до тех пор, пока не будут обработаны все клетки;

В качестве среды разработки была выбрана среда MicrosoftVisualStudio 2019, язык программирования – C.

Целью данной курсовой работы является разработка на языке C. Именно с его помощью в данном курсовом проекте реализуются алгоритмы построения лабиринта, перевода лабиринта в граф и его обхода.

# **Постановка задачи**

Нужно разработать программу, которая пройдет лабиринт, используя алгоритм поиска в глубину.

Сначала с помощью определенного алгоритма должен строиться лабиринт или считываться с уже сохранённого файла. Данный лабиринт будет сохранен в двумерный динамический массив, после обработки которого можно будет создать матрицу смежности. По созданной матрице смежности, используя алгоритм поиска в глубину, находится выход из лабиринта.

Программа должна быть поделена на логические модули, где будут находится отдельные функции программы. Это позволит в случае чего расширить функционал программы. К примеру, добавление новых алгоритмов построения лабиринта.

Режим работы видеосистемы-текстовый.

Устройства ввода информации-клавиатура. Нужно проверять что вводит пользователь во избежание ошибок.

Пользовательский интерфейс должен быть построен на основе меню и панели инструментов. Таким образом использование программы станет более интуитивно.

Задания выполняются в соответствии с вариантом №12.

# **Теоретическая часть**

Лабиринт — структура (обычно в двухмерном или трёхмерном пространстве), состоящая из запутанных путей к выходу (и/или путей, ведущих в тупик). В этой курсовой работе лабиринт представлен в виде графа и строится посредством алгоритма двоичного дерева [4].

Граф – совокупность точек, соединенных линиями. Точки называются вершинами, или узлами, а линии – ребрами, или дугами.

Графы также подразделяются на ориентированные и неориентированные.В ориентированном графе ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.В неориентированном графе по каждому из ребер можно осуществлять переход в обоих направлениях[3].

В данной курсовой работе используется неориентированный граф, представленный матрицей смежности.

Матрица смежности графа G (V, X) (рисунок 2) – граф, где V = {v1, v2, …, vn}, X={x1, x2,…, xm} называется квадратная матрица A(G)=[aij] (рисунок 1) порядка n, у которой

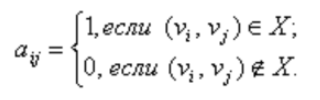


Рисунок 1 – условие матрицы смежности

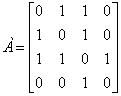


Рисунок 2 – пример матрицы смежности

# **Описание алгоритма**

Для программной реализации алгоритма понадобится четыре массива: maze(int\*\*) –для хранения лабиринта (0 – стенка, 1 – проход), G (int\*\*) – матрица смежности, построенная на основании массива maze, массив vis (int\*) – который хранит в себе все посещенные вершины (0 – не посещена, 1 – посещена) и массив parent (int\*) – который хранит в себе родителей всех вершин (родителем является предыдущая вершина).

После создания лабиринта и сохранения его в массив maze и текстовый файл, создается матрица смежности G, используя функцию Graph. Данная функция просматривает каждый элемент массива maze. Если он будет равен единицы (maze[i][j] = 1), то алгоритм просмотрит соседние элементы. Если они тоже будут равняться единицы (maze[i-1][j]=1; maze[i][j-1]=1; maze[i+1][j]=1; maze[i][j+1]=1), то функция Graph построит соответствующую связь в графе размером n2, где n – количество строк/столбцов в лабиринте.

Инициализируем массивы visи parent размером n2, где n – количество строк/столбцов в лабиринте и вызываем функцию DFS, которая реализует алгоритм поиска в глубину. Данная функция принимает на вход такие переменные как: G, vis, parent, output- выход из лабиринта, input - вход в лабиринт.

В качестве стартовой точки алгоритма поиска в глубину выбирается переменная input и ей приписывается что она посещена: vis[input] = 1. Далее мы находим связанные с ней вершины. Если вершина не посещена, и мы не дошли до конца, то заносим в массив parentинформацию что родителем данной вершины является input. Вызываем рекурсию и на вход уже подаем эту вершину и так пока выход не будет посещен (vis[output] = 1).

Так как алгоритм поиска в глубину в моей программе имеет рекурсивное представление, то каждый раз в консоль будет выводиться лабиринт с текущим местонахождением (будет появляться символ).

Ниже представлен псевдокод некоторых функций: Graph(), DFS().

**Graph**

int\*\* CreateGraph(int n, int \*\*Maze)

{

int SizeG = n \* n; //Размер графа = количество элементво массива лабиринта

int \*\*Graph = (int\*\*)malloc(sizeof(int) \* SizeG);

for (int i = 0; i < SizeG; i++)

{

Graph[i] = (int\*)malloc(SizeG \* sizeof(int));

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

{

Graph[i][j] = 0;

}

}

for (int i = 1; i < n - 1; i++)

{

for (int j = 1; j < n - 1; j++)

{

if (Maze[i][j] == 1)

{

if (Maze[i - 1][j] == 1)

{

Graph[i \* n + j][i \* n + j - n] = 1;

Graph[i \* n + j - n][i \* n + j] = 1;

}

if (Maze[i + 1][j] == 1)

{

Graph[i \* n + j][i \* n + j + n] = 1;

Graph[i \* n + j + n][i \* n + j] = 1;

}

if (Maze[i][j - 1] == 1)

{

Graph[i \* n + j][i \* n + j - 1] = 1;

Graph[i \* n + j - 1][i \* n + j] = 1;

}

if (Maze[i][j + 1] == 1)

{

Graph[i \* n + j][i \* n + j + 1] = 1;

Graph[i \* n + j + 1][i \* n + j] = 1;

}

}

}

}

return Graph;

**DFS**

void DFS(int\*\* Graph, int\*\* Maze, int\* Visited, int\* Parent, int NumCells, int Input, int Exit)

{

int SizeG = NumCells \* NumCells;

Visited[Input] = 1;

Maze[Input / NumCells][Input % NumCells] = 2;

for (int i = 0; i < NumCells; i++)

{

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

{

if (Maze[i][j] == 0) printf(" \*");

if (Maze[i][j] == 1) printf(" ");

if (Maze[i][j] == 2) printf(" +");

}

printf("\n");

}

printf("\n");

Sleep(100);

system("cls");

Maze[Input / NumCells][Input % NumCells] = 1;

for (int i = 0; i < SizeG; i++)

{

if (Graph[Input][i] == 1 && Visited[i] == 0 && Visited[Exit] == 0)

{

Parent[i] = Input;

DFS(Graph, Maze, Visited, Parent, NumCells, i, Exit);

}

}

for (int i = 0; i < SizeG; i++)

{

if (Visited[Exit] == 0)

{

Input = Parent[Input];

DFS(Graph, Maze, Visited, Parent, NumCells, Input, Exit);

}

}

}

# **Описание программы**

Разработанная программа состоит из нескольких модулей, которые реализовывались в следующем порядке:

1. Source.cpp – основной файл.
2. GenerMaze.cpp – файл, который проводит с лабиринтом какие-то операции (файл генерации лабиринта).
3. PrintMazeFromFiles.cpp – файл, который проводит с лабиринтом какие-то операции (файл загрузки лабиринта)
4. InitGrid.cpp – файл создания сетки лабиринта.
5. PrintMazeToFiles.cpp – сохраняет лабиринт в файл.
6. CreateGraph.cpp – создает матрицу смежности на основе лабиринта.
7. DFS.cpp – алгоритм поиска в глубину и вывод лабиринта на экран.

