Оглавление

[Глава 1. Теоретический обзор алгