ОЛИМПИАДА ШКОЛЬНИКОВ «ШАГ В БУДУЩЕЕ»

НАУЧНО-ОБРАЗОВАТЕЛЬНОЕ СОРЕВНОВАНИЕ «ШАГ В БУДУЩЕЕ, МОСКВА»

ШМ0237

регистрационный номер

Информатика и системы управления
название факультета

ИУ-5

название кафедры

Поиск пути в топографических условиях

название работы

Автор: **Титаев Денис Витальевич**
фамилия, имя, отчество

ГБОУ Школа №1363, 11 класс

наименование учебного заведения, класс

Научный руководитель:

фамилия, имя, отчество

_____

место работы

_____

звание, должность

_____

подпись научного руководителя

Москва - 2018

Содержание работы:

ЗАДАЧА

В современном мире существует множество интересных и важных задач, в которых нахождение оптимального маршрута или пути является необходимым условием. Поиск кратчайшего пути – востребованная задача в современном мире, которая используется повсеместно: поиск кратчайшего маршрута на картах рельефа местности, поиск кратчайшего расстояния между контактами на печатной плате или наиболее удачного маршрута для передвижения роботов, машин. Существуют различные подходы к решению данных задач. В рамках данного проекта была поставлена задача продемонстрировать и провести сравнение двух востребованных методов поиска кратчайшего пути, а также адаптация этих алгоритмов для решения задачи нахождения кратчайшего пути в топографических условиях (условия рельефа представляют собой перепады высот, разный тип местности и т.п. которые влияют на сложность прохождения через конкретную точку.)

Приложение VII (лист 1)

ТОПОГРАФИЧЕСКАЯ КАРТА 1

Приложение VII (лист 2)

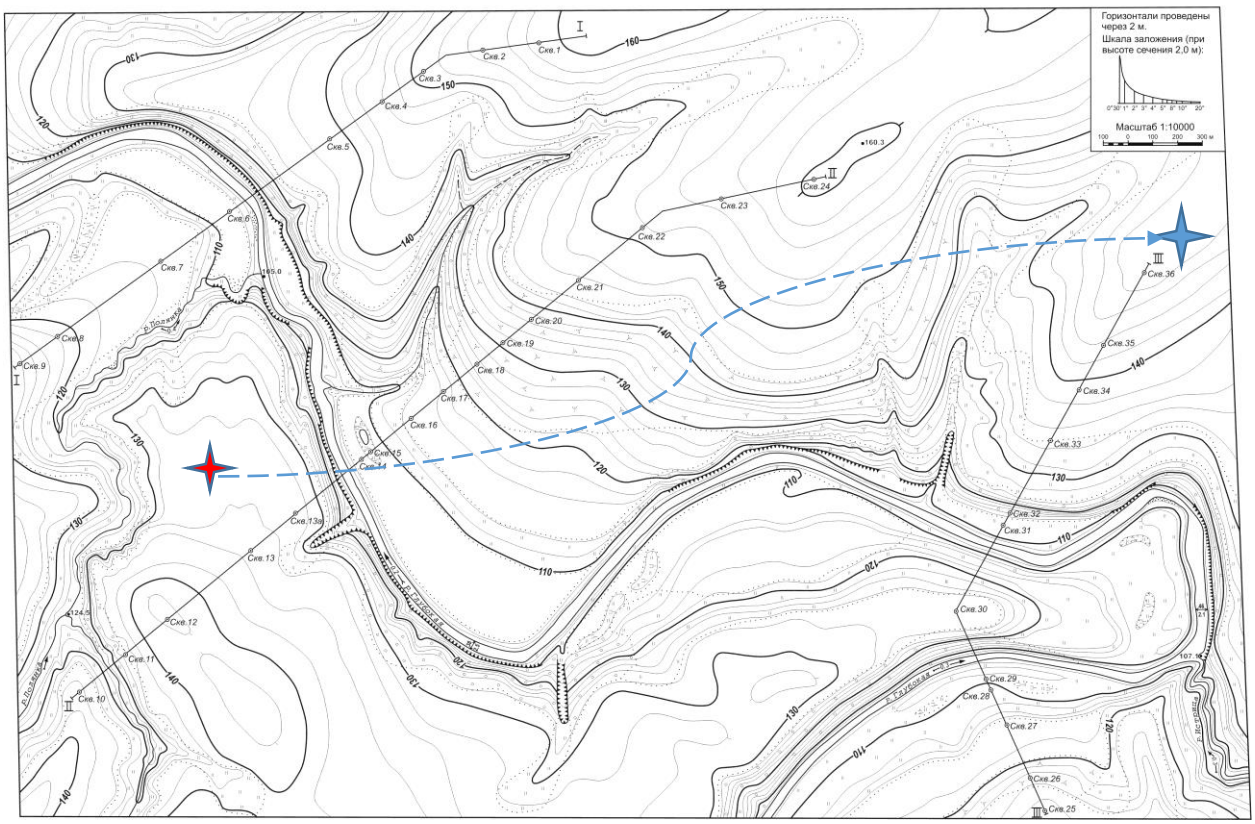


Рисунок 1.

ЦЕЛЬ РАБОТЫ

Цель работы – разработка и реализация наглядной программы, позволяющей строить наиболее оптимальный путь между выбранными пунктами, сравнивая множественные алгоритмы поиска пути, оценивая их скорость работы, а также методы ускорения работы этих алгоритмов.

Для достижения поставленной цели необходимо решить следующие задачи:

- Создать алгоритм поиска пути, оптимизировать его для необходимых для поставленной задачи потребностей.
- Исследовать алгоритмы поиска пути, а также изучить для них практическое применение.
- Сравнить эффективность алгоритмов поиска пути при различной сложности рельефа местности.
- Написать удобную и наглядную программу на языке программирования C++, использующую данный алгоритм и позволяющую описывать и сравнивать алгоритмы, скорость работы программы.
- Создать рекурсивную функцию, являющейся функцией поиска пути в двумерном массиве (Волновой алгоритм).
- Изучить и использовать графическую библиотеку OpenGL.

АКТУЛЬНОСТЬ

В условиях пересеченной местности, которая характеризуется наличием перепадов высот, всевозможных преград, непроходимых мест трудно определить оптимальный маршрут движения. Порой безошибочно оценить все сложности пути, сравнить различные маршруты, выбрать из них наиболее оптимальный не используя специальный алгоритм выбора пути, не представляется возможным. Наиболее известные алгоритмы делятся на две группы: Алгоритмы поиска в ширину и в глубину. Выбор алгоритма, который будет наиболее точно и быстро искать оптимальный путь (кратчайший, выгодный), учитывая все условия, значимая задача в различных сферах жизни.

Где применяются алгоритмы поиска пути?

- Многие люди пользуются навигаторами, электронными устройствами с печатными платами и микропроцессорами. Планируя поход или выбирая маршрут для прогулки люди используют алгоритм поиска пути, чтобы найти наиболее выгодный маршрут;
- При разработке устройств используются алгоритмы поиска кратчайшего расстояния между контактами на печатных платах;
- В мореходстве, а также в робототехнике используют алгоритмы поиска пути.

