Санкт-Петербургский политехнический университет Петра великого

Институт машиностроения, материалов и транспорта

Курсовая работа

По дисциплине: «Объектно-ориентрованное программирование»

Выполнил:

студент гр. 3331506/90401 Малышев М.В.

Преподаватель Ананьевский М.С.

«___»_____2022 г.

Санкт-Петербург

2022 г

1. ВВЕДЕНИЕ

Алгоритм Ли (волновой алгоритм) — алгоритм поиска пути, алгоритм поиска кратчайшего пути на планарном графе. Принадлежит к алгоритмам, основанным на методах поиска в ширину.

В основном используется при компьютерной трассировке (разводке) печатных плат, соединительных проводников на поверхности микросхем. Другое применение волнового алгоритма — поиск кратчайшего расстояния на карте в компьютерных стратегических играх. Также волновой алгоритм является простейшим вариантом навигатора.

2. ИДЕЯ АЛГОРИТМА

2.1 Задача поиска кратчайшего пути

Алгоритм работает на *дискретном рабочем поле* (ДРП), представляющем собой ограниченную замкнутой линией фигуру, не обязательно прямоугольную, разбитую на прямоугольные ячейки, в частном случае — квадратные. Множество всех ячеек ДРП разбивается на подмножества: «проходимые» (свободные), т. е при поиске пути их можно проходить, «непроходимые» (препятствия), путь через эту ячейку запрещён, стартовая ячейка (источник) и финишная (приемник). Назначение стартовой и финишной ячеек условно, достаточно — указание пары ячеек, между которыми нужно найти кратчайший путь.

Алгоритм предназначен для поиска кратчайшего пути от стартовой ячейки к конечной ячейке, если это возможно, либо, при отсутствии пути, выдать сообщение о непроходимости

Пример ДРП с решенной задачей кратчайшего пути между двумя точками представлен на рисунке 2.1.

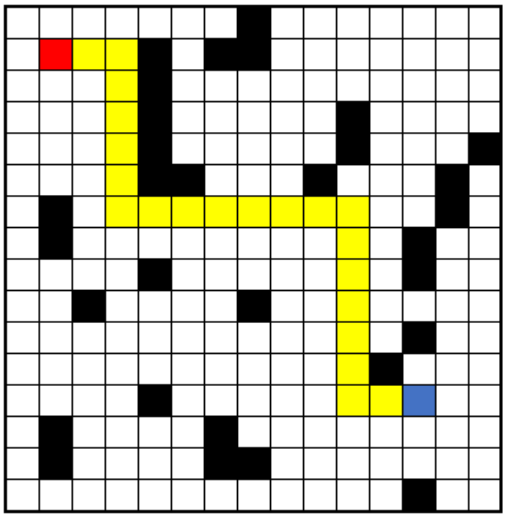

Рисунок 2.1 - Пример ДРП с решенной задачей кратчайшего пути между двумя точками.

Существует два вида алгоритма (пути): ортогональный (окрестность фон Неймана, соседние 4 ячейки) и орто-диагональный (окрестность Мура, 8 соседних ячеек). Примеры окрестностей представлены на рисунках 2.2 (а) и 2.2 (б) соответственно.

а б

Рисунок 2.2 – Окрестность фон Неймана (а) и окрестность Мура (б)

2.2 Алгоритм Ли

Алгоритм Ли состоит из трех этапов: инициализация, распространение волны, прокладывание маршрута.

Алгоритм Ли:

- Возвращает набор координат точек, входящих в найденный путь (если путь существует)
- Если пути не существует, алгоритм возвращает сообщение о том, что пути нет
- Если заданная начальная и/или конечная точка является стеной, алгоритм возвращает ошибку
- Если возможных путей несколько – алгоритм возвращает самый короткий

Разберем этапы на примере поля 15X15, соседними точками будем считать точки в окрестности фон Неймана

1 этап. Инициализация поля и задание стартовой и конечной точки.

