Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра Информатики

Дисциплина: Программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

Реализация алгоритмов Мура и Дейкстры

на примере игры «Лабиринт»

Выполнил: студент: гр. 653501

Кацкель П.С.

Руководитель: ассистент кафедры информатики Козуб В.Н.

Минск 2017

**Оглавление**

[Введение 3](#_Toc482476563)

[Анализ предметной области 4](#_Toc482476564)

[Постановка задачи 4](#_Toc482476565)

[Теоретическая часть 4](#_Toc482476566)

[Исследование аналогов 5](#_Toc482476567)

[Разработка программы 6](#_Toc482476568)

[Описание структур данных 6](#_Toc482476569)

[Математический аппарат 7](#_Toc482476570)

[Работа программы: 10](#_Toc482476571)

[Заключение 14](#_Toc482476572)

[Список литературы: 15](#_Toc482476573)

[Приложение 16](#_Toc482476574)

# **Введение**

В современном мире время играет очень важную роль в обществе. Время ходьбы от дома до университета, время, затраченное стоя в пробках, время передачи пакетов информации в компьютерных системах.

Перед человечеством уже давно возник вопрос о нахождении наиболее кратчайших путей для наиболее быстрого решения возникающих проблем. Уже более чем 50 лет назад были разработаны первые решения «Задачи кратчайших путей».

На данный момент существует множество алгоритмов поиска минимального пути:

1. Алгоритмы Дейкстры, Беллмана — Форда находит кратчайший путь от одной точки до любой из остальных.
2. Алгоритм поиска A\* находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).
3. Алгоритм Флойда — Уоршелла находит кратчайшие пути между точками.
4. Алгоритм Джонсона находит кратчайшие пути между всеми парами точек.
5. Алгоритм Мура (волновой алгоритм) основан на методе поиска в ширину. Находит путь между двумя вершинами, содержащий минимальное количество промежуточных точек.

В данной работе будут исследованы алгоритмы Мура и Дейкстры. Для этого была написана программа на языке программирования С++ в среде разработки С++ Builder XE. Программа наглядно показывает работы этих алгоритмов, удобна в использовании и не требует особых навыков работы с компьютером.

# **Анализ предметной области**

## *Постановка задачи*

Целью курсовой работы является написание программы, способной находить кратчайшие пути прохода по лабиринту. Для этого она должна выполнять следующие функции:

1.1. Построение лабиринта пользователем

1.2. Выбор начальных и конечных точек пути

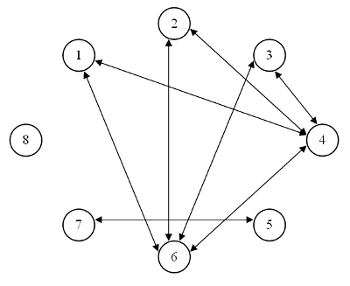
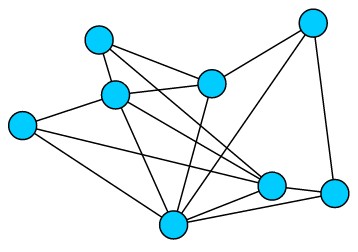
1.3. Поиск кратчайшего пути от начальной до конечной точки

## *Теоретическая часть*

Для решения поставленной задачи будем использовать графы.

*Граф* — абстрактный математический объект, представляющий из себя множество вершин графа и набор рёбер, то есть соединений между парами вершин. Например, за множество вершин можно взять множество аэропортов, обслуживаемых некоторой авиакомпанией, а за множество рёбер взять регулярные рейсы этой авиакомпании между городами.

На языке математики, граф G – это упорядоченная пара G : (V, E), где V – это непустое множество вершин (точек, узлов) , а E – множество упорядоченных/неупорядоченных пар вершин, называемых рёбрами.

Примеры графов:

Вершины u и v называются концевыми вершинами (или просто концами) ребра e = {u,v}. Ребро, в свою очередь, соединяет эти вершины. Две концевые вершины одного и того же ребра называются соседними.

Два ребра называются смежными, если они имеют общую концевую вершину.

Маршрутом в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершиной ребром.

## *Исследование аналогов*

Для работы с графами применяются следующие программные средства:

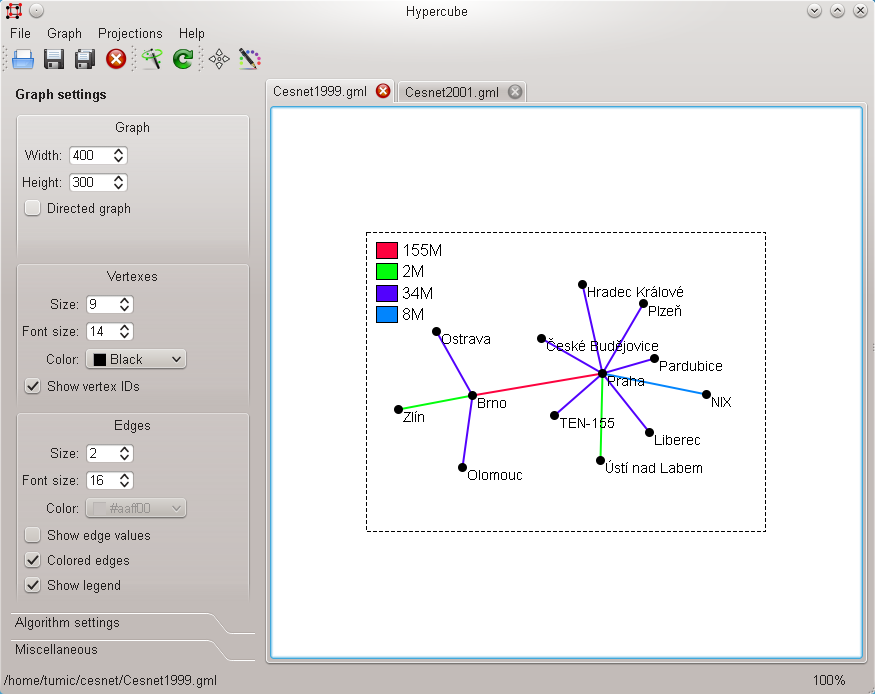
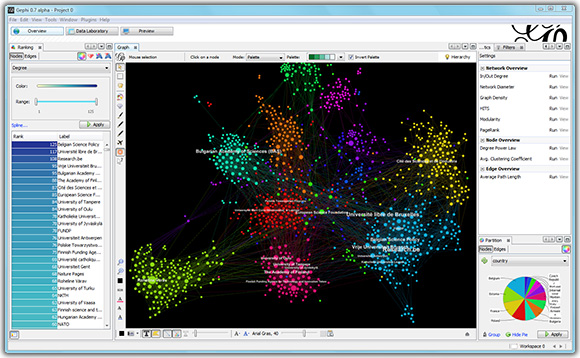
А) Graphviz