Исходя из порядка реализации модулей можно понять, что используется нисходящий подход реализации программы. Сначала создаются основные компоненты программы, а потом добавляются мелкие детали для большей удобности использования программы.

Программа предназначена для реализации поиска путей в лабиринте. Программа содержит понятный интерфейс, а также такие операции как загрузка или сохранение лабиринта и изменение размера лабиринта.

В начале вы можете выбрать как будет строиться лабиринт (создать его или загрузить уже сохраненный) (рисунок 3).

int main()

{

char\* Check = (char\*)malloc(100 \* sizeof(char));

setlocale(LC\_ALL, "RU");

printf("Загрузить лабиринт или сгенерировать? (сгенерировать - 1, загрузить - 2)\n");

if (Check)

{

memset(Check, 0, 100);

do

{

scanf("%s", Check);

if (Check[0] != '1' && Check[0] != '2')

{

printf("Введите правильное значение\n");

}

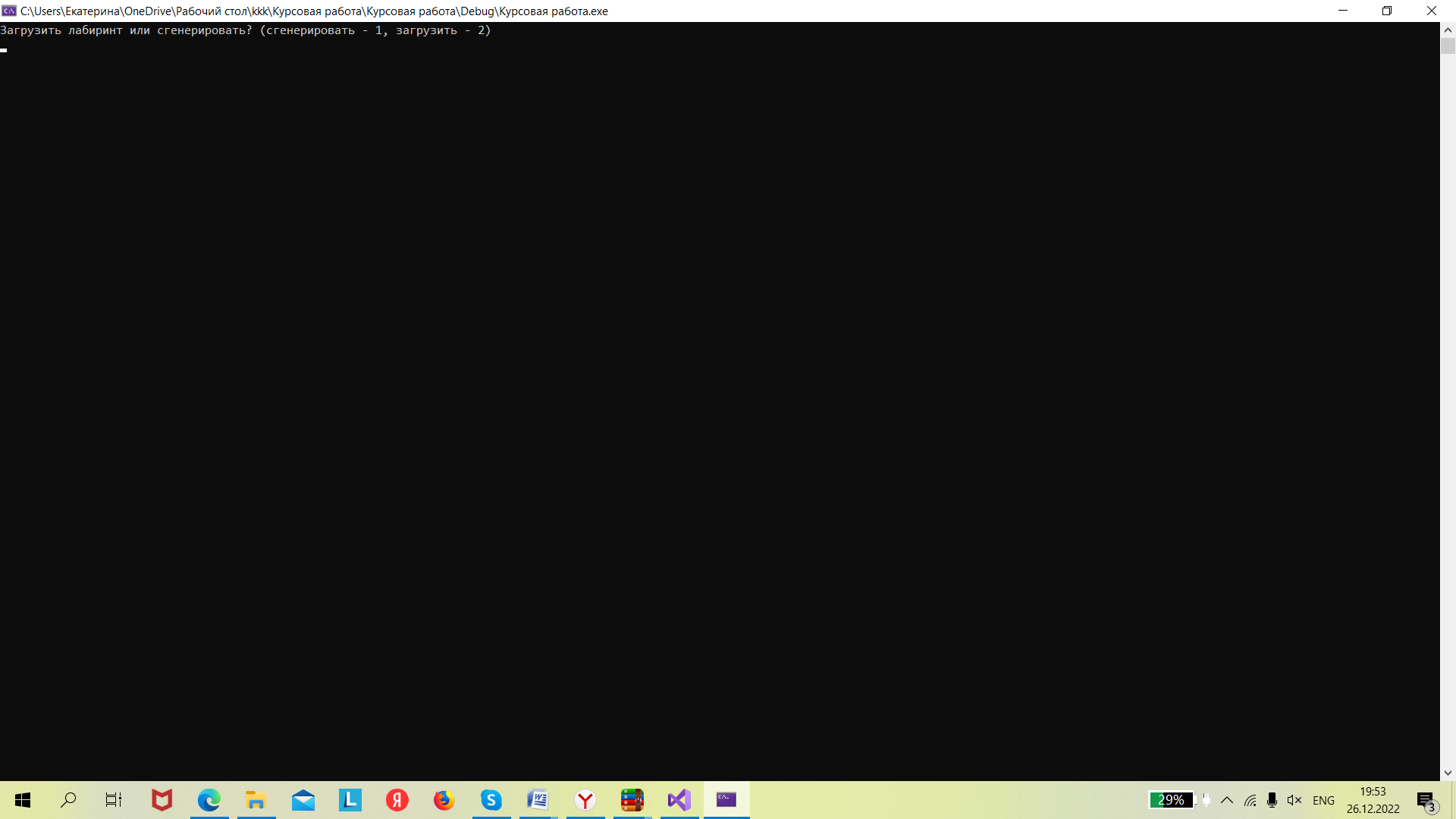
} while (Check[0] != '1' && Check[0] != '2'); 

Рисунок 3 – интерфейс программы

Если вы выбрали загрузку лабиринта, тогда вам потребуется ввести имя файла, в котором хранится данный лабиринт (рисунок 4).

void PrintMazeToFiles(int\*\* Maze, int NumCells, int Entry, int Exit)

{

char filename[25];

FILE\* file;

printf("Введите имя файла для того чтобы сохранить лабиринт (формата .txt)\n");

do

{

scanf("%s", filename);

file = fopen(filename, "w");

if (file == NULL)

{

printf("Имя файла не соответствует регламенту\n");

}

} while (file == NULL);