Пример представлен на рисунке 2.3 (красная точка – старт, синяя – финиш).

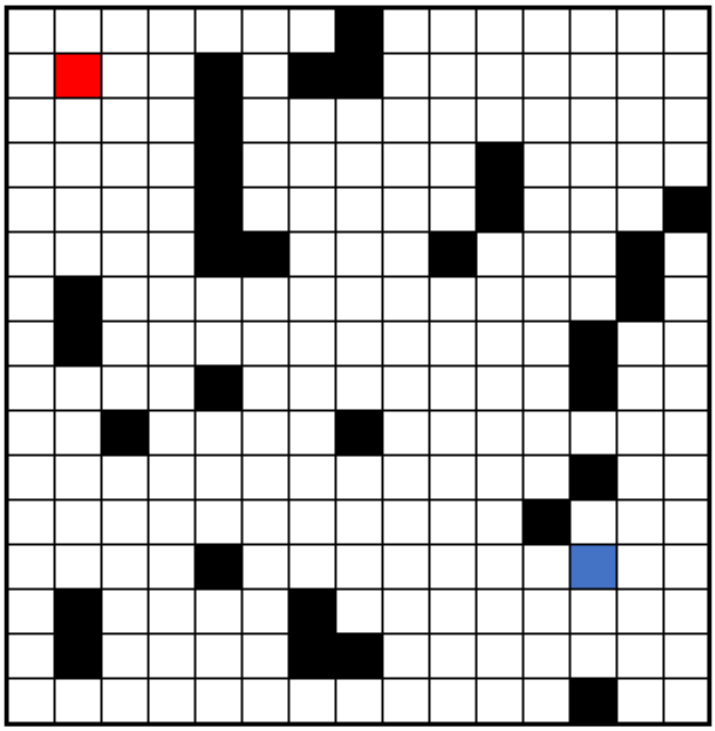

Рисунок 2.3 – Инициализация поля.

2 этап. Распространение волны.

Стартовая точка помечается как нулевая волна. От нее в четырех направлениях (или меньше, если по пути распространения волны стоит непроходимая «стена» или граница поля) распространяется первая волна. Точки, на которые распространяется первая волна, помечаются числом 1.

Пример распространения первой волны представлен на рисунке 2.4.

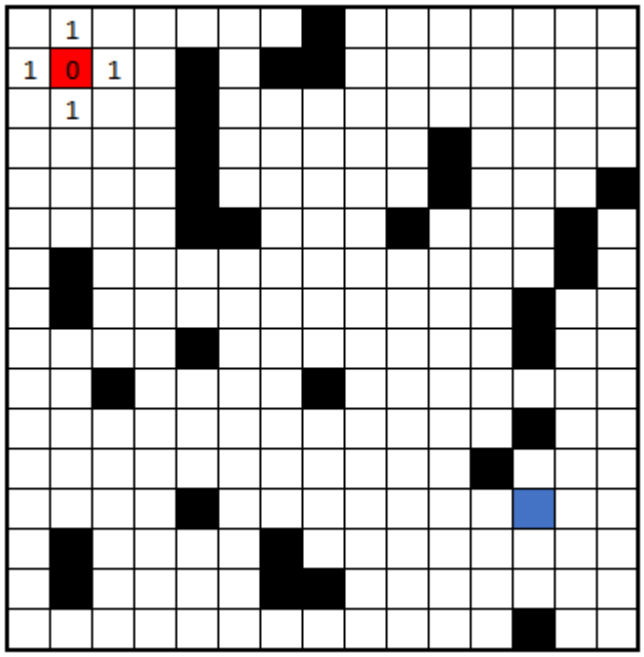

Рисунок 2.4 – Распространение первой волны.

От каждой точки, помеченной числом первой волной, распространяется вторая волна (во всех беспрепятственных направлениях). Точки, до которых распространилась вторая волна, помечаются числом 2. От этих же точек распространяется третья волна и так далее. На рисунке 2.5 изображено распространение четвертой волны.