Б) LION Graph Visualizer.

В) Графоанализатор — русскоязычная программа, с простым пользовательским интерфейсом (только для Windows).

Г) Gephi — графическая оболочка для представления и изучения графов.

Д) Библиотека GraphX — свободная библиотека для .NET для расчёта и отрисовки графов, использует WPF 4.0.

Е) Библиотека MSAGL — свободная библиотека для .NET.

Hypercube

Gephi

Однако данные программы предназначены в основном для построения и визуализации графов, а не для использования алгоритмов поиска кратчайших путей. К тому же, алгоритм Мура в большей мере используется при создании печатных плат, нежели в программном обеспечении, потому его реализация используется крайне редко. Более того, нет аналогов для «игровой» версии реализации алгоритмов поиска путей. Это всё послужило причиной написания проекта.

# **Разработка программы**

## *Описание структур данных*

Лабиринт состоит из игрового поля, описанного структурой Field. Однако перед этим опишем структуру одной клетки лабиринта – Points:

struct Points

{

bool isPassed; //переменная, показывающая, была ли пройдена

//алгоритмом данная точка

TColor color; //цвет точки

int distance; //расстояние от начальной точки поиска пути до

//данной

int posX, posY; //позиция точки в лабиринте

int GetDistance(); //метод, позволяющий определить «проходимость»

//данной точки в зависимости от её цвета

};

Структура данных Field следующая:

struct Field

{

int horizontalNumber; //количество клеток по горизонтали

int verticalNumber; //количество клеток по вертикали

bool havePath; //переменная, отвечающая за то, нарисован на

//данный момент на поле путь

Points \*begin, \*end; //указатели на начальную и конечную точки

Points \*\*points; //указатель на двумерный массив точек Points

//методы работы с объектом Field

Field(int \_horizontalNumber, int \_verticalNumber); //конструктор

~Field(); //деструктор

void Initialize(int distance); //инициализатор точек поля

void DrawField(TImage\* image); //рисование поля в

//комноненте TImage

//рисование кратчайшего пути из данной точки в начальную

void DrawPath(TImage \*image, Points \*point);

//рисование линии из точки start в точку end цвета TColor

void DrawLine(TImage \*image, Points \*start, Points \*finish, TColor color);

//раскраска точек поля в нужные цвета

void PaintField(TImage\* image);

//изменение цвета точки поля

void ChangeFieldColor (TImage\* image, Points\* point);

//установление начальной точки поиска пути

void SetBeginPoint(TImage \*image, int xCount, int yCount);

//установление конечной точки поиска пути

void SetEndPoint(TImage \*image, int xCount, int yCount);

## *Математический аппарат*

Рассмотрим реализацию наших алгоритмов:

1. Алгоритм Мура

Работа алгоритма включает в себя 3 этапа:

А) Инициализация. Всем точкам приписывается, что они не были пройдены алгоритмом. Начальной точке присваивается дистанция «0».

Б) Распространение волны. От точки начинаются шаги в соседние (соседнюю по стороне) клетки, при этом проверяется, были ли они уже пройдены. Если нет, то им присваивается [дистанция клетки, из которой был сделан шаг + 1], также помечается, что эта клетка была пройдена. Затем из этой точки выполняются аналогичные шаги в соседние точки. Алгоритм выполняется до тех пор, пока не будут пройдены все точки или конечная точка не будет помечена как «пройденная».

В) Восстановление пути. Когда известны все дистанции, алгоритм начинает из конечной точки искать точки, дистанция до которых от начальной на единицу меньше, чем до конечной. Найдя её (а она всегда существует, так как, если путь найден, то он проходит через эту точку), алгоритм рисует туда линию, а затем делает аналогичные действия до тех пор, пока не достигнет начальной точки.

1. Алгоритм Дейкстры

Работа алгоритма включает в себя также 3 этапа:

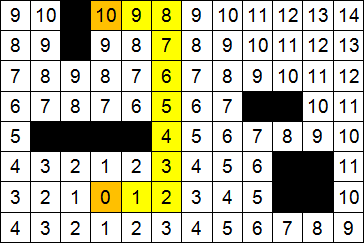
А) Инициализация. Всем точкам приписывается, что они не были пройдены алгоритмом. Начальной точке присваивается дистанция 0, остальным точкам – бесконечность (достаточно большое число).

Б) Шаг алгоритма. Если все вершины были посещены, то алгоритм завершается. Если нет, то находится ещё не посещённая вершина X, у которой минимальная дистанция до начальной точки. Затем проверяются все её соседние вершины (в нашем случае, соседние по сторонам клетки). Если сумма [дистанция X + расстояние до соседней клетки] больше дистанции этой соседней клетки, то её дистанция меняется на [дистанция X + расстояние до соседней клетки]. После данных действий со всеми соседями, клетка Х помечается как посещённая, и затем повторяется шаг алгоритма.