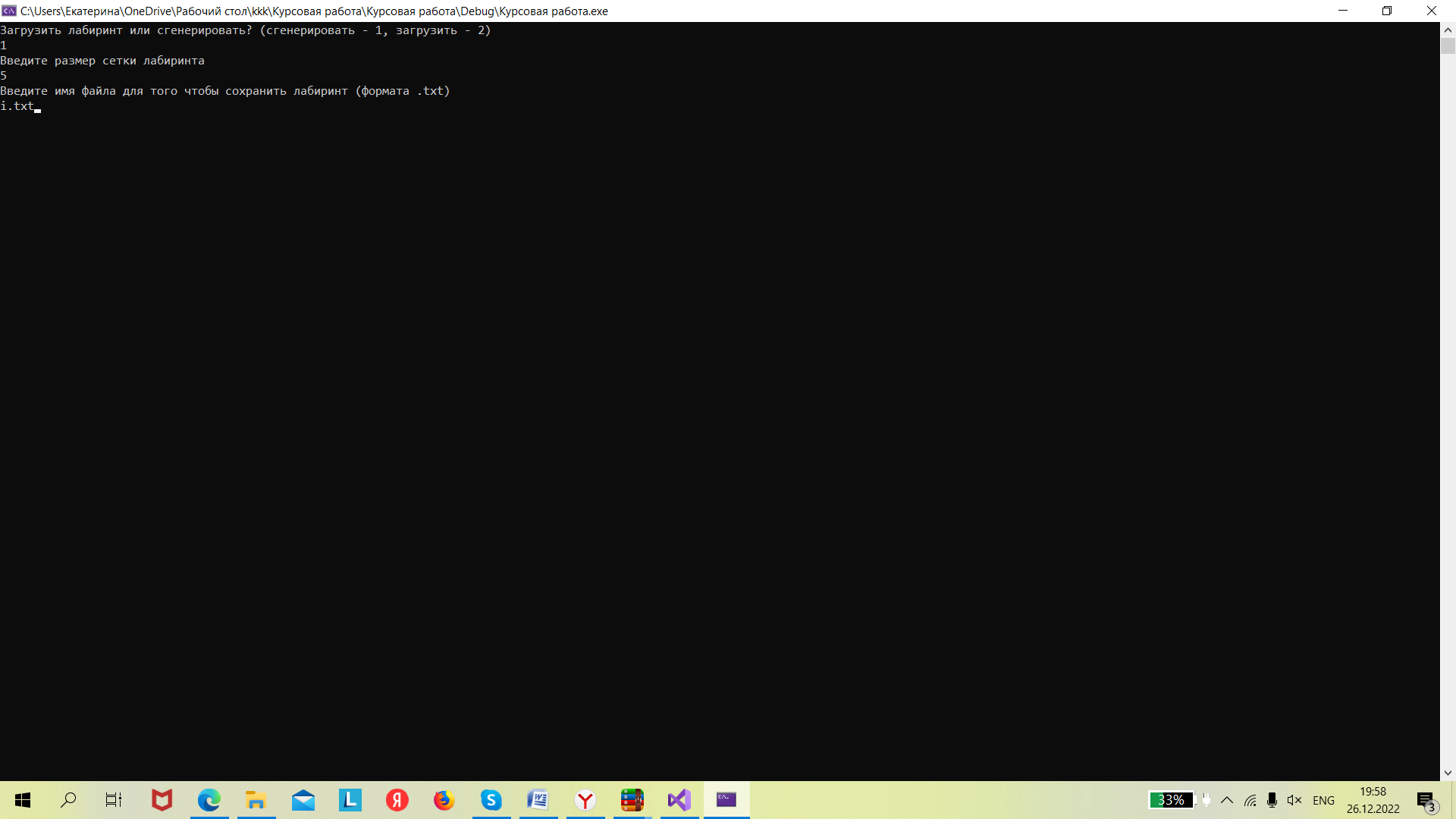


Рисунок 4 – вызов генерации лабиринта

Дальше на экран выводится лабиринт и начинается поиск пути. Пример лабиринта приведен на рисунке 5. Плюс является нынешней точкой (вершиной) обхода в глубину.

for (int i = 0; i < NumCells; i++)

{

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

{

if (Maze[i][j] == 0) printf(" \*");

if (Maze[i][j] == 1) printf(" ");

if (Maze[i][j] == 2) printf(" +");

}

printf("\n");

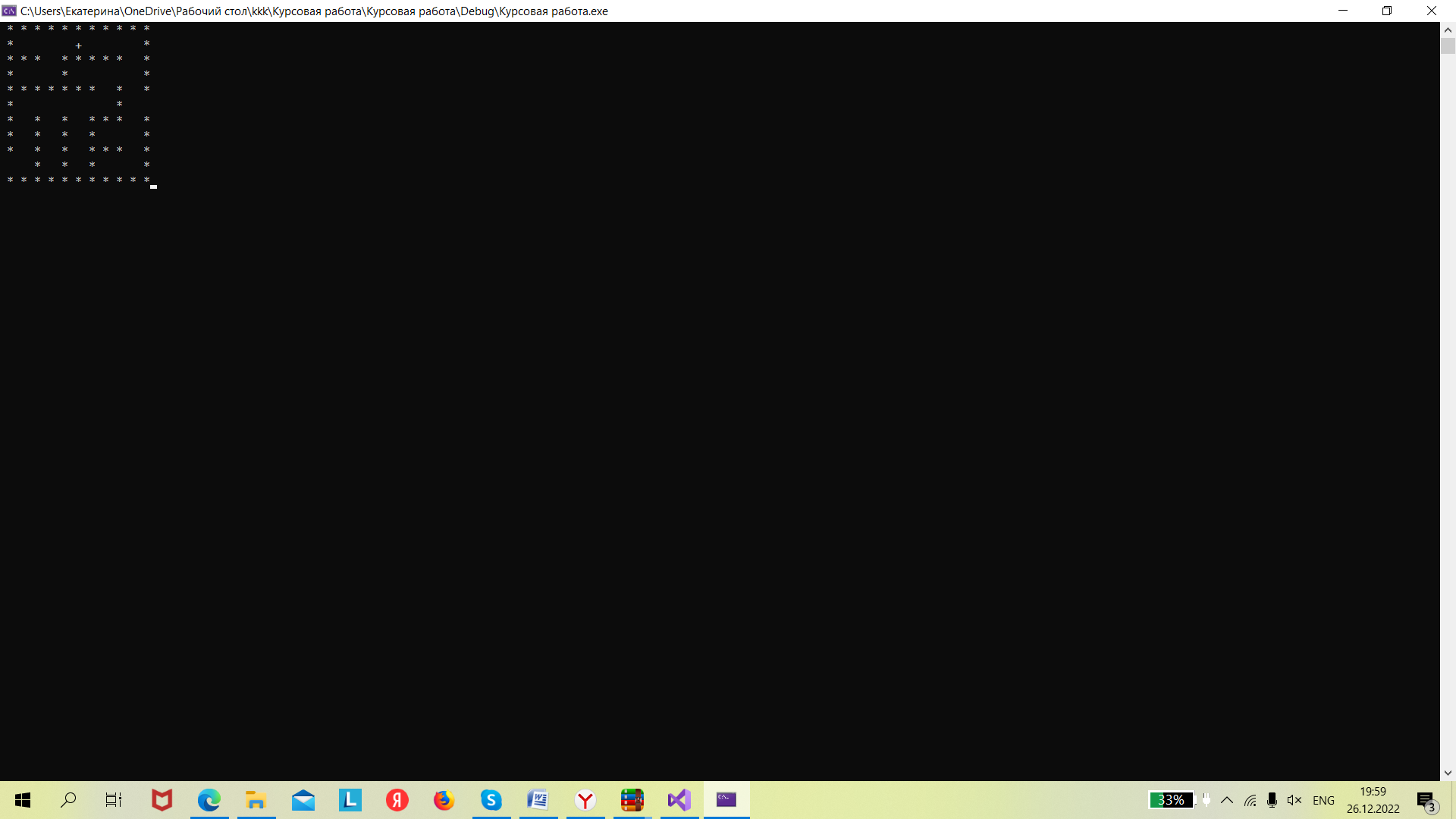
}

Рисунок 5 – пример работы лабиринта

После этого на экран выводятся поздравления (рисунок 6).