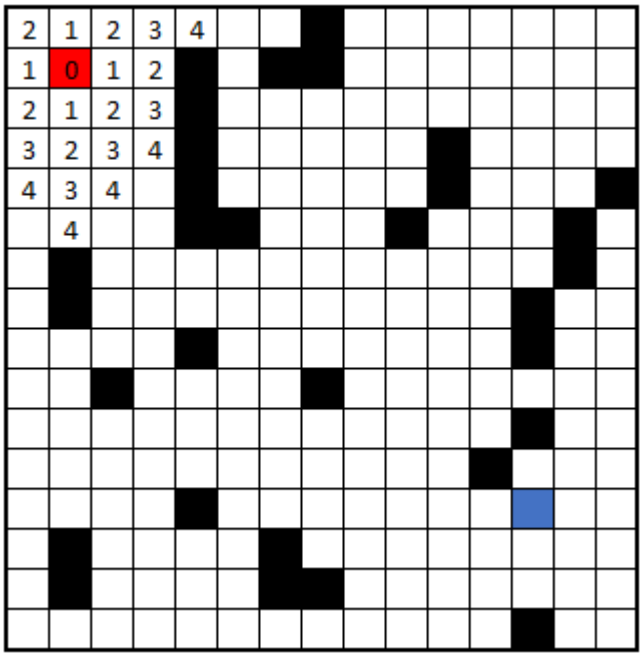

Рисунок 2.5 – Распространение четвертой волны.

На рисунке 2.6 изображено распространение 13-ой волны.

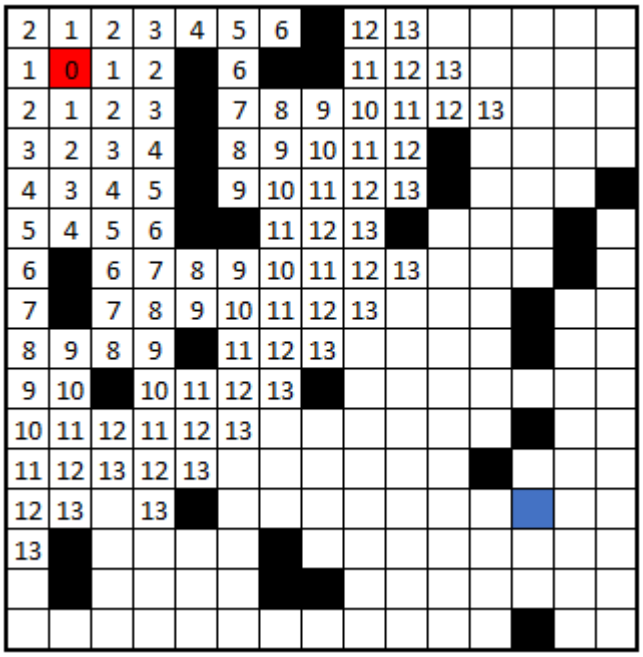

Рисунок 2.6 – Распространение 13-ой волны.

Числа, которые записываются в клетках, означают то количество шагов, за которое можно добраться от старта до этой точки или наоборот. Распространения волн продолжается до тех пор, пока не будет помечена числом финишная клетка или пока волна не распространится на все достжимые клетки поля.

Пример достижения финишной клетки 22-ой волной представлен на рисунке 2.7.

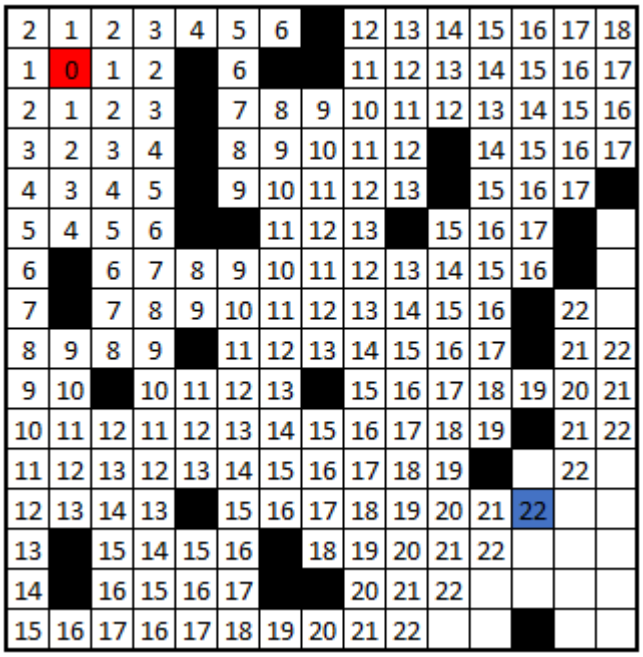

Рисунок 2.7 - Достижение финишной клетки 22-ой волной.

В случае, если до финишной клетки добраться невозможно (мешают «стены»), то есть алгоритм распространит волны на все достижимые клетки поля, но финишная клетка так и останется непомеченной, программа выдаст сообщение о невозможности построить путь.

Пример недостижимой финишной клетки представлен на рисунке 2.8.

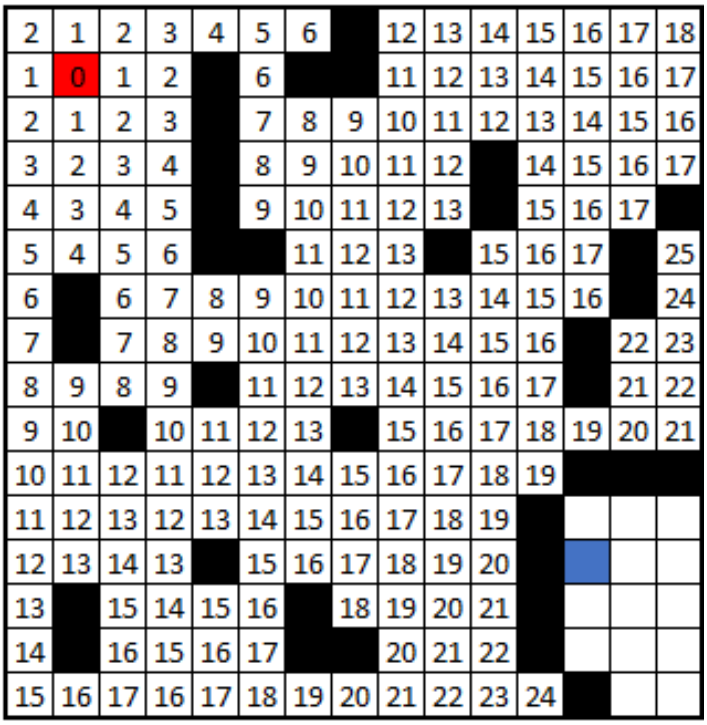

Рисунок 2.8 - Пример недостижимой финишной клетки.

3 этап. Прокладывание маршрута

Когда финишная клетка помечена, мы знаем сколько нужно шагов, чтобы добраться до нее. Маршрут до клетки строится в обратную сторону, то есть сначала мы записываем координаты финишной клетки, затем берем число, записанное в финишной клетке, и находим число на 1 меньше в соседних клетках, «перемещаемся» в эту клетку и записываем ее координаты. Далее в соседях уже этой клетки ищем число еще на 1 меньше и «перемещаемся» в следующую клетку, записывая ее координаты, и так далее пока не дойдем до 0 (стартовой клетки). Итого мы получаем последовательность координат клеток, которая является нашим маршрутом от стартовой до финишной точки.

Маршрутов минимальной длины может быть несколько, выбранный алгоритмом маршрут зависит от того, в каком порядке мы сравниваем соседей клетки, когда ищем число меньшее на единицу.

Пример построенного пути представлен на рисунке 2.9.