Анализируя эффективность использования различных алгоритмов поиска пути можно определить наиболее оптимальный по всем параметрам алгоритм для различных условий.

РЕАЛИЗАЦИЯ

Ход проекта

Осмыслить цели проекта;

Составлен план работы над проектом;

Изучить литературу по данной тематике;

Отобрать и проанализировать материал;

Продумать удобный графический интерфейс программы;

Написать программу на языке программирования C++;

Протестировать программу;

Сравнить алгоритмы по скорости;

Исправить ошибки;

Исключить “Излишние подсчёты” пути, а также найти путь за ещё меньшее время;

Представить результат.

Методика реализации

Любая программа построена на чётком алгоритме. Для задачи поиска оптимального пути необходимо рассмотреть несколько алгоритмов, чаще всего использующихся для этих целей. Мной был создан алгоритм поиска пути, просматривающий пути и заполняющий массив, а позже с помощью него находился кратчайший путь. Позже я узнал, что я на деле я написал алгоритм поиска в глубину.

Данные алгоритм определяют поиск пути в 8(или 4) направлениях, то есть наряду с основными путями (четыре направления) могу ещё использоваться диагональные пути. Использование движения по диагонали обусловлено потребностью двигаться действительно кратчайшим путём (как в реальности), даже если путь проходит не в узлах координат.

Для конкретной реализации алгоритма продуманна система оценки окружающих клеток. Формула для определения цены клетки:

$$C = Q*S,$$

где S – цена прохождения по прямой, Q – сложность прохождения конкретной клетки, C – цена пути до этой клетки относительно начала поиска.

S принимает значения:

- Если клетка вертикально или горизонтально, то S = 1 у.е.
- Если клетка расположена по диагонали, то S = sqrt(2) у.е.

Т.к. в реальности мы можем двигаться в любом направлении (360 градусов), то расстояние, пройденное по диагонали до следующей клетки будет равное не 1, как в вертикальном и горизонтальном направлении, а корень из двух. (Доказывается, использую теорему Пифагора)

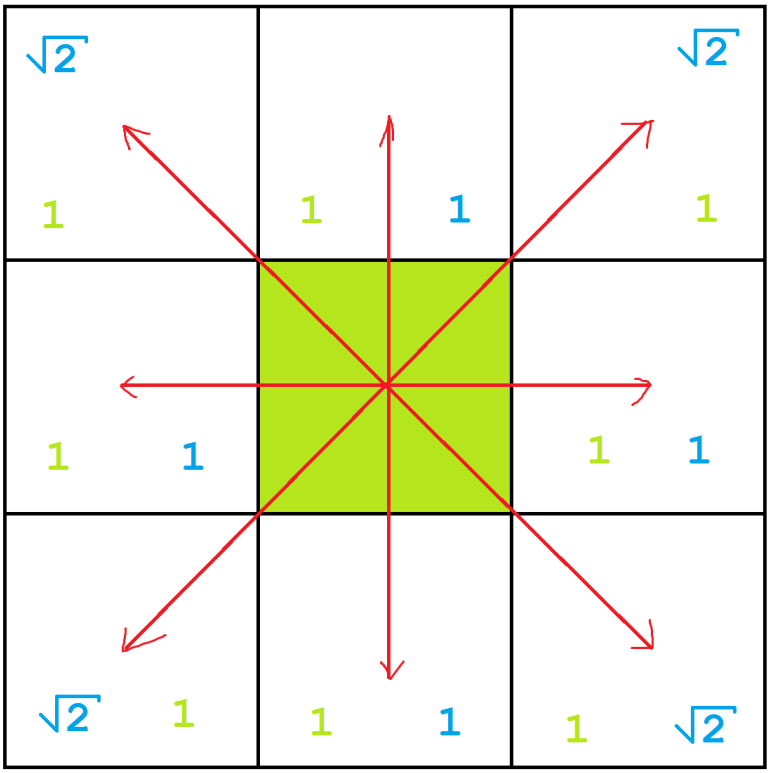


Рисунок 2.

На рисунке 2:

Красным цветом обозначены стрелки-направления, которые можно выбрать для движения.

Зелёная клетка – начало поиска пути, всего есть 8 направлений движения из неё (общий случай, если нет препятствий).

Синим цветом указана S цена прохождения по прямой из точки прародителя.

Зелёным цветом (цифра 1) – сложность прохождения конкретной клетки

Описание алгоритма поиска в глубину

У алгоритма поиска в глубину **(DFS)** существует два этапа.

Первый этап: Создание карты сложности проходимости, исходящей из начала пути.

Второй этап: Из конца пути, по карте из первого этапа, найти минимальный набор связанных клеток до начала (с минимальной ценой) и отобразить их.

Подробно о первом этапе. Определив начало поиска, рассмотреть окружающие. Для каждой окружающей точки определить С – сложность прохождения до этой точки, если этой цены нет в списке цен для этой точки или новая цена меньше старой, то цена заменяется на новую. Далее, из каждой из 8 (4) точек, окружающих начальную, проделать тоже самое, только в этот раз сложить С с клеткой – прародителем. Просматривать точку-прародитель новой точки не обязательно и не нужно. Заполнив карту с минимальными ценами пути от старта, можно переходить к следующему этапу

Подробно о втором этапе. Определив финиш пути, запомнить эту точку как конец пути и рассмотреть окружающие точки. Найти минимальный С из окружающих точек и записать его. Далее рассмотреть окружающие точки вокруг записанной точки и т.д. Как только цена С станет равной 0, это будет означает, что алгоритм нашёл начало пути. Соединяя записанные точки можно получить кратчайший и самый оптимальный путь.

(Наглядно показано в Приложении 1, указаны примеры и объяснено визуально)