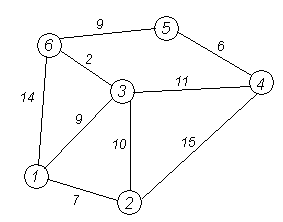
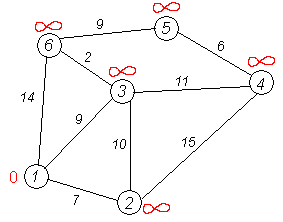
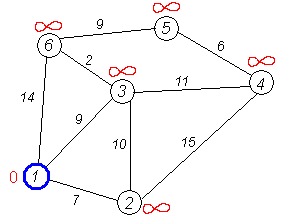
В) Восстановление пути. Когда известны все дистанции, алгоритм начинает из конечной точки искать точки, дистанция до которых от начальной меньше чем до конечной на величину, равную проходимости данной клетки. Найдя её (а она всегда существует, так как, если путь найден, то он проходит через эту точку), алгоритм рисует туда линию, а затем делает аналогичные действия до тех пор, пока не достигнет начальной точки.

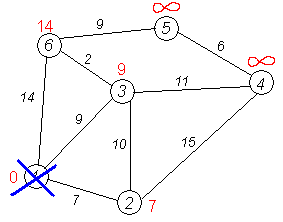
Разница между алгоритмами в том, что алгоритм Дейкстры должен проверить все точки и посчитать все дистанции до них, что затрачивает больше времени. Однако он позволяет находить оптимальные пути на взвешенных графах (то есть, в нашей задаче, с клетками разной проходимости). Алгоритм Мура работает гораздо быстрее, потому что завершает свою работу на моменте, когда «волна» достигла конечной точки. Однако этот алгоритм работает лишь на графах с одинаковыми весами вершин (в нашей задаче, клетки с одинаковой проходимостью).

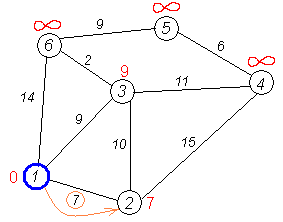
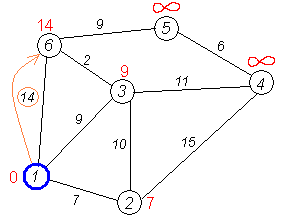
На рисунках наглядно показаны примеры алгоритмов:

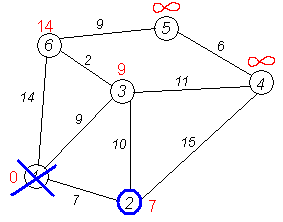
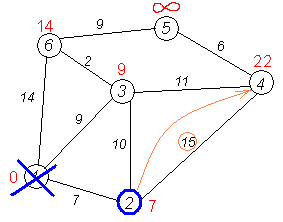


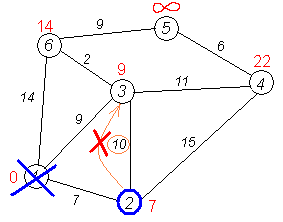
Алгоритм Мура

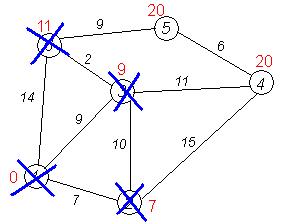


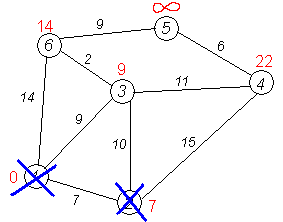
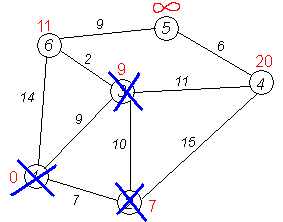


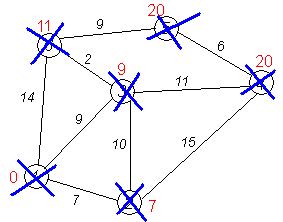
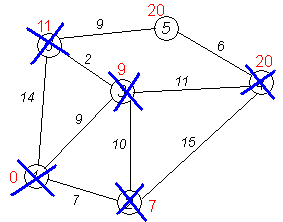






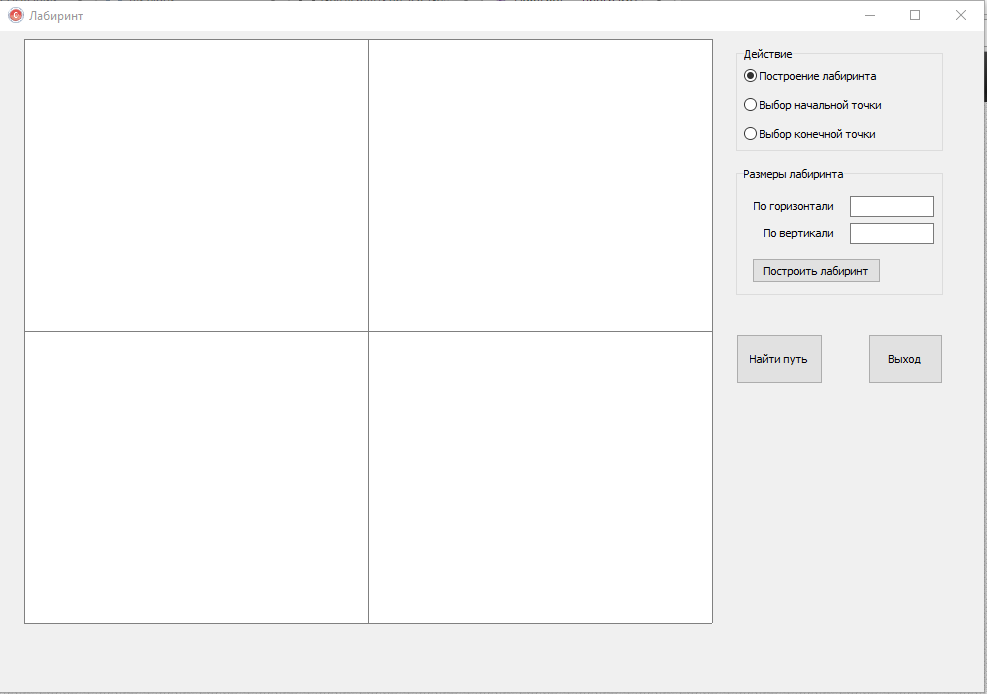






Алгоритм Дейкстры

# **Работа программы:**

 При запуске программы появится следующее окно:

Здесь уже создано поле по умолчанию размером 2x2, готовое к использованию. Помимо поля, программа имеет следующие компоненты

1. Компонент TRadioGroup «Действия»

Данные компонент содержит несколько радиокнопок, позволяющих работать с построением лабиринта. Рассмотрим каждую из них:

* 1. «Построение лабиринта». При выборе данной радиокнопки пользователь, нажимая на любую клетку лабиринта, будет изменять её цвет (то есть, проходимость) в указанном порядке по кругу:

- Белая клетка (проходимость 1)

- Жёлтая клетка (проходимость 2)

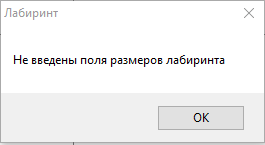
- Коричнево-зелёная клетка (проходимость 3)

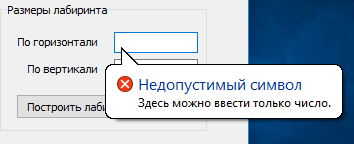
- Чёрная клетка (непроходима)

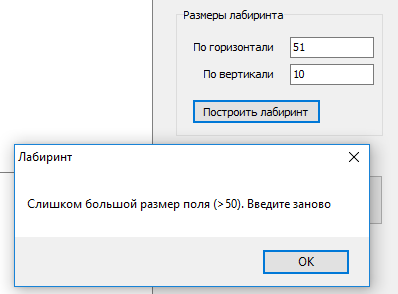
* 1. «Выбор начальной точки» и «Выбор конечной точки». При выборе данной радиокнопки пользователь, нажимая на любую клетку лабиринта, будет рисовать на ней зелёный(красный) кружок – стартовую(конечную) клетку, из которой будет начинаться(заканчиваться) искомый путь. При наличии уже на поле существующей стартовой(конечной) клетки она будет стираться, а новая рисоваться в месте, куда пользователь нажал мышью. Заметим, что данные точки нельзя ставить на чёрную (непроходимую клетку), то есть, подобные нажатия будут игнорироваться программой. Также, при нажатии на клетку с режимом «Построение лабиринта» на клетки, содержащие начальную/конечную точки, они будут стираться.

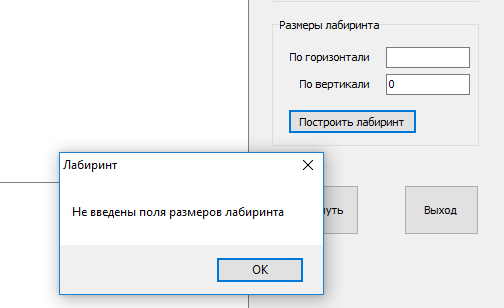
1. Компонент TGroupBox «Размеры лабиринта»

Данные объект содержит поля TEdit для ввода информации и кнопку «Построить лабиринт». При нажатии на данную кнопку будет взята информация из полей «По горизонтали» и «По вертикали», а затем построен новый лабиринт размером, указанным в этих полях. Все клетки нового поля будут окрашены в белый цвет, начальная и конечная точки будут стёрты.

Данные, введённые в полях «По горизонтали» и «По вертикали» должны быть целыми положительными числами. Если одно из полей не будет заполнено, выскочит следующее окно:

Помимо этого, компонент не будет позволять вводить что-либо, кроме чисел, предупредив при этом соответствующей подсказкой:

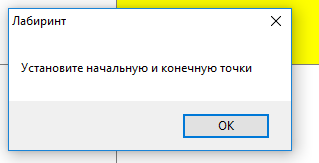
Так как при большом количестве клеток становится затруднительным, а иногда и невозможным редактирование лабиринта, было поставлено ограничение на максимальное количество строк и столбцов. Если одно из полей будет больше 50, высветится соответствующее сообщение:

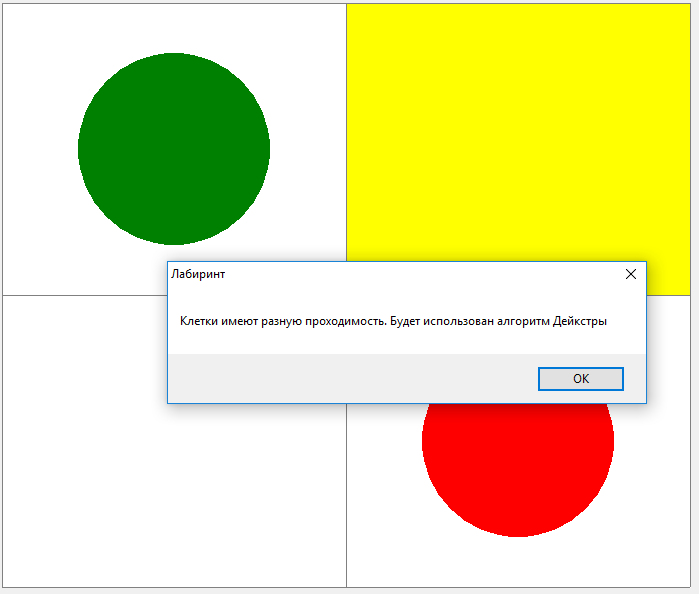
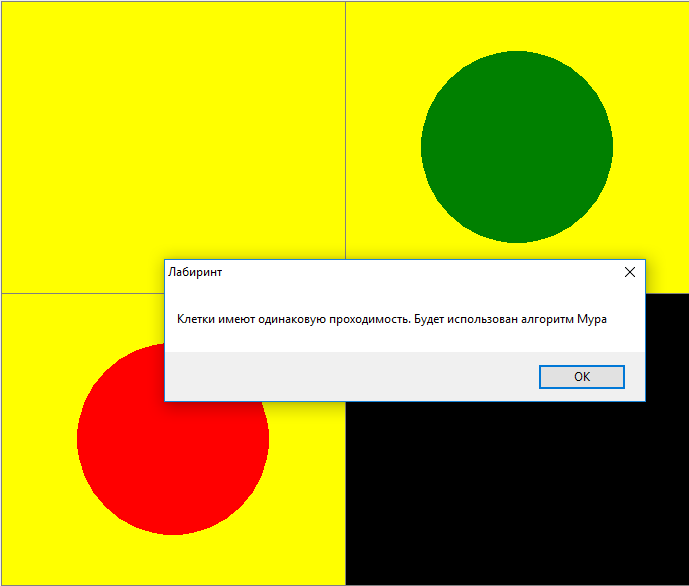