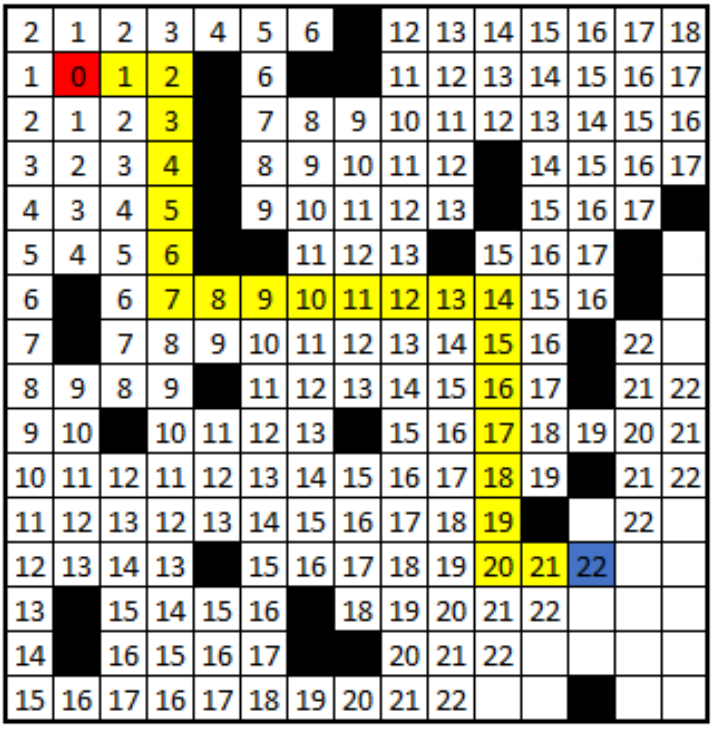

Рисунок 2.9. - Пример построенного пути.

2.3 Анализ метода

Оценим временную сложность алгоритма Ли:

Количество клеток на поле h * w = n

Этап распространения волны имеет временную сложность O(n), и в худшем случае цикл выполняется меньше, чем $\frac{4n}{2} = 2n$ раз.

Этап восстановления пути также имеет временную сложность O(n), и в любом случае цикл выполняется меньше, чем n раз.

Итог: Алгоритм Ли имеет линейную временную сложность O(n).

3. ИССЛЕДОВАНИЕ АЛГОРИТМА

Для исследования алгоритма для каждого эксперимента было выбрано:

- окрестность фон Неймана
- квадратное поле
- стартовая точка – левая верхняя (0, 0)
- финишная точка – правая нижняя (W-1, H-1)
- случайная генерация поля (каждая клетка с шансом 20% может сгенерироваться «стеной»)

Для разных экспериментов менялся размер поля (H * W = n)

Экспериментальное время работы алгоритма представлено в таблице 1.

Таблица 1 – Экспериментальные данные.

№ экспе-римена	сторона квадрата	100	320	550	700	900	1000
	количество клеток (n)	10000	102400	302500	490000	810000	1000000
1	время, мс	15	202	702	1655	3058	3734
2		15	155	687	1295	2718	3890
3		15	140	671	1405	2734	3803
4		15	140	671	1327	2750	3905
5		15	156	625	1343	2827	4375
6		15	155	655	1296	2734	4108
7		15	156	655	1343	2828	4078
8		15	140	640	1265	2718	4030
9		15	171	640	1296	2765	4015
10		15	171	686	1311	2812	4186
	среднее значение	15	158,6	663,2	1353,6	2794,4	4012,4

На рисунке 3.1 представлен график зависимости среднего время выполнения программы от количества клеток поля.