if (Check[0] == '1')

{

GenerMaze(); // Генерация лабиринта (по инициализированной сетке)

printf("\nПоздравляю вы прошли лабиринт!!!\n");

}

else

{

PrintMazeFromFile(); // Считывание лабиринта с файла и вывод в консоль

printf("\nПоздравляю вы прошли лабиринт!!!\n");

}

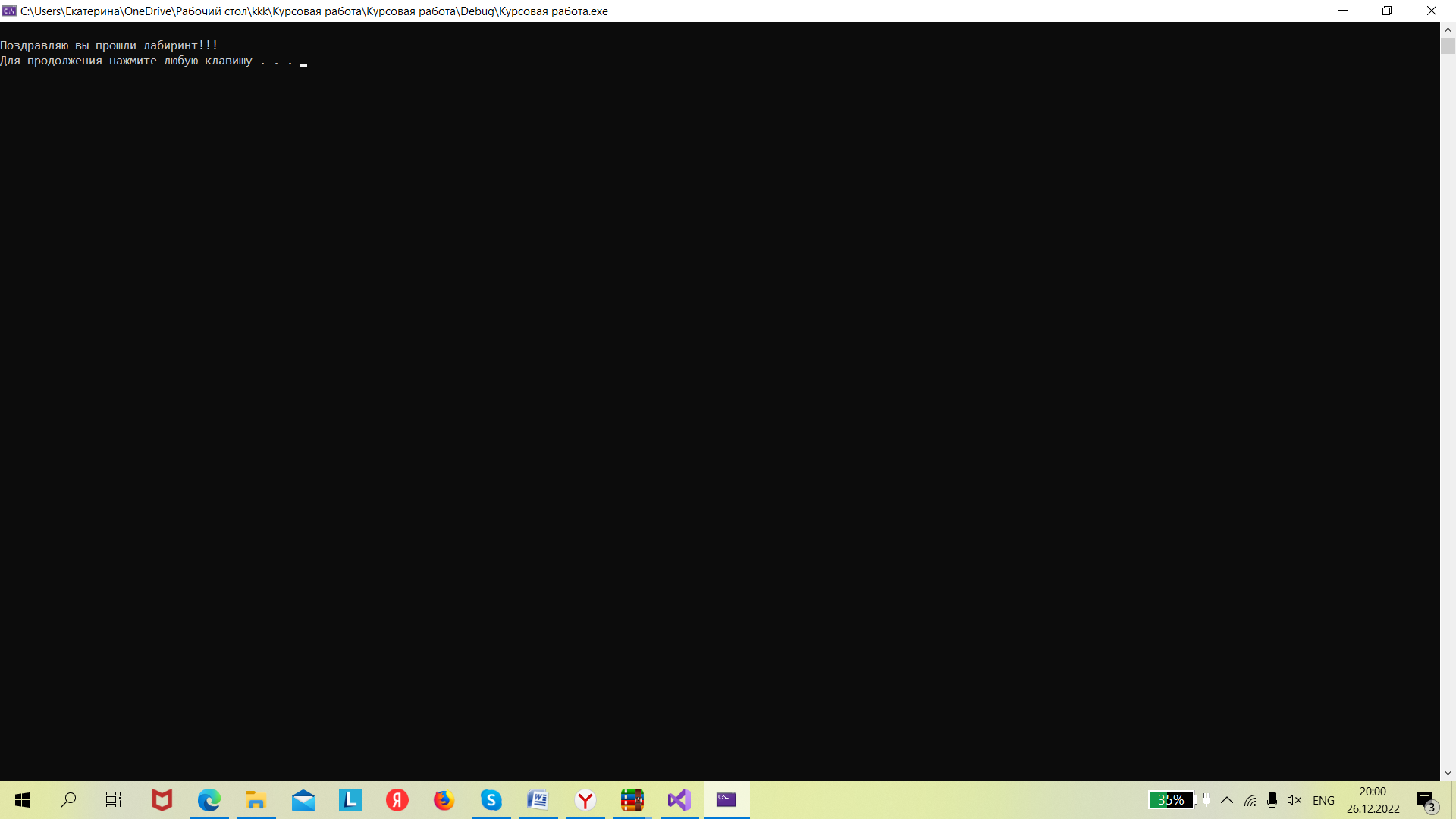


Рисунок 6 – результат работы лабиринта

Если же пользователь решил сгенерировать новый лабиринт, то ему предлагается вручную ввести размер лабиринта и сохранить его в какой-то файл (рисунок 7). После этого вывод в консоли ничем не отличается от загрузки лабиринта с файла.

char filename[25];

FILE\* file;

printf("Введите имя файла для того чтобы сохранить лабиринт (формата .txt)\n");

do

{

scanf("%s", filename);

file = fopen(filename, "w");

if (file == NULL)

{

printf("Имя файла не соответствует регламенту\n");

}

} while (file == NULL);