Программная реализация алгоритма

```
void pathfinder(short x,short y,double temp){
if(temp > n_max || map[x][y]==-10)return;  //выход если глубоко или не пройти
spisok[x][y]=temp;                          //заполнение карты
if(x==finish_point_x && y==finish_point_y){n_max=temp;way_exist=1;}
    if(finish_point_x-x<0 && finish_point_y-y<0)//этот и 7 след. Определяют
//направление, куда будет “углубляться функция”
{napr[1]=0;napr[8]=1;napr[2]=2;napr[7]=3;napr[3]=4;
napr[6]=5;napr[4]=6;napr[5]=7;}
    if(finish_point_x-x==0 && finish_point_y-y<0)
{napr[2]=0;napr[1]=1;napr[3]=2;napr[8]=3;napr[4]=4;
napr[7]=5;napr[5]=6;napr[6]=7;}
    if(finish_point_x-x>0 && finish_point_y-y<0)
{napr[3]=0;napr[2]=1;napr[4]=2;napr[1]=3;napr[5]=4;
napr[8]=5;napr[6]=6;napr[7]=7;}
    if(finish_point_x-x>0 && finish_point_y-y==0)
{napr[4]=0;napr[3]=1;napr[5]=2;napr[2]=3;napr[6]=4;
napr[1]=5;napr[7]=6;napr[8]=7;}
    if(finish_point_x-x>0 && finish_point_y-y>0)
{napr[5]=0;napr[4]=1;napr[6]=2;napr[3]=3;napr[7]=4;
napr[2]=5;napr[8]=6;napr[1]=7;}
    if(finish_point_x-x==0 && finish_point_y-y>0)
```

```
{napr[6]=0;napr[5]=1;napr[7]=2;napr[4]=3;napr[8]=4;
napr[3]=5;napr[2]=6;napr[1]=7;}
        if(finish_point_x-x<0 && finish_point_y-y>0)
{napr[7]=0;napr[6]=1;napr[8]=2;napr[5]=3;napr[1]=4;
napr[4]=5;napr[2]=6;napr[3]=7;}
        if(finish_point_x-x<0 && finish_point_y-y==0)
{napr[8]=0;napr[7]=1;napr[1]=2;napr[6]=3;napr[2]=4;
napr[5]=5;napr[3]=6;napr[4]=7;}
for(int ioi=0;ioi<8;ioi++){//вызов в порядке приоритета, распространение
if(napr[1]-ioi==0)if(spisok[x-1][y-1]>temp && (x-1)>=0 && (y-1)>=0 &&
temp+map[x-1][y-1]*1.414213<spisok[x-1][y-1])pathfinder(x-1,y-1,temp+map[x-
1][y-1]*1.414213);//-1 -1 1.414.. это корень из двух, для движения диагонально
if(napr[2]-ioi==0)if(spisok[x][y-1]>temp && (y-1)>=0 && temp+map[x][y-
1]*1<spisok[x][y-1])pathfinder(x,y-1,temp+map[x][y-1]); //0 -1
if(napr[3]-ioi==0)if(spisok[x+1][y-1]>temp&& (x+1)<map_size && (y-1)>=0 &&
temp+map[x+1][y-1]*1.414213<spisok[x+1][y-1])pathfinder(x+1,y-
1,temp+map[x+1][y-1]*1.414213);//1 -1
if(napr[8]-ioi==0)if(spisok[x-1][y]>temp&& (x-1)>=0 && temp+map[x-
1][y]*1<spisok[x-1][y])pathfinder(x-1,y,temp+map[x-1][y]); //-1 0
if(napr[4]-ioi==0)if(spisok[x+1][y]>temp && (x+1)<map_size &&
temp+map[x+1][y]*1<spisok[x+1][y])pathfinder(x+1,y,temp+map[x+1][y]); //1 0
if(napr[7]-ioi==0)if(spisok[x-1][y+1]>temp && (x-1)>=0 && (y+1)<map_size
&& temp+map[x-1][y+1]*1.414213<spisok[x-1][y+1])pathfinder(x-
1,y+1,temp+map[x-1][y+1]*1.414213);//-1 1
```

```
if(napr[6]-ioi==0)if(spisok[x][y+1]>temp && (y+1)<map_size  &&
temp+map[x][y+1]*1<spisok[x][y+1])pathfinder(x,y+1,temp+map[x][y+1]); //0 1

if(napr[5]-ioi==0)if(spisok[x+1][y+1]>temp&& (x+1)<map_size &&
(y+1)<map_size&&temp+map[x+1][y+1]*1.414213<spisok[x+1][y+1])pathfinder
(x+1,y+1,temp+map[x+1][y+1]*1.414213);//+1 +1

}}
```

Данная функция – наглядный пример реализации первого этапа алгоритма поиска в глубину.

Двумерный массив spisok имеет ту же размерность что и карта и сохраняет для каждой точки цену C.

Функция принимает координаты точки, вокруг который нужно рассмотреть следующие, а также цену пути в данный момент (temp)

Окружающие точки рассчитываются так, что C следующий точки считается как C предыдущей плюс S.

Корень из двух для оптимизации был приближенно взят как 1.414213.

(Он используется для поиска S)

Также видно использования n_max, ограничивает глубину в рекурсии.

Действительно, т.к. поиск происходит во всех направлениях, то найдя финиш любым из путём, оптимальный будет меньше либо равен этому. (Поэтому на последней картинке заполнен не весь массив, а только некоторая часть).

Программная реализация второго этапа

```
void rev_path(){
int c_x=finish_point_x, c_y=finish_point_y,temp_x=0,temp_y=0;double max = 0;
true_path[0][0]=finish_point_x;true_path[0][1]=finish_point_y; // 0 - x, 1 - y
    for(int i = 1;i<1000;i++){ max=0;//поиск пути из финиша
if(spisok[c_x][c_y]-spisok[c_x-1][c_y-1]>max){max=abs(spisok[c_x-1][c_y-1]-spisok[c_x][c_y]);temp_x=c_x-1;temp_y=c_y-1;}
if(spisok[c_x][c_y]-spisok[c_x][c_y-1]>max){max=abs(spisok[c_x][c_y-1]-spisok[c_x][c_y]);temp_x=c_x;temp_y=c_y-1;}
if(spisok[c_x][c_y]-spisok[c_x+1][c_y-1]>max){max=abs(spisok[c_x+1][c_y-1]-spisok[c_x][c_y]);temp_x=c_x+1;temp_y=c_y-1;}
if(spisok[c_x][c_y]-spisok[c_x+1][c_y]>max){max=abs(spisok[c_x+1][c_y]-spisok[c_x][c_y]);temp_x=c_x+1;temp_y=c_y;}
if(spisok[c_x][c_y]-spisok[c_x+1][c_y+1]>max){max=abs(spisok[c_x+1][c_y+1]-spisok[c_x][c_y]);temp_x=c_x+1;temp_y=c_y+1;}
if(spisok[c_x][c_y]-spisok[c_x][c_y+1]>max){max=abs(spisok[c_x][c_y+1]-spisok[c_x][c_y]);temp_x=c_x;temp_y=c_y+1;}
if(spisok[c_x][c_y]-spisok[c_x-1][c_y+1]>max){max=abs(spisok[c_x-1][c_y+1]-spisok[c_x][c_y]);temp_x=c_x-1;temp_y=c_y+1;}
if(spisok[c_x][c_y]-spisok[c_x-1][c_y]>max){max=abs(spisok[c_x-1][c_y]-spisok[c_x][c_y]);temp_x=c_x-1;temp_y=c_y;}
true_path[i][0]=temp_x;true_path[i][1]=temp_y;c_x=temp_x;c_y=temp_y;
if(temp_x==start_point_x && temp_y==start_point_y){break;}}}
```

Здесь можно увидеть последовательный поиск пути, где используется двумерный массив true_path, у которого размерность [N][2], то есть список точек, с координатами x и y.
Задаётся конкретная точка (точка финиша), и от неё происходит выполнение алгоритма.
Происходит поиск наиболее подходящих точек вокруг конкретных c_x и c_y, далее новая точка становится началом поиска. Наиболее подходящая точка – с минимальным C, то есть значением в массиве spisok.
После выполнения алгоритма записывается в массив true_path готовый путь. Отобразив его программа выполнить свою главную цель.

Описание алгоритма поиска в ширину (Волновой алгоритм, BFS)

Как и у алгоритма поиска в глубину, у волнового алгоритма (алгоритма Ли) тоже существует два этапа.
Первый этап: Создание волновой карты, исходящей из начала пути.
Второй этап: Из конца пути, по волновой карте из первого этапа, найти минимальный набор связанных клеток до начала (с минимальной ценой) и отобразить их.