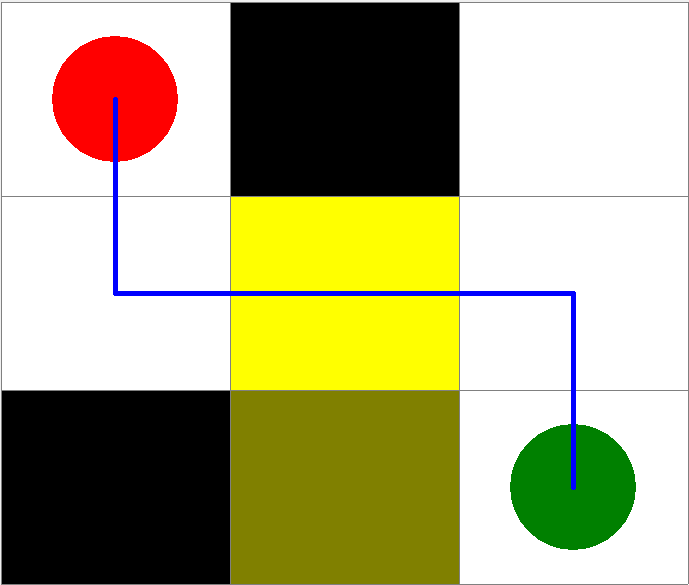
Если одно из полей по ошибке будет заполнено нулём, также появится следующее предупреждение:

1. Кнопка «Найти путь»

При нажатии на данную кнопку осуществится ряд действий:

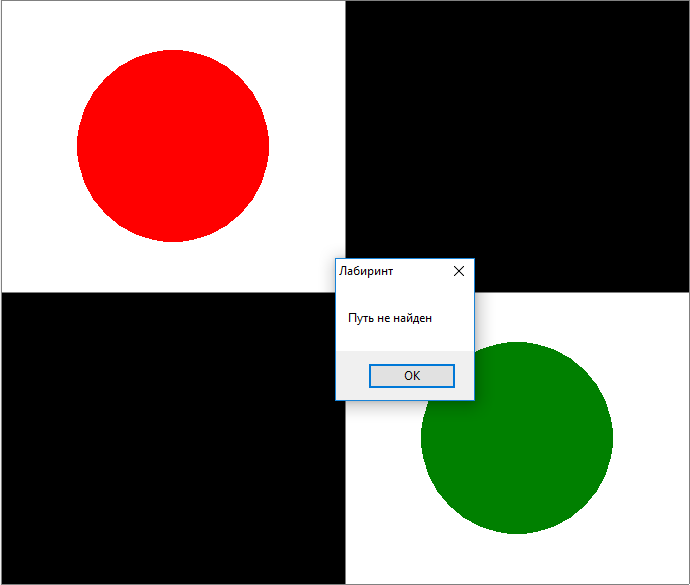
* 1. Осуществляется проверка на наличие на поле начальных и конечных клеток. При отсутствии хотя бы одной из них появляется сообщение:
  2. После прохождении пункта 3.1 вызывается метод bool isUseDijkstra(Field\* field), который определяет, нужно ли использовать алгоритм Дейкстры на взвешенном графе, или достаточно алгоритма Мура. Заметим, что все клетки необязательно должны быть все белыми для вызова алгоритма Мура. Они должны быть одного цвета.



* 1. После определения алгоритма данный алгоритм будет запущен.

В случае нахождения пути будет вызван метод void Field::DrawPath(TImage\* image, Points \*point), который нарисует искомый путь:

При следующем нажатии пользователя на поле данный путь будет стёрт, предоставляя возможность изменить положение начальной/конечной точек, а также сам лабиринт.



В случае, когда пути из начальной точки в конечную не существует, высветится соответствующее сообщение:

1. Кнопка «Выход».

По нажатию на данную кнопку происходит освобождение динамически выделенной памяти под поле field, а также под точки, содержащие его. После этого происходит выход из программы.

# **Заключение**

При выполнении курсовой работы были выполнены следующие задачи:

- Выполнен анализ предметной области, рассмотрен математический объект «граф» и описаны основные определения, связанные с ним.

- Были исследованы алгоритмы поиска кратчайшего пути Мура и Дейкстры, описана их работа, а также приведён наглядный пример их работы.

- Выполнен анализ аналогов данной программы и описана причина, по которой было принято решение о написании данного приложения.

- Были исследованы основные принципы работы со средой С++ Builder XE, изучена работа с основными его компонентами.