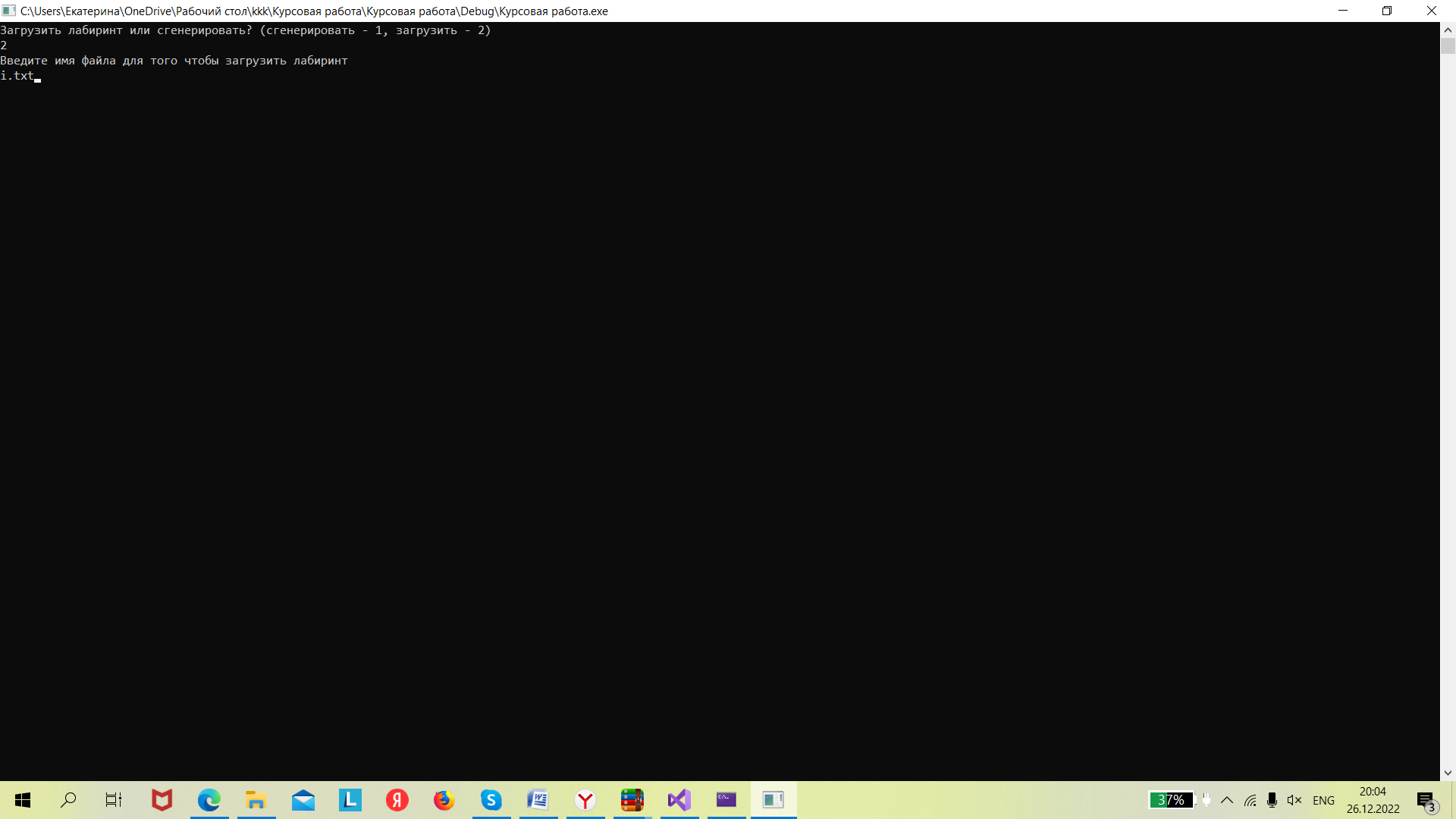


Рисунок 7 – вызов загрузки в файл

# **Тестирование**

В качестве среды разработки была выбрана программа MicrosoftVisualStudio 2019. Для отладки использовалось несколько возможностей данной программы. Такие как трассировка, точка остановки, шаг с обходом.

Тестирование проводилось на всех этапах разработки программы. В ходе тестирования было исправлено некоторое количество проблем, связанных с вводом данных, выводом лабиринта, работой алгоритма построения лабиринта и алгоритма обхода в глубину.

Ниже продемонстрирован результат тестирования программы при вводе пользователем различных данных.

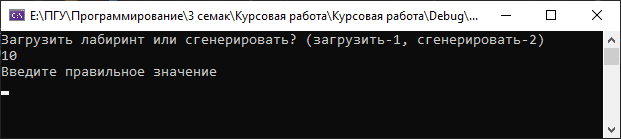


Рисунок 8 - вывод сообщения, если пользователь допустил ошибку

Если пользователь сохранит лабиринт в уже заполненный файл, то лабиринт перезапишется (рисунок 9-10).