Подробно о первом этапе. В первом этапе определяется два списка, открытый и закрытый. Определив начало поиска, рассмотреть окружающие. Каждую окружающую точку добавить в открытый список (если местность проходима) попутно определив C – сложность прохождения до этой точки.
Как только возможные точки вокруг нашей добавлены в список, эта точка добавляется в закрытый список. Далее, проделать тоже самое из следующей точки из открытого списка. Просматривать точку-прародитель алгоритм не будет, т.к. она в закрытом списке. Заполнив карту с минимальными ценами пути от старта, можно переходить к следующему этапу

Второй этап очень схож со вторым этапом поиска в глубину. Определив финиш пути, запомнить эту точку как конец пути и рассмотреть окружающие точки. Найти минимальный С из окружающих точек и записать его. Далее рассмотреть окружающие точки вокруг записанной точки и т.д. Как только цена С станет равной 0, это означает что алгоритм нашёл начало пути. Соединяя записанные точки можно получить кратчайший и самый оптимальный путь

(Наглядно показано в Приложении 2, указаны примеры и приведены объяснения)

Программная реализация алгоритма

```
void BFS(){//для четырёх направлений
int x,y,k=1;
for(int i = 0;i<9000;i++){
    x=spisok_open[i][0];y=spisok_open[i][1];//след. в открытом списке
    if(x==finish_point_x && y==finish_point_y){way_exist=1;break;}//конец
    if(spisok_closed[x][y-1]!=1   && y-1>=0 && spisok[x][y-
1]==0){spisok_open[k][0]=x;spisok_open[k][1]=y-1;k++;spisok[x][y-
1]=spisok[x][y]+map[x][y-1];}//в этом и следующих ведётся поиск
//подходяших клеток для открытого списка
    if(spisok_closed[x+1][y]!=1 && x+1<map_size&&
spisok[x+1][y]==0){spisok_open[k][0]=x+1;spisok_open[k][1]=y;k++;spisok[x+1
][y]=spisok[x][y]+map[x+1][y];}
    if(spisok_closed[x][y+1]!=1 && y+1<map_size &&
spisok[x][y+1]==0){spisok_open[k][0]=x;spisok_open[k][1]=y+1;k++;spisok[x][y
+1]=spisok[x][y]+map[x][y+1];}
```

```
if(spisok_closed[x-1][y]!=1 && x-1>=0 && spisok[x-1][y]==0
){spisok_open[k][0]=x-1;spisok_open[k][1]=y;k++;spisok[x-
1][y]=spisok[x][y]+map[x-1][y];}

spisok_closed[x][y]=1;}//текущая точка попадает в закрытый список

if(spisok[finish_point_x][finish_point_y]!=0)way_exist=1;}
```

Данная функция – функций первого этапа алгоритма поиска в ширину (волнового алгоритма).

Двумерный массив spisok имеет ту же размерность что и карта и сохраняет для каждой точки цену C.

Алгоритм заполняет spisok_open, а после идёт по точкам в нём, пройденные точки добавляются в закрытый список.

Окружающие точки рассчитываются так, что C следующий точки считается как C предыдущей плюс S.

Действительно, так как волна распространяет во всех направлениях, то найдя финиш любым из путём, оптимальный будет меньше либо равен этому. (Поэтому на последней картинке (приложение 2, волновой алгоритм) заполнен не весь массив, а только нужная часть).

Второй этап подобен второму этапу из алгоритма поиска пути в глубину, поиск от финиша к началу.

Сравнение алгоритмов

Сравнение по применимости:

Алгоритм поиска в глубину можно применять как на картах с одной проходимостью, так и на картах с разной.

Волновой алгоритм применяется только для карт с одинаковой проходимостью (непроходимые препятствия могут быть).

Сравнение по времени работы:

Волновой алгоритм работает в несколько раз быстрее, чем алгоритм в глубину, вследствие того, что точки, который уже прошёл алгоритм не приходится больше проходить. (В Приложении 2 есть сравнение этих алгоритмов)

Сравнение по памяти:

По затратам памяти алгоритмы примерно равны.

Исходя из этих особенностей можно выбрать, для каких случаев, какой алгоритм лучше:

Если на карте плоская местность и отдельные непроходимые участки, то лучше использовать волновой алгоритм поиска.

Если проходимость не «равномерна» (рельефная местность, множественные преграды разных типов и т.п.) на карте, то нужно использовать «поиск в глубину».

Заключение

В ходе данной исследовательской работы были рассмотрены и проанализированы разные алгоритмы поиска пути, связанные с задачей поиска как можно менее затратного по времени пути. Выявлены условия и особенности, при которых используется определенные типы алгоритмов поиска пути. В программе наглядно продемонстрированы результаты работы алгоритмов поиска пути.

Разработанная программа позволяет строить оптимальные маршруты в различных топографических условиях (сложность которых можно задавать), с демонстрацией временных затрат.

Возможности применения данной программы:

1. Ее можно использовать для наглядной демонстрации работы выбранного алгоритма

2. Ее можно использовать для сравнения выбранного алгоритма с другим алгоритмом поиска пути, находя наилучший для представленных условий, определяя проблемные места.
3. Ее можно использовать в расчетах времени пути в реальных условиях.

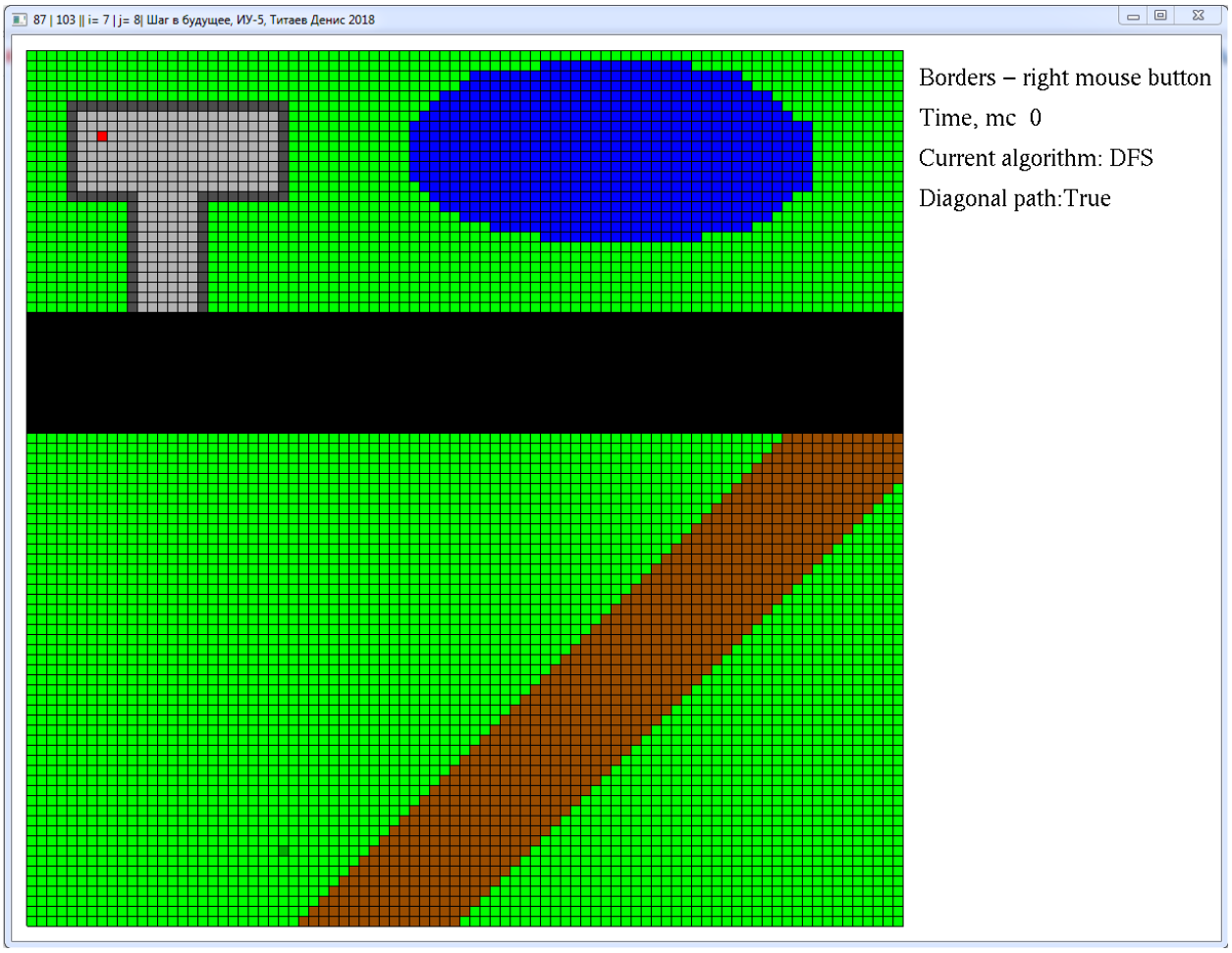
Список литературы.