- Было разработано игра-приложение для наглядного демонстрирования работы данных алгоритмов. Помимо реализаций, добавлена возможность собственного конструирования лабиринтов любого вида. Потенциал данной программы не ограничивается данными функциями. Например, есть возможность добавления движения не только по клеткам, соседним по стороне, но и по вершине (8 соседей вместо 4). Также существует возможность добавления строгого направления движения от начальной до конечной точки. (В данной программе можно двигаться во все стороны, но можно, например, рассмотреть случай, когда можно двигаться только вверх, вправо и влево). Возможно добавление функций, способных анализировать изображение и строить лабиринты на основе полученных данных. Существует вариант добавления в данную программу так называемой «промежуточной точки», что позволит найти применение данной программы и в логистике (в грузоперевозках, например).

Таким образом, было создано полноценное приложение для построения и работы с лабиринтами, актуальность которого очевидна: от поиска кратчайшего маршрута между домом и магазином до планирования маршрутов грузоперевозок и поиска оптимальных путей на пересечённой местности.

# **Список литературы:**

1. Алгоритм Ли. [Электронный ресурс] – Режим доступа: <https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9B%D0%B8>. Дата доступа: 20.04.2017
2. Граф (математика). [Электронный ресурс] – Режим доступа: <https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)>. Дата доступа: 20.04.2017
3. Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры. [Электронный ресурс] – Режим доступа: <http://e-maxx.ru/algo/dijkstra>. Дата доступа: 20.04.2017
4. Жвакина А.В. Разработка Windows – приложений. /Пособие – Минск: Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники», 2017.

# **Приложение**

Код программы:

***Модуль “main.h”***

#include <vcl.h>

#pragma hdrstop

#include "main.h"

//---------------------------------------------------------------------------

#pragma package(smart\_init)

#pragma resource "\*.dfm"

TForm1 \*Form1;

Field \*field;

//---------------------------------------------------------------------------

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

: TForm(Owner)

{

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Image1MouseDown(TObject \*Sender, TMouseButton Button,

TShiftState Shift, int X, int Y)

{

int width = (Image1->Width - 1) / field->verticalNumber;

int height = (Image1->Height - 1) / field->horizontalNumber;

int xCount = Y/height;

int yCount = X/width;

if(Image1->Canvas->Pixels[X][Y] == clGray || Y >= field->horizontalNumber \* height

|| X >= field->verticalNumber \* width)

return;

if(field->havePath) {

switch(RadioGroup1->ItemIndex) {

case 0: field->ChangeFieldColor(Form1->Image1, &field->points[xCount][yCount]); break;

case 1: field->SetBeginPoint(Form1->Image1, xCount, yCount); break;

case 2: field->SetEndPoint(Form1->Image1, xCount, yCount); break;

}

}

else {

field->PaintField(Form1->Image1);

field->havePath = true;

}

}

void \_\_fastcall TForm1::FormActivate(TObject \*Sender)

{

field = new Field(2, 2);

Image1->Canvas->Brush->Color = clWhite;

Image1->Canvas->FillRect(Rect(0, 0, Image1->Width, Image1->Height));

field->DrawField(Form1->Image1);

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Button2Click(TObject \*Sender)

{

delete field;

Application->Terminate();

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Button3Click(TObject \*Sender)

{

Points \*end = field->end, \*begin = field->begin;

if(field->begin->posX == -1 || field->end->posX == -2) {

Application->MessageBox(String("Установите начальную и конечную точки").c\_str(),

String("Лабиринт").c\_str(), NULL);

return;

}

bool isUse = isUseDijkstra(field);

bool answer = isUse ? Dijkstra(field) : Moore(field);

String message = isUse? "Клетки имеют разную проходимость. Будет использован алгоритм Дейкстры":

"Клетки имеют одинаковую проходимость. Будет использован алгоритм Мура";

Application->MessageBox(message.c\_str(), String("Лабиринт").c\_str(), NULL);

if(answer == false)

Application->MessageBox(String("Путь не найден").c\_str(),

String("Лабиринт").c\_str(), NULL);

else {

field->DrawPath(Form1->Image1, &field->points[end->posX][end->posY]);

field->havePath = false;

}

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

try {

int horizontalNumber = StrToInt(Edit1->Text);

int verticalNumber = StrToInt(Edit2->Text);

if( horizontalNumber < 1 || verticalNumber < 1) {

Application->MessageBox(String("Размеры поля введены неверно").c\_str(),

String("Лабиринт").c\_str(), NULL);

return;

}

if( horizontalNumber > 50 || verticalNumber > 50) {

Application->MessageBox(String("Слишком большой размер поля (>50). Введите заново").c\_str(),

String("Лабиринт").c\_str(), NULL);

return;

}

delete field;

field = new Field(horizontalNumber, verticalNumber);

field->DrawField(Form1->Image1);

}

catch(Exception &e) {

Application->MessageBox(String("Не введены поля размеров лабиринта").c\_str(),

String("Лабиринт").c\_str(), NULL);

}

}

***Модуль “Points.h”***

#pragma hdrstop

#include "Points.h"

int Points::GetDistance()

{

int distance;

switch(color) {

case clWhite:

distance = 1;

break;

case clYellow:

distance = 2;

break;

case clOlive:

distance = 3;

break;

}

return distance;

}

void CreatePoints(Points\*\* &point, int horizontalNumber, int verticalNumber)

{

point = new Points\* [horizontalNumber];

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

point[i] = new Points[verticalNumber];

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

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

point[i][j].isPassed = 0;

point[i][j].distance = -1;

point[i][j].posX = i;

point[i][j].posY = j;

point[i][j].color = clWhite;

}

}

}

void DeletePoints(Points\*\* &point, int horizontalNumber, int verticalNumber)

{

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

delete[] point[i];

delete[] point;

}

void DrawPoint(TImage\* image, int width, int height, Points\* point, TColor color)