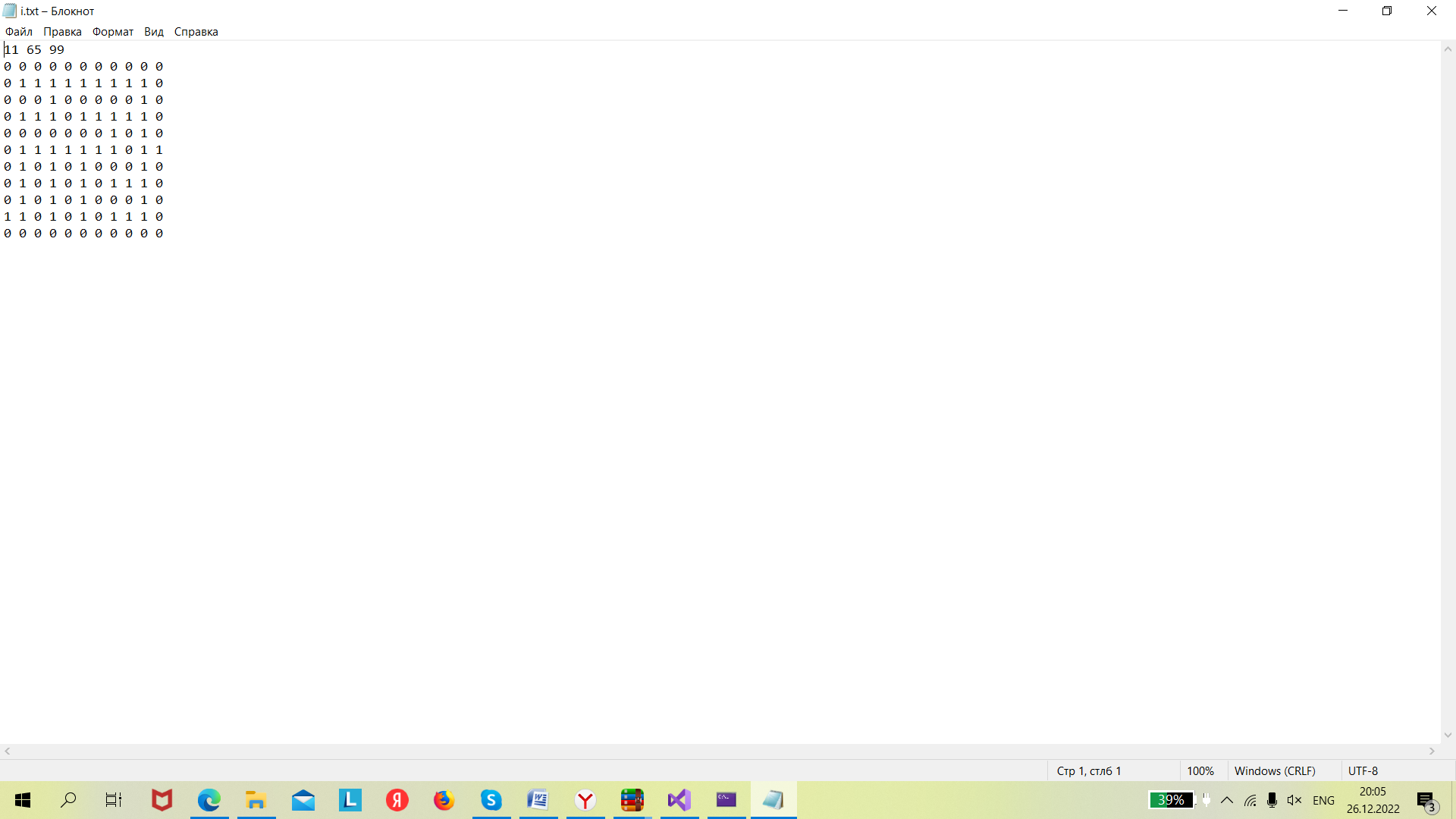


Рисунок 9

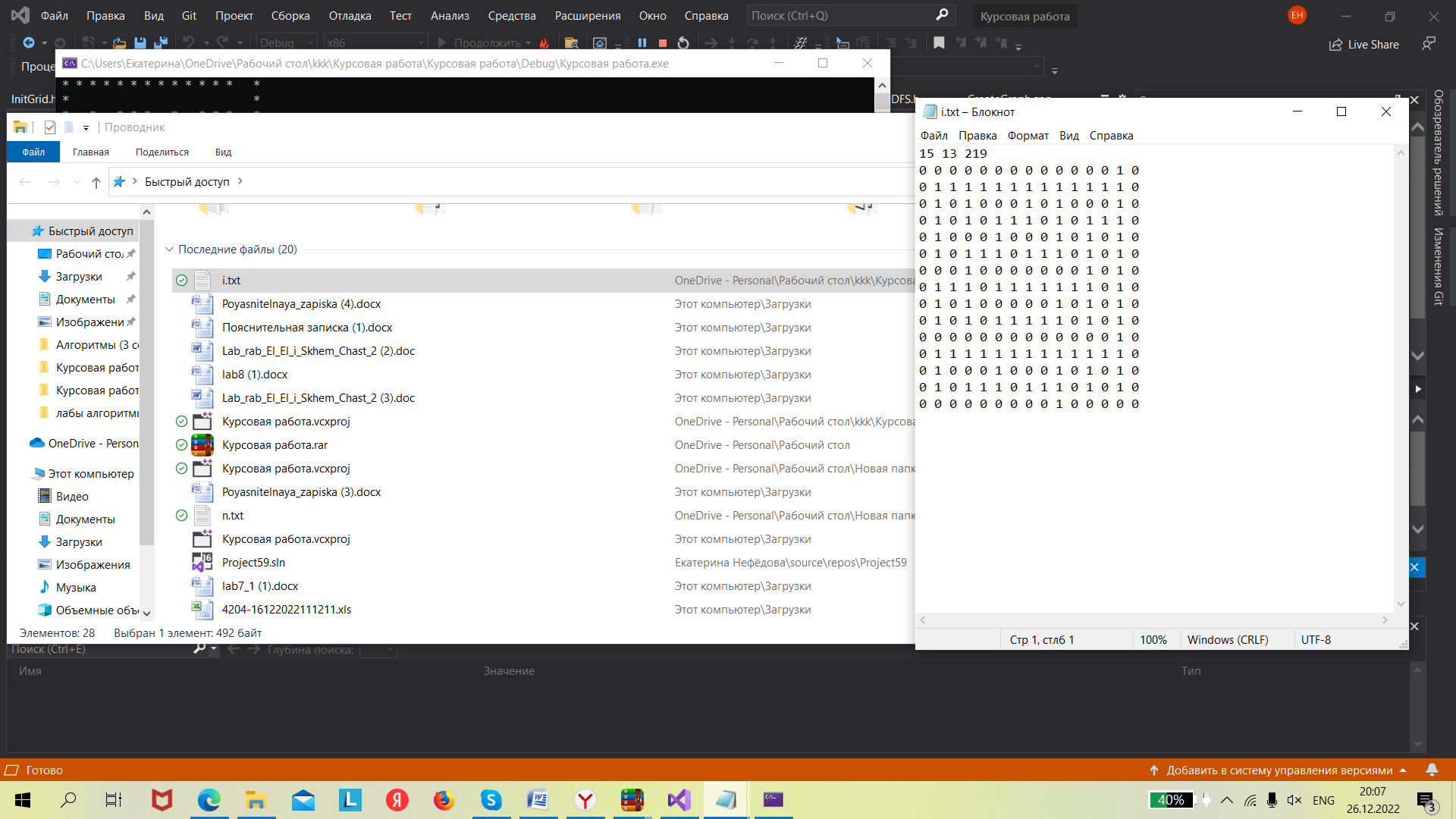


Рисунок 10

Таблица 1 – Описание поведения программы при тестировании

|  |  |  |
| --- | --- | --- |
| Описание теста | Ожидаемый результат | Полученный результат |
| Запуск программы | Вывод сообщения о выборе создания лабиринта | Верно |
| Проверка на ошибки ввода | Вывод сообщения о том что значение не верное | Верно |
| Проверка на ввод размера лабиринта | Вывод сообщения о том куда сохранять лабиринт | Верно |
| Проверка вывод лабиринта на экран | По лабиринту должен бегать плюсик | Верно |
| Проверка вывода поздравлений | Программа должна вывести сообщение | Верно |

В результате тестирования было успешно установлено, что программа после изменений всех недочетов, работает исправно.

# **Ручной расчет задачи**

Проведем проверку программы посредством ручных вычислений на примере лабиринта с размерной сеткой равной одному. После того как создается полноценный лабиринт, на его основе формируется матрица смежности (рисунок 11).