Васильев А.Н. “Самоучитель C++ с примерами и задачами 4 издание”, издательство “Нит”, 2016

М.Ву, Т.Девис, Дж.Нейдер, Д.Шрайнер “OpenGL Руководство по программированию 4 издание”, издательство “Питер”

Приложение №1 к работе

Пример работы алгоритма поиска в глубину



Условные обозначения:

Тёмно-зелёным обозначена точка начала поиска пути.

Красным обозначена точка конца пути.

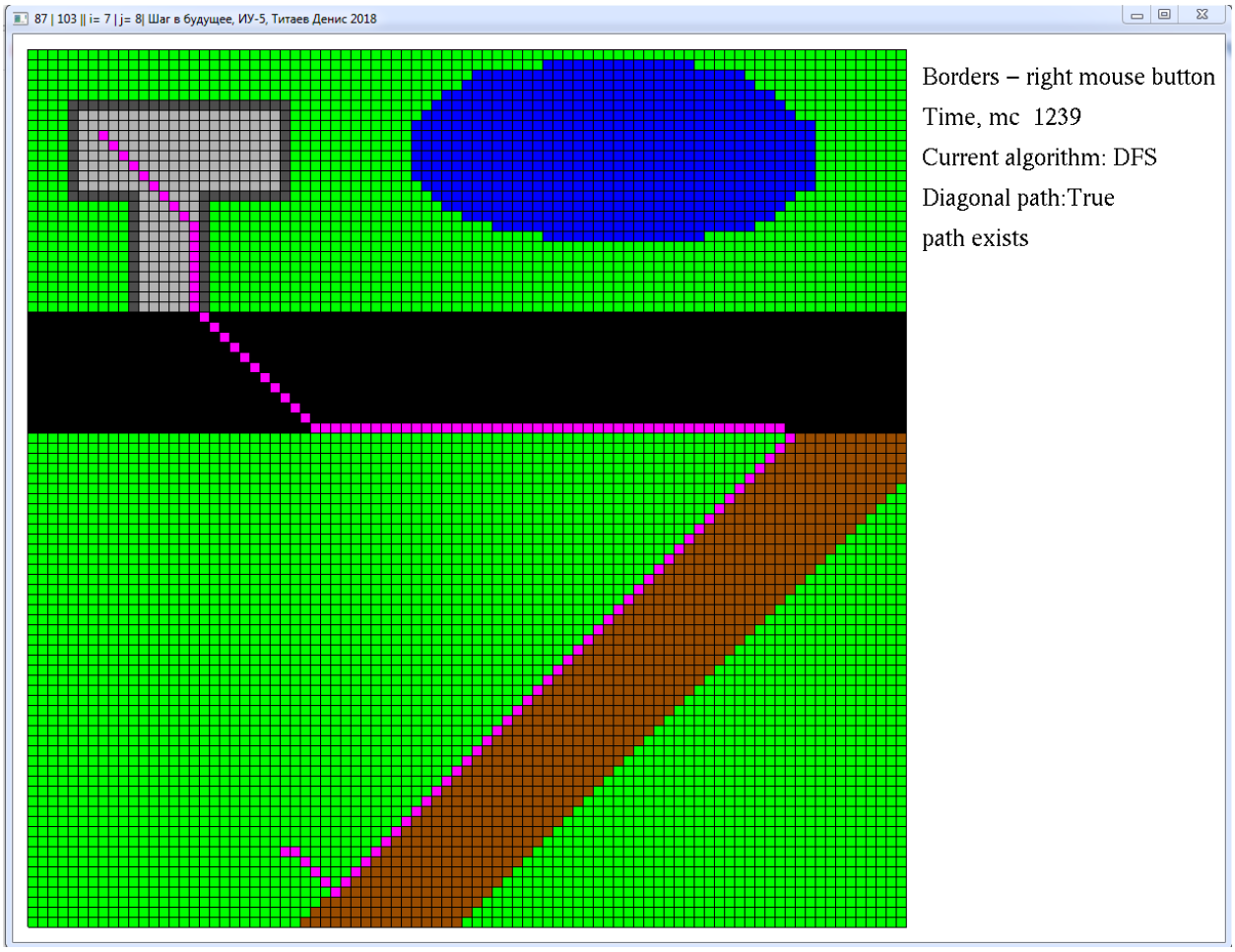
В данном примере чёрный цвет-цвет шоссе, наибольшая скорость прохождения

Серым и тёмно-серым цветом обозначено поле, скорость его прохождения меньше шоссе

Коричневым – просёлочная дорога, ещё медленнее чем у серого поля.

Зелёный цвет – цвет леса, пересечённой местности, скорость прохождения низкая. (Синий соответственно озеро, самая низкая скорость).