{

if(!point)

return;

int xCenter = (2 \* point->posY + 1) \* width / 2;

int yCenter = (2 \* point->posX + 1) \* height / 2;

image->Canvas->Brush->Color = color;

image->Canvas->Pen->Color = color;

int r = width < height ? (width / 3 - 1) : (height / 3 - 1);

image->Canvas->Ellipse(xCenter-r, yCenter-r, xCenter+r, yCenter+r);

}

void ClearPoint(TImage\* image, int width, int height, TColor borderColor, TColor cellColor, Points\* point)

{

int xCenter = (2 \* point->posY + 1) \* width / 2;

int yCenter = (2 \* point->posX + 1) \* height / 2;

image->Canvas->Brush->Color = cellColor;

image->Canvas->FloodFill(xCenter, yCenter, borderColor, fsBorder);

}

void ChangePointColor(Points\* point)

{

switch(point->color) {

case clBlack:

point->color = clWhite;

break;

case clWhite:

point->color = clYellow;

break;

case clYellow:

point->color = clOlive;

break;

case clOlive:

point->color = clBlack;

break;

}

}

//---------------------------------------------------------------------------

#pragma package(smart\_init)

***Модуль “Field.h”***

#pragma hdrstop

#include "Field.h"

Field::Field(int \_horizontalNumber, int \_verticalNumber)

{

horizontalNumber = \_horizontalNumber;

verticalNumber = \_verticalNumber;

CreatePoints(points, horizontalNumber, verticalNumber);

begin = new Points();

begin->posY = begin->posX = -1;

begin->color = clGreen;

end = new Points();

end->posY = end->posX = -2;

end->color = clRed;

havePath = true;

}

Field::~Field()

{

DeletePoints(points, horizontalNumber, verticalNumber);

delete begin;

delete end;

}

void Field::Initialize(int distance)

{

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

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

if(points[i][j].color == clBlack)

points[i][j].isPassed = 1;

else

points[i][j].isPassed = 0;

points[i][j].distance = distance;

}

}

}

void Field::DrawField(TImage\* image)

{

image->Canvas->Pen->Width = 1;

image->Canvas->Brush->Color = clWhite;

image->Canvas->FillRect(Rect(0, 0, image->Width, image->Height));

image->Canvas->Pen->Color = clGray;

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

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

image->Canvas->MoveTo(0, i \* height);

image->Canvas->LineTo(width \* verticalNumber, i \* height);

}

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

image->Canvas->MoveTo(i \* width, 0);

image->Canvas->LineTo(i \* width, height \* horizontalNumber);

}

}

void Field::PaintField(TImage\* image)

{

DrawField(image);

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

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

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

int X = (2 \* points[i][j].posY + 1) \* width / 2;

int Y = (2 \* points[i][j].posX + 1) \* height / 2;

image->Canvas->Brush->Color = points[i][j].color;

image->Canvas->FloodFill(X, Y, clGray, fsBorder);

}

}

DrawPoint(image, width, height, end, end->color);

DrawPoint(image, width, height, begin, begin->color);

}

//---------------------------------------------------------------------------

void Field::DrawPath(TImage\* image, Points \*point)

{

if(point->distance == 0)

return;

int xCur = point->posX;

int yCur = point->posY;

if(xCur != 0) {

if(points[xCur-1][yCur].distance == point->distance - point->GetDistance()) {

DrawLine(image, point, &points[xCur - 1][yCur], clBlue);

DrawPath(image, &points[xCur - 1][yCur]);

return;

}

}

if(xCur != horizontalNumber - 1) {

if(points[xCur + 1][yCur].distance == point->distance - point->GetDistance()) {

DrawLine(image, point, &points[xCur + 1][yCur], clBlue);

DrawPath(image, &points[xCur + 1][yCur]);

return;

}

}

if(yCur != 0) {

if(points[xCur][yCur - 1].distance == point->distance - point->GetDistance()) {

DrawLine(image, point, &points[xCur][yCur - 1], clBlue);

DrawPath(image, &points[xCur][yCur - 1]);

return;

}

}

if(yCur != verticalNumber - 1) {

if(points[xCur][yCur + 1].distance == point->distance - point->GetDistance()) {

DrawLine(image, point, &points[xCur][yCur + 1], clBlue);

DrawPath(image, &points[xCur][yCur + 1]);

return;

}

}

}

void Field::DrawLine(TImage \*image, Points \*start, Points \*finish, TColor color)

{

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

image->Canvas->Pen->Color = color;

image->Canvas->Pen->Width = 5;

int moveToX = (2 \* start->posY + 1) \* width / 2;

int moveToY = (2 \* start->posX + 1) \* height / 2;

int lineToX = (2 \* finish->posY + 1) \* width / 2;

int lineToY = (2 \* finish->posX + 1) \* height / 2;

image->Canvas->MoveTo(moveToX, moveToY);

image->Canvas->LineTo(lineToX, lineToY);

}

void Field::ChangeFieldColor(TImage\* image, Points\* point)

{

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

int X = (2 \* point->posY + 1) \* width / 2;

int Y = (2 \* point->posX + 1) \* height / 2;

ChangePointColor( point );

if(end->posY == point->posY && end->posX == point->posX)

end->posY = end->posX = -2;

if(begin->posY == point->posY && begin->posX == point->posX)

begin->posY = begin->posX = -1;

image->Canvas->Brush->Color = point->color;

image->Canvas->FloodFill(X, Y, clGray, fsBorder);

}

void Field::SetBeginPoint(TImage \*image, int xCount, int yCount)

{

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

if((xCount!=end->posX || yCount!=end->posY) && points[xCount][yCount].color!=clBlack) {

if(begin->posY != -1 && begin->posX != -1)

ClearPoint(image, width, height, clGray, points[begin->posX][begin->posY].color, begin);

begin->posX = xCount;

begin->posY = yCount;