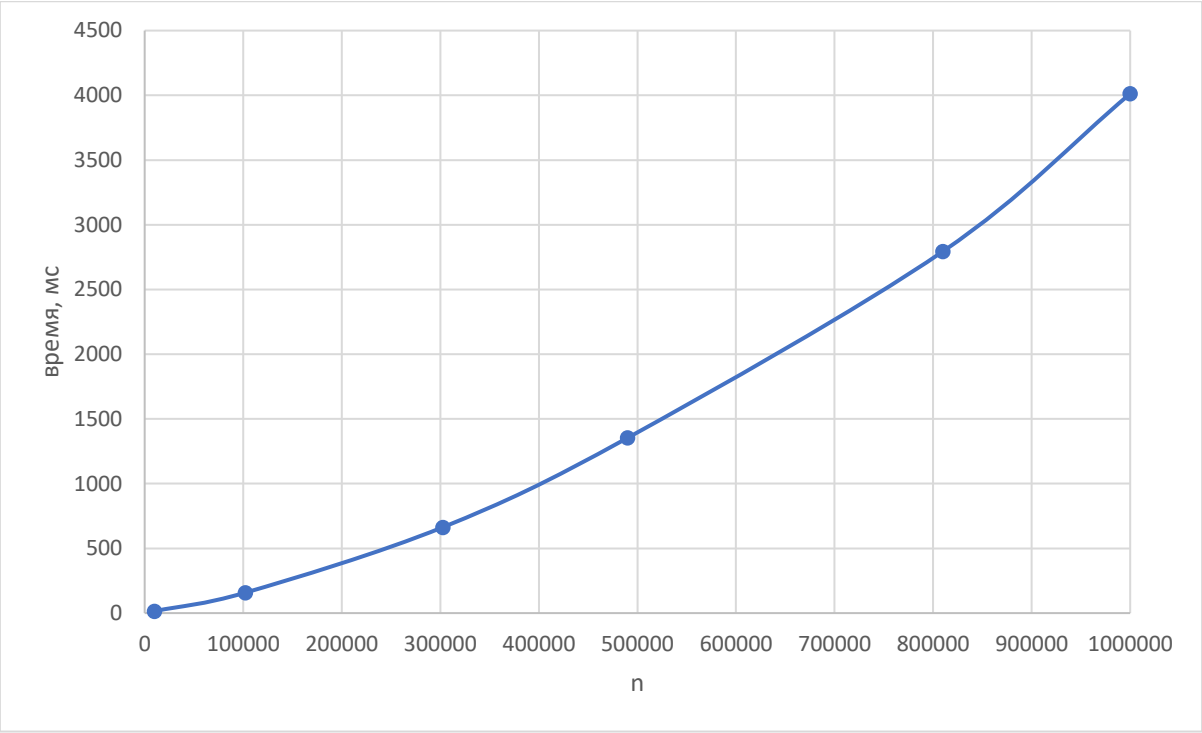

Рисунок 3.1 - График зависимости среднего время выполнения программы от количества клеток поля.

4. ЗАКЛЮЧЕНИЕ

В результате работы был изучен и разработан алгоритм Ли (волновой алгоритм), аналитическим и экспериментальным способом была получена средняя временная сложность алгоритма

СПИСОК ЛИТЕРАТУРЫ

- *Frank Rubin.* The Lee path connection algorithm // IEEE Transactions on Computers. — 1974.
- *Steven M. Rubin.* Computer Aids for VLSI Design. — 1994.
- *Максим Мозговой.* Занимательное программирование: Самоучитель. — СПб.: Питер, 2004.

ПРИЛОЖЕНИЕ

Полный код для исследования алгоритма на языке C++

lee.h

```
#ifndef LEE_LEE_H
#define LEE_LEE_H

class Grid
{
private:
    int W;        // ширина рабочего поля
    int H;        // высота рабочего поля
    int **grid;

private:
    void memory_alloc();
    void free_memory();

public:
    Grid();
    Grid(int W, int H);
    void Random (); // случайная генерация поля
    bool lee(int ax, int ay, int bx, int by);     // поиск пути из клетки (ax, ay) в клетку (bx,
by)
};

#endif
```