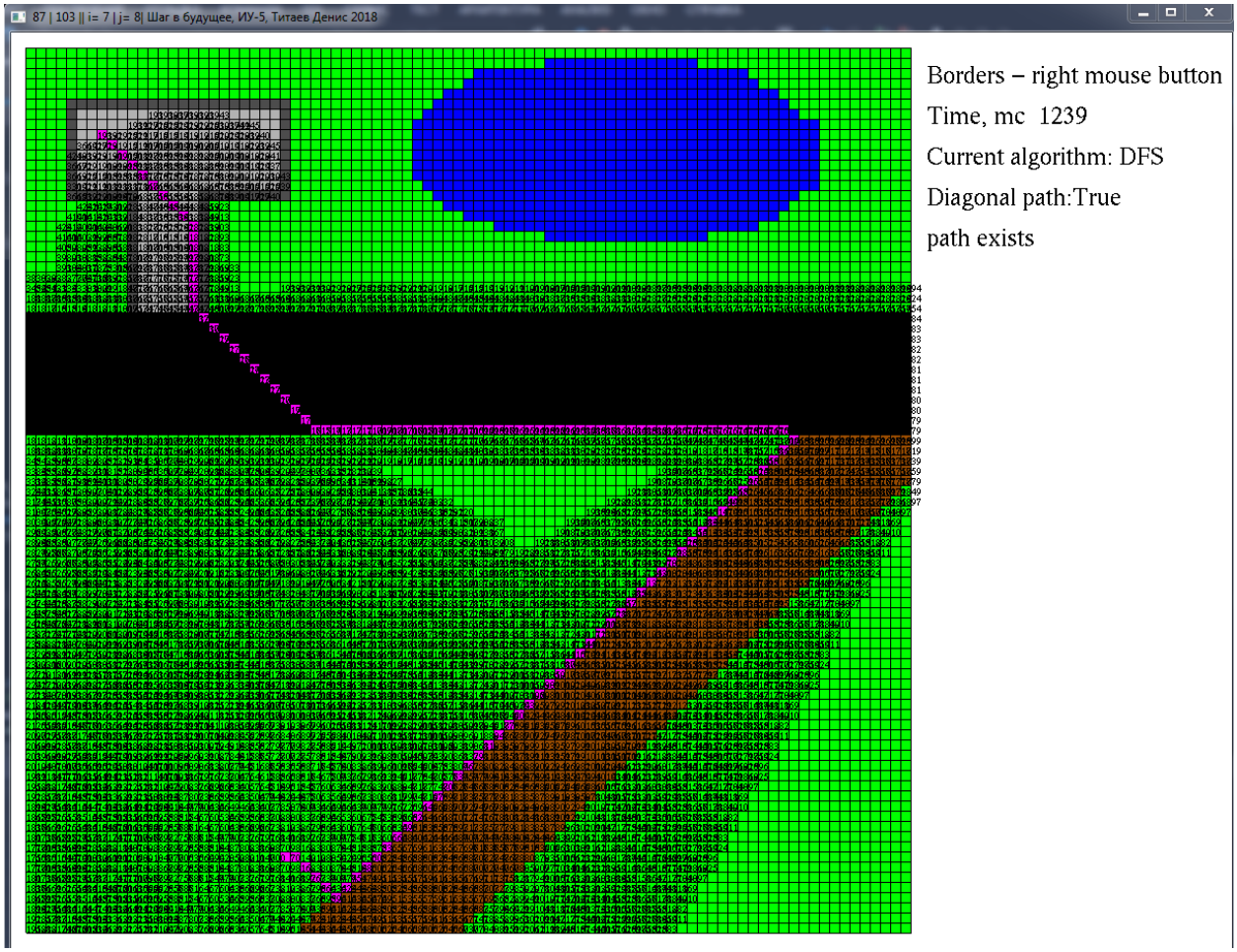
В программе есть учёт времени работы алгоритма (прим. Обоих этапов), время в миллисекундах.



Здесь уже показан результат работы программы.

Фиолетовым цветом показан кратчайший путь.

Важно заметить, что идти по диагонали дороже чем по вертикале или горизонтали, но всё же дешевле чем прохождение двух клеток. Это наглядно показывает смысл движения по диагонали.



Здесь можно увидеть просмотренные пути, исходящие из начала пути до конечной точки.

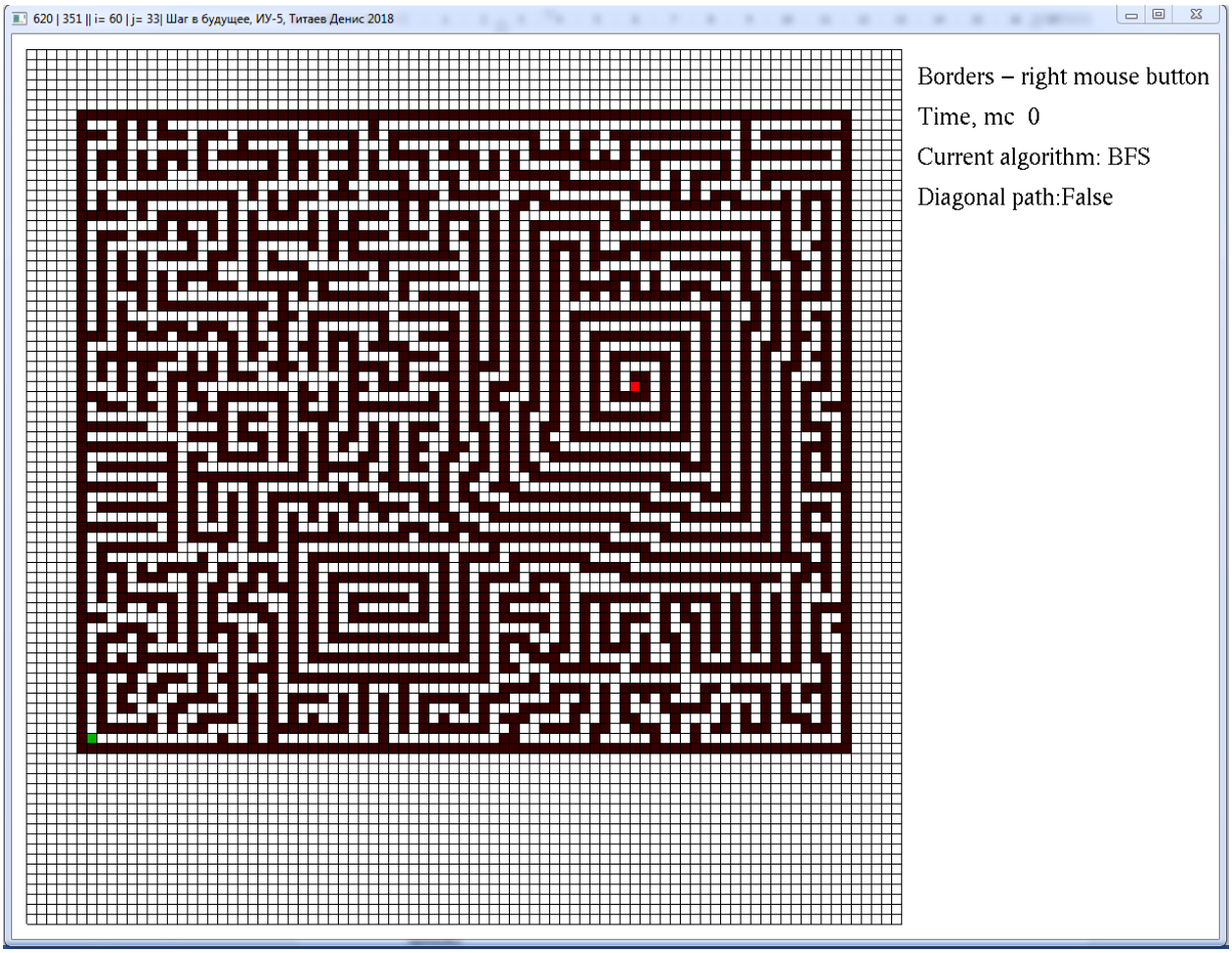
Найденный путь кратчайший, можно увидеть в программе цену до него, это 193.