DrawPoint(image, width, height, begin, begin->color);

}

}

void Field::SetEndPoint(TImage \*image, int xCount, int yCount)

{

int width = (image->Width - 1) / verticalNumber;

int height = (image->Height - 1) / horizontalNumber;

if((xCount!=begin->posX || yCount!=begin->posY) && points[xCount][yCount].color!=clBlack) {

if(end->posY != -2 && end->posX != -2)

ClearPoint(image, width, height, clGray, points[end->posX][end->posY].color, end);

end->posX = xCount;

end->posY = yCount;

DrawPoint(image, width, height, end, end->color);

}

}

//---------------------------------------------------------------------------

#pragma package(smart\_init)

***Модуль “Algorithms.h”***

#pragma hdrstop

#include "Algorithms.h"

bool Dijkstra(Field \*field)

{

int xCur, yCur;

field->Initialize(1000000000);

Points \*\*points = field->points;

Points \*end = field->end;

Points \*begin = field->begin;

points[begin->posX][begin->posY].distance = 0;

while(!allIsPassed(field)) {

if(!FindMinDistance(field, xCur, yCur))

break;

if(xCur != 0)

if( points[xCur][yCur].distance + points[xCur - 1][yCur].GetDistance() <

points[xCur - 1][yCur].distance && points[xCur - 1][yCur].color != clBlack )

points[xCur - 1][yCur].distance = points[xCur][yCur].distance + points[xCur-1][yCur].GetDistance();

if(xCur != field->horizontalNumber - 1)

if(points[xCur][yCur].distance + points[xCur + 1][yCur].GetDistance() <

points[xCur + 1][yCur].distance && points[xCur + 1][yCur].color != clBlack)

points[xCur + 1][yCur].distance = points[xCur][yCur].distance + points[xCur + 1][yCur].GetDistance();

if(yCur != field->verticalNumber-1)

if(points[xCur][yCur].distance + points[xCur][yCur + 1].GetDistance() <

points[xCur][yCur + 1].distance && points[xCur][yCur + 1].color != clBlack)

points[xCur][yCur + 1].distance = points[xCur][yCur].distance + points[xCur][yCur + 1].GetDistance();

if(yCur != 0)

if(points[xCur][yCur].distance + points[xCur][yCur - 1].GetDistance() <

points[xCur][yCur - 1].distance && points[xCur][yCur - 1].color != clBlack)

points[xCur][yCur - 1].distance = points[xCur][yCur].distance + points[xCur][yCur - 1].GetDistance();

points[xCur][yCur].isPassed = 1;

}

if(points[end->posX][end->posY].distance == 1000000000)

return false;

else

return true;

}

bool Moore(Field \*field)

{

field->Initialize(-1);

Points \*\*points = field->points;

Points \*end = field->end;

Points \*begin = field->begin;

std::deque<Points> myDeque;

points[begin->posX][begin->posY].distance = 0;

points[begin->posX][begin->posY].isPassed = 1;

myDeque.push\_back(points[begin->posX][begin->posY]);

while(!myDeque.empty()) {

Points cur = myDeque.front();

myDeque.pop\_front();

int xCur = cur.posX;

int yCur = cur.posY;

if(xCur != 0) {

if(points[xCur - 1][yCur].color != clBlack && points[xCur - 1][yCur].isPassed != 1) {

points[xCur - 1][yCur].isPassed = 1;

points[xCur - 1][yCur].distance = cur.distance + 1;

myDeque.push\_back(points[xCur - 1][yCur]);

}

}

if(xCur != field->horizontalNumber - 1) {

if(points[xCur + 1][yCur].color != clBlack && points[xCur + 1][yCur].isPassed != 1) {

points[xCur + 1][yCur].isPassed = 1;

points[xCur + 1][yCur].distance = cur.distance + 1;

myDeque.push\_back(points[xCur + 1][yCur]);

}

}

if(yCur != field->verticalNumber - 1) {

if(points[xCur][yCur + 1].color != clBlack && points[xCur][yCur + 1].isPassed != 1) {

points[xCur][yCur + 1].isPassed = 1;

points[xCur][yCur + 1].distance = cur.distance + 1;

myDeque.push\_back(points[xCur][yCur + 1]);

}

}

if(yCur != 0) {

if(points[xCur][yCur-1].color != clBlack && points[xCur][yCur - 1].isPassed != 1) {

points[xCur][yCur - 1].isPassed = 1;

points[xCur][yCur - 1].distance = cur.distance + 1;

myDeque.push\_back(points[xCur][yCur - 1]);

}

}

}

return points[end->posX][end->posY].isPassed != 0;

}

bool FindMinDistance(Field \*field, int& x, int& y)

{

x=0; y=0;

int dist = 999999999;

for(int i = 0; i < field->horizontalNumber; i++) {

for(int j = 0; j < field->verticalNumber; j++) {

if(field->points[i][j].distance < dist && field->points[i][j].isPassed == 0) {

dist = field->points[i][j].distance;

x = i;

y = j;

}

}

}

return dist != 999999999;

}

bool allIsPassed(Field \*field)

{

for(int i = 0; i < field->horizontalNumber; i++)

for(int j = 0; j < field->verticalNumber; j++)

if(field->points[i][j].isPassed == 0)

return false;

return true;

}

bool isUseDijkstra(Field \*field)

{

bool flag = true;

TColor color;

for(int i = 0; i < field->horizontalNumber && flag; i++)

for(int j = 0; j < field->verticalNumber && flag; j++)

if(field->points[i][j].color != clBlack) {

color = field->points[i][j].color;

flag = false;

}

for(int i = 0; i < field->horizontalNumber; i++)

for(int j = 0; j < field->verticalNumber; j++)

if(field->points[i][j].color != clBlack && field->points[i][j].color != color)

return true;

return false;

}

//---------------------------------------------------------------------------

#pragma package(smart\_init)