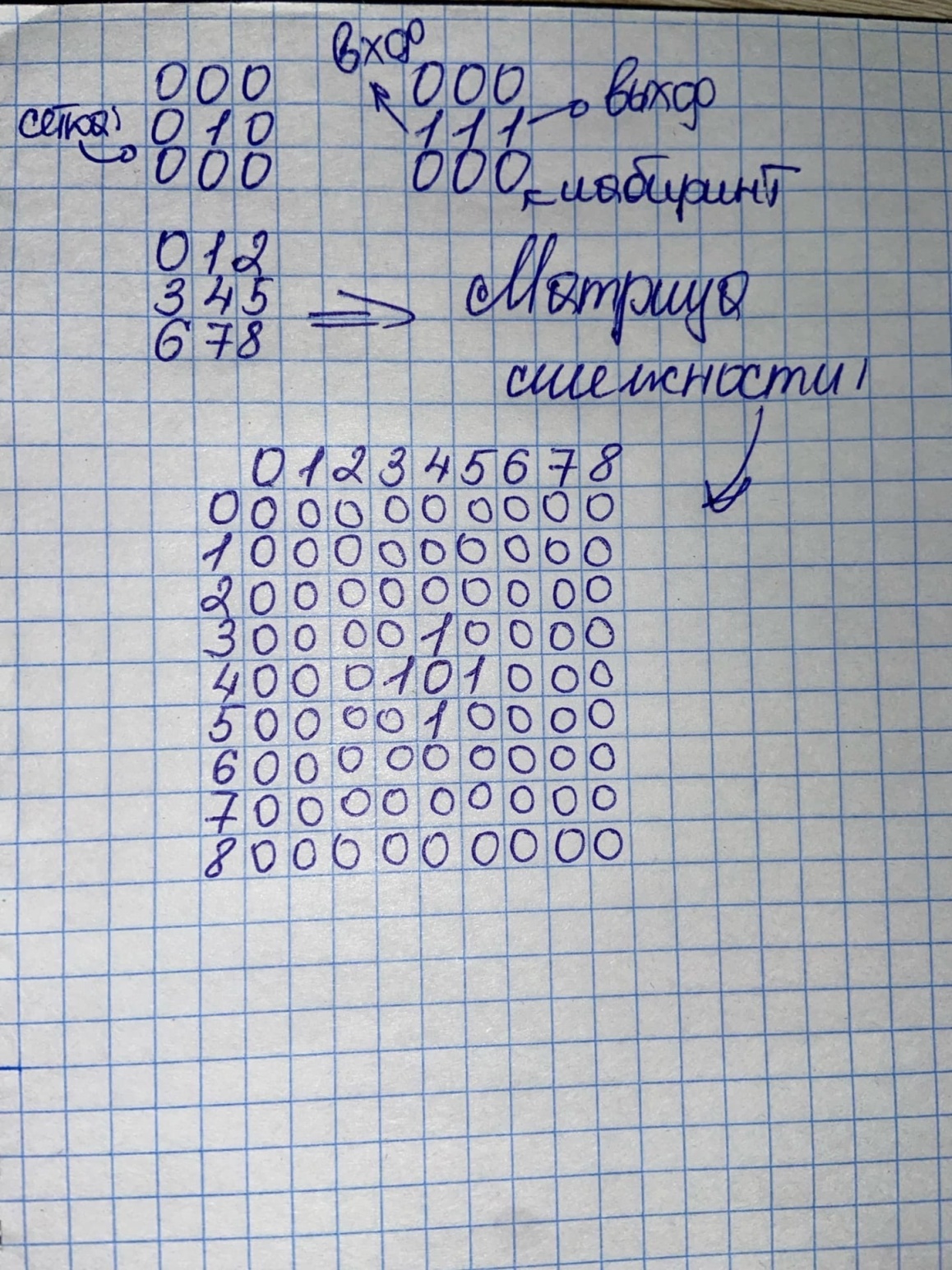


Рисунок 11 – ручной расчет

После того как построилась матрица смежности DFS начинает проходить ее до момента пока vis[3] не будет равняться 1 (выходом является 3 вершина).

В качестве входного значения берется 5 вершина. Отмечаем, что она посещена (vis[5] = 1). Дальше проходим матрицу и находим что G[5][4] = 1, и vis[4] = 0, и vis[3] = 0. Следовательно отмечаем что родителем 4 вершины является 5 (parent[4] = 5) и рекурсивно возвращаем 4 вершину. Отмечаем, что 4 вершина посещена (vis[4] = 1) и находим что G[4][3] = 1 и vis[3] = 0. Отмечаем что родителем 3 вершины является 4 (parent[3] = 4) и рекурсивно возвращаем 3 вершину. Отмечаем, что третья вершина посещена (vis[3] = 1) и на этом наш алгоритм заканчивает свой обход. Мы дошли до конца лабиринта.

Пример последовательного прохода графа в консоли (рисунок 12-15).

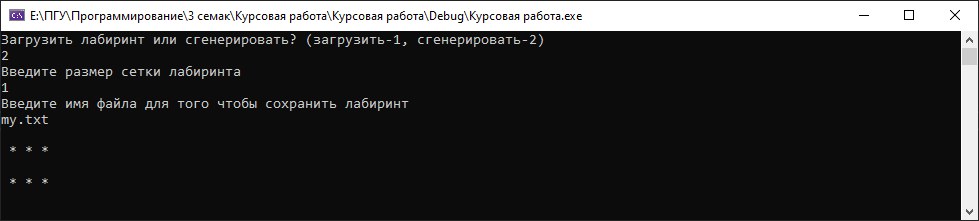


Рисунок 12 – генерация лабиринта

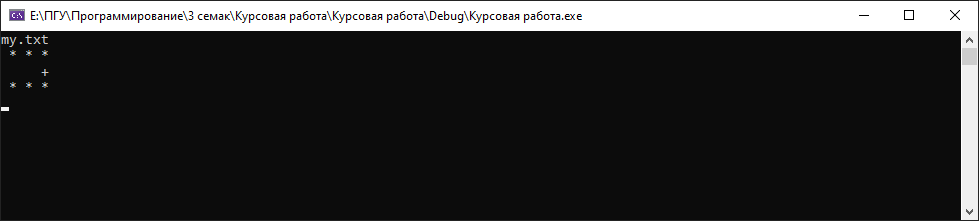


Рисунок 13 – поиск выхода

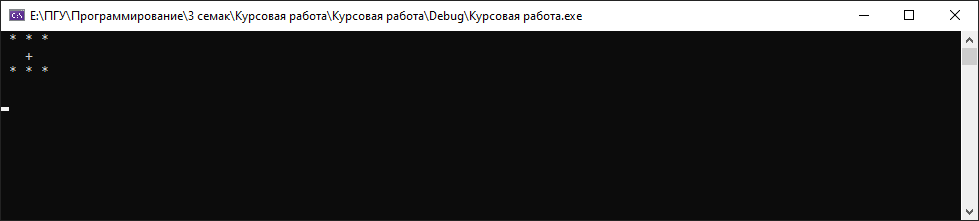


Рисунок 14 – работа лабиринта

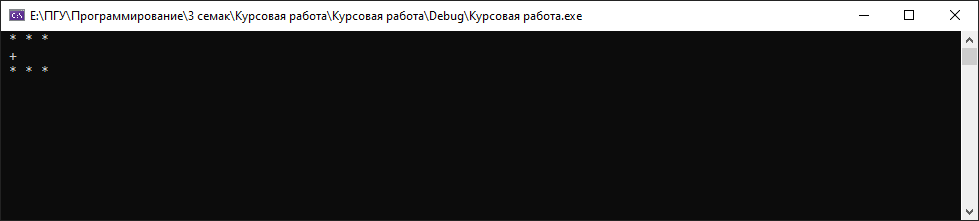


Рисунок 15 – найден выход

И так сравнив ручной расчет с результатами программы можно сделать вывод о ее полной правильности и работоспособности.

# **Заключение**

При выполнении данной курсовой работы были получены навыки разработки многомодульной программы с пользовательским интерфейсом. Также были получены навыки отладки и тестирования программ в MicrosoftVisualStudio 2019.

Мною был написан алгоритм поиска путей в лабиринте, были расширены знания языка Си. В ходе написания программы были улучшены навыки написания матриц смежности и алгоритма поиска в глубину.

# **Список используемых источников**

1. Прата С. Язык программирования С++. 2019 г.
2. А.А. Тюгашев. Языки программирования. Учебное пособие. 2018 г.
3. Графы: [https://prog-cpp.ru/data-graph/#height]
4. Классические алгоритмы генерации лабиринтов: [https://habr.com/ru/post/320140/]

# **Листингипрограммы**

Main.cpp:

#define\_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<locale.h>

#include"GenerMaze.h"

#include"UploadFiles.h"

usingnamespacestd;

intmain() {

int type;

setlocale(LC\_ALL, "RU");

cout<<"Загрузитьлабиринтилисгенерировать? (загрузить-1, сгенерировать-2)"<<endl;

do {

cin>> type;

if (type != 1 && type != 2) {

cout<<"Введите правильное значение"<<endl;

}

} while (type != 1 && type != 2);

if (type == 2) {

GenerMaze(); // Генерирует лабиринт (по инициализированной сетке)

cout<<endl<<"Congratulations!!!";

}

else {

UploadFiles(); // Считывает лабиринт с файла и выводит его в консоль

cout<<endl<<"Congratulations!!!";

}

return 0;

}

GenerMaze.cpp:

#include<time.h>

#include<iostream>

#include"InitGrid.h"

#include"PrintMazeFiles.h"

#include"Graph.h"