В каждой клетке указана цена C, режим отображения их в программе – F1.

Видно, что программа не рассматривала все варианты, то есть остались поля, который программа не рассмотрела. Это показывает её эффективность.

Приложение №2 к работе

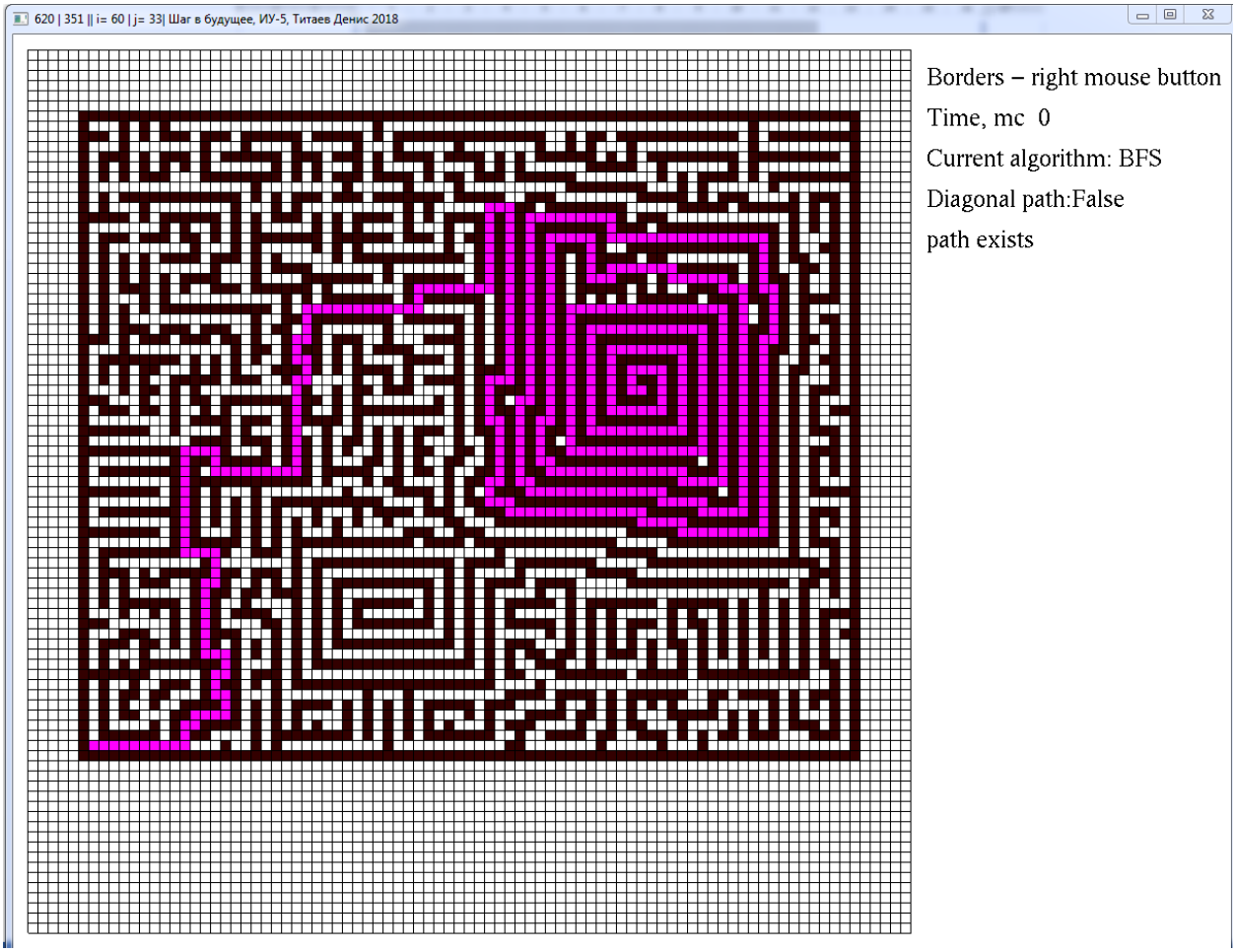
Пример работы волнового алгоритма



Тёмно-зелёным обозначена точка начала поиска пути.

Красным обозначена точка конца пути.

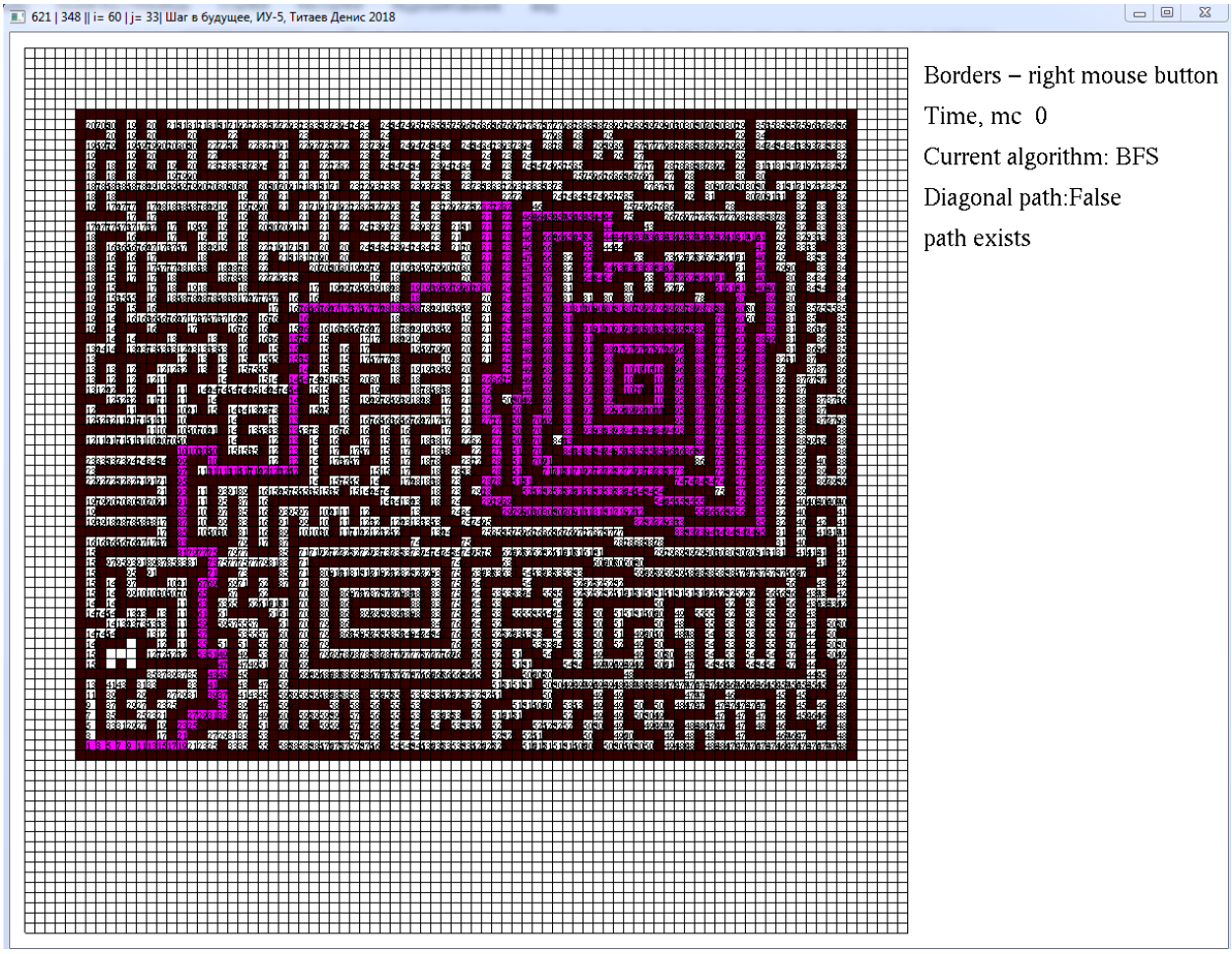
В данном примере чёрный цвет непроходимая поле (равносильно сложность прохождения стремится к бесконечностью, по сравнению с белым полем)