lee.cpp

```
#include <iostream>
#include <ctime>
#include "lee.h"

const int WALL   = -1;          // непроходимая клетка
const int BLANK  = -2;          // свободная непомеченная клетка



Grid::Grid() {
    H = 0;
    W = 0;
    grid = nullptr;
}

Grid::Grid(int W, int H) {
    this->H = H;
    this->W = W;

    for (int i = 0; i < W; i++) {
        for (int j = 0; j < H; j++) {
            grid[i][j] = BLANK;
        }
    }
}

void Grid::memory_alloc() {
    grid = new int* [H];
    for (int i = 0; i < H; i++) {
        grid[H] = new int[W];
    }
}

void Grid::free_memory() {
    for (int i = 0; i < H; i++) {
        free(grid[i]);
    }
    free(grid);
}
```

```
void Grid::Random() {    // случайная генерация поля
    for (int i = 0; i < W; i++) {
        for (int j = 0; j < H; j++) {
            if (rand() % 5 != 0) { // с вероятностью 1/5 каждая клетка может оказаться стеной
                grid[i][j] = BLANK;
            } else {
                grid[i][j] = WALL;
            }
        };
    };
}

bool Grid::lee(int ax, int ay, int bx, int by) {

        int px[W * H], py[W * H];       // координаты клеток, входящих  путь
        int len;                        // длина пути

        int dx[4] = {1, 0, -1, 0};    // смещения для определения соседей клетки  по x
        int dy[4] = {0, 1, 0, -1};    // и по y
        int wave, x, y, k;
        bool mark;

        if (grid[ay][ax] == WALL || grid[by][bx] == WALL) return false; // ячейка (ax, ay) или
(bx, by) - стена

        // распространение волны

        mark = false;  // не все возможные клетки помечены
        wave = 0;
        grid[ay][ax] = 0;             // стартовая ячейка помечена 0
        while ( !mark && grid[by][bx] == BLANK ) // пока не все возможные клетки помечены и
финишная клетка не помечена
        {
            mark = true;                // предполагаем, что все свободные клетки уже помечены
            for ( y = 0; y < H; ++y )
                for ( x = 0; x < W; ++x )
                    if ( grid[y][x] == wave )              // ячейка (x, y) помечена числом wave
                    {
                        for ( k = 0; k < 4; ++k )         // проходим по всем непомеченным соседям
                        {
                            int iy=y + dy[k], ix = x + dx[k];
                            if ( iy >= 0 && iy < H && ix >= 0 && ix < W &&
                                 grid[iy][ix] == BLANK )
                            {
                                mark = false;               // найдены непомеченные клетки
                                grid[iy][ix] = wave + 1;       // распространяем волну
                            }
                        }
                    }
            wave++;
        }


        if (grid[by][bx] == BLANK) {
            return false;
        }

        // восстановление пути

        len = grid[by][bx];             // длина кратчайшего пути из (ax, ay) в (bx, by)
        x = bx;
        y = by;
        wave = len;
        while ( wave > 0 )             // пока не достигнем 0
        {
            px[wave] = x;
            py[wave] = y;                    // записываем ячейку (x, y) в путь
            wave--;
            for (k = 0; k < 4; ++k)
            {
                int iy=y + dy[k], ix = x + dx[k];
                if ( iy >= 0 && iy < H && ix >= 0 && ix < W &&
                     grid[iy][ix] == wave)
                {
                    x = x + dx[k];
                    y = y + dy[k];            // переходим в ячейку, которая на 1 ближе к старту
                    break;
                }
            }
        }
```

```
        px[0] = ax;
        py[0] = ay;                    // теперь px[0..len] и py[0..len] - координаты ячеек пути
        return true;
}
```

main.cpp

```
#include <iostream>
#include <ctime>
#include "lee.h"

using namespace std;

int H = 100;
int W = 100;

int main() {

    srand(time(NULL));

    Grid Pole(H, W);
    Pole.Random();
    Pole.lee(0, 0, H-1, W-1);

    //cout << clock();
    return 0;
}
```