#include"DFS.h"

usingnamespacestd;

voidGenerMaze() {

inti, j, \*\* maze, n, input, output, \*vis, \*parent, \*\*G, ran, SizeG;

srand(time(NULL));

cout<<"Введитеразмерсеткилабиринта"<<endl;

cin>> n;

n = 2 \* n + 1;

maze = (int\*\*)malloc(sizeof(int) \* n);

InitGrid(n, maze); // Инициализирует сетку (n - количество клеток сетки)

for (i = 1; i < n; i += 2) { // Алгоритм создания лабиринта (двоичное дерево)

for (j = 1; j < n; j += 2) {

if (rand() % 2 == 0) {

if (j != n - 2) {

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

}

else {

maze[i - 1][j] = 1;

}

}

else {

if (i> 1) { maze[i - 1][j] = 1; }

else {

if (j != n - 2) {

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

}

else {

maze[i - 1][j] = 1;

}

}

}

}

}

maze[0][n - 2] = 0;

if (rand() % 2 == 0) {

ran = rand() % n;

while (ran % 2 == 0) {

ran = rand() % n;

}

maze[0][ran] = 1; // входсверху

input = ran;

ran = rand() % n;

while (ran % 2 == 0) {

ran = rand() % n;

}

maze[n - 1][ran] = 1; //выходснизу

output = n \* (n - 1) + ran;

}

else {

ran = rand() % n;

while (ran % 2 == 0) {

ran = rand() % n;

}

maze[ran][n - 1] = 1; // входсправа

input = n \* (ran + 1) - 1;

ran = rand() % n;

while (ran % 2 == 0) {

ran = rand() % n;

}

maze[ran][0] = 1; // выходслева

output = n \* ran;

}

PrintMazeFiles(maze, n, input, output); // Выводит на экран лабиринт и сохраняет его в файл

G = Graph(n, maze); // Создает матрицу смежности по созданному лабиринту

SizeG = n \* n; // Размер графа = количество элементво массива лабиринта

vis = (int\*)malloc(SizeG \* sizeof(int));

mass(SizeG, vis);

parent = (int\*)malloc(SizeG \* sizeof(int));

mass(SizeG, parent);

DFS(n, input, vis, G, parent, output, maze);

}

UploadFiles.cpp:

#define\_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include"Graph.h"

#include"DFS.h"

usingnamespacestd;

voidUploadFiles() {

int n, \*\*maze, \*\*G, SizeG, \*vis, \*parent, input, output;

charfilename[25];

FILE\* file;

cout<<"Введитеимяфайладлятогочтобызагрузитьлабиринт"<<endl;

do

{

cin>> filename;

file = fopen(filename, "r");

if (file == NULL)

{

cout<<"Имя файла не соответствует регламенту"<<endl;

}

} while (file == NULL);

fscanf(file, "%d ", &n);

fscanf(file, "%d ", &input);

fscanf(file, "%d ", &output);

maze = (int\*\*)malloc(sizeof(int) \* n);

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

maze[i] = (int\*)malloc(sizeof(int) \* n);

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

fscanf(file, "%d", &maze[i][j]);

}

}

fclose(file);

G = Graph(n, maze);

SizeG = n \* n;

vis = (int\*)malloc(SizeG \* sizeof(int));

mass(SizeG, vis);

parent = (int\*)malloc(SizeG \* sizeof(int));

mass(SizeG, parent);

DFS(n, input, vis, G, parent, output, maze);

}

InitGrid.cpp:

#include<malloc.h>

voidInitGrid(intn, int\*\* grid) {

inti, j;

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

grid[i] = (int\*)malloc(n \* sizeof(int));

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

if (i % 2 == 0) {

grid[i][j] = 0;

}

else {

if (j % 2 == 0) {

grid[i][j] = 0;

}

else{ grid[i][j] = 1; }

}

}

}

}

PrintMazeFiles.cpp:

#define\_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

usingnamespacestd;

voidPrintMazeFiles(int\*\* mass, intn, intinput, intoutput) {

inti, j;

charfilename[25];

FILE\* file;

cout<<"Введите имя файла для того чтобы сохранить лабиринт"<<endl;

do

{

cin>> filename;

file = fopen(filename, "w");

if (file == NULL)

{

cout<<"Имя файла не соответствует регламенту"<<endl;

}

} while (file == NULL);

cout<<endl;

fprintf(file, "%d ", n);

fprintf(file, "%d ", input);

fprintf(file, "%d ", output);

fprintf(file, "\n");

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

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

if (mass[i][j] == 0) { cout<<" \*"; }

else{ cout<<" "; }

fprintf(file, "%d ", mass[i][j]);

}

cout<<endl;

fprintf(file, "\n");

}

fclose(file);

}

Graph.cpp:

#define\_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

usingnamespacestd;

int \*\*Graph(intn, int \*\*maze) {

int\*\* G, i, j;

intSizeG = n \* n; //Размер графа = количество элементво массива лабиринта

G = (int\*\*)malloc(sizeof(int) \* SizeG);

for (i = 0; i<SizeG; i++) {

G[i] = (int\*)malloc(SizeG \* sizeof(int));

for (j = 0; j <SizeG; j++) {

G[i][j] = 0;

}

}

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

for (j = 1; j <n - 1; j++) {

if (maze[i][j] == 1) {

if (maze[i - 1][j] == 1) {

G[i \* n + j][i \* n + j - n] = 1;

G[i \* n + j - n][i \* n + j] = 1;

}

if (maze[i + 1][j] == 1) {

G[i \* n + j][i \* n + j + n] = 1;

G[i \* n + j + n][i \* n + j] = 1;

}

if (maze[i][j - 1] == 1) {

G[i \* n + j][i \* n + j - 1] = 1;

G[i \* n + j - 1][i \* n + j] = 1;

}

if (maze[i][j + 1] == 1) {

G[i \* n + j][i \* n + j + 1] = 1;

G[i \* n + j + 1][i \* n + j] = 1;

}

}

}

}

return G;

}

DFS.cpp:

#include<iostream>

#include<Windows.h>

usingnamespacestd;

voidmass(intn, int \*mass) {

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

mass[i] = 0;

}

}

intDFS(intn, intnum, int\* vis, int\*\* G, int\* parent, intoutput, int\*\* maze) {

inti, j, SizeG = n \* n;

vis[num] = 1;

maze[num / n][num % n] = 2;

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

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

if (maze[i][j] == 0) { cout<<" \*"; }

if (maze[i][j] == 1) { cout<<" "; }

if (maze[i][j] == 2) { cout<<" +"; }

}

cout<<endl;

}

cout<<endl;

Sleep(100);

system("cls");

maze[num / n][num % n] = 1;

for (i = 0; i<SizeG; i++) {

if (G[num][i] == 1 &&vis[i] == 0 &&vis[output] == 0) {

parent[i] = num;

DFS(n, i, vis, G, parent, output, maze);

}

}

for (i = 0; i<SizeG; i++) {

if (vis[output] == 0) {

num = parent[num];

DFS(n, num, vis, G, parent, output, maze);

}

}

return 0;

}