Здесь уже показан результат работы программы.

Программа определила, что путь существует и нашла кратчайший кратчайший.

Фиолетовым цветом показан кратчайший путь.

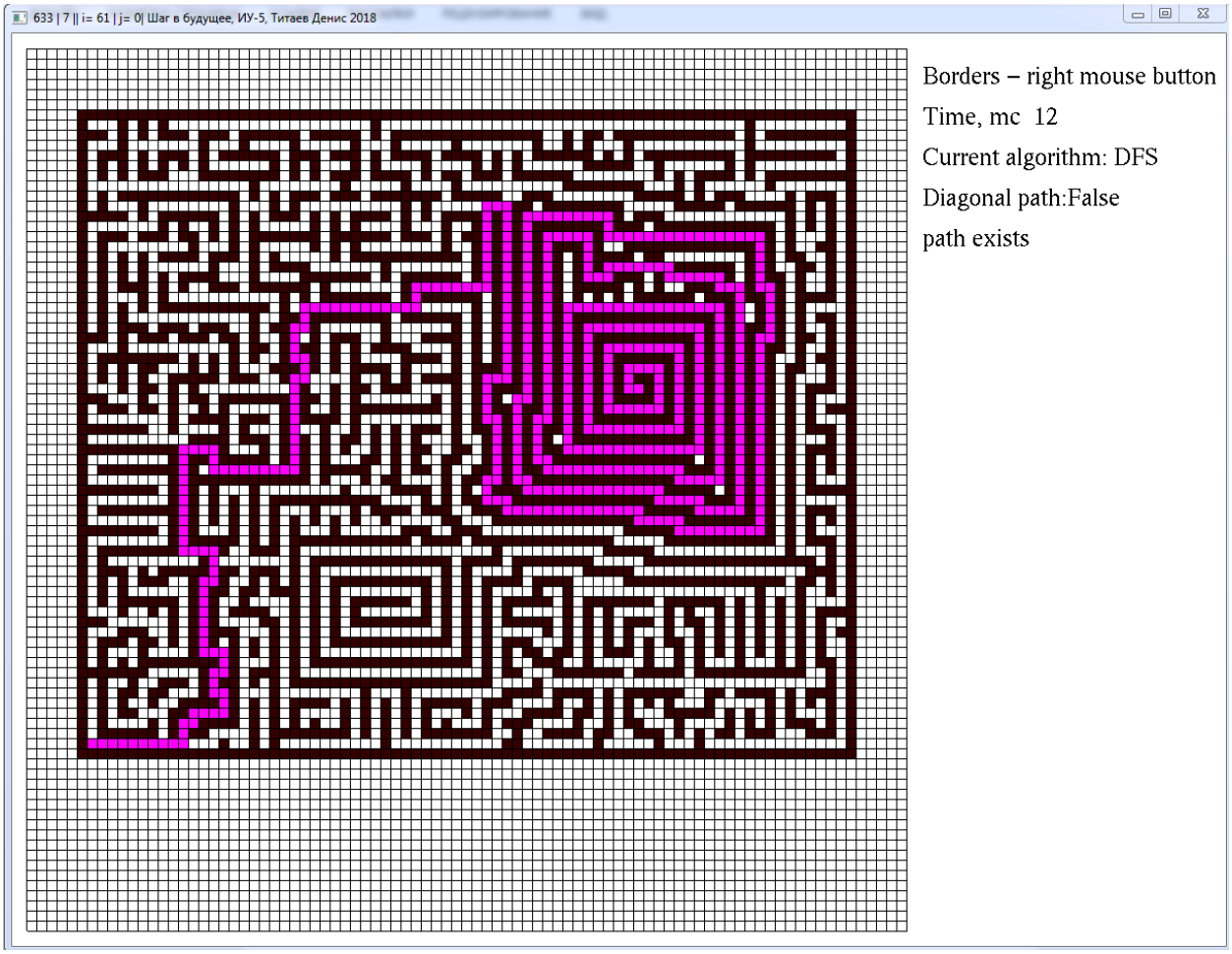


Цифры указывают дальность кратчайшего пути от начала.

Видно, что непроходимый вариант не рассмотрен, в лабиринте рассмотрены все проходимые пути, т.к. начало и финиш находятся в большом расстоянии.

Время – меньше миллисекунды (крайне малое количество операций)

Для сравнение запустим эти же точки, но теперь используя алгоритм поиска в глубину:



Видно, что путь найден кратчайший, на теперь время работы – 12 миллисекунд. Это объясняется тем, что поиск в глубину ещё и оценивает цену каждой клетки и стремится вглубь. Также, поиск в глубину реализован рекурсивно, а поиск в ширину с использованием цикла.

Приложение №3 (системные требования, управление)

Системные требования программы:

Наличие видеокарты и видеодрайвера, поддерживающего OpenGL 1.4 (Лучше 2.1 и новее версии)

Операционная система: Windows 7 и выше

Также должны быть установлены распространяемы пакеты MSVC (как минимум от visual studio 2012, а также framework 4.0)

Наличие библиотеки freeglut.dll в директории вместе с программой.

Управление в программе:

F1-отобразить массив с ценой пути до узла, считая от начала (отобразить spisok)

F2-отобразить массив со сложностью каждого узла отдельно

F3-убрать все пометки

END- очистить, можно заново точки начала и конца ставить

F6 –записать карту вместо заготовленной карты 3(map3.txt с приложением)

F7 – разрешить /запретить поиск пути по диагонали

F8 – BFS/DFS (выбрать алгоритм, в глубину или в ширину)

F9- считать заготовленную карту 1 (map.txt с приложением)

F10- считать заготовленную карту 2 (map2.txt с приложением)

F11- считать заготовленную карту 3 (map3.txt с приложением)

Левая кнопка мыши задаёт координаты начала и финиша

Правая кнопка мыши задаёт непроходимые точки (препятствия)

PAGE UP – начать поиск пути и вывести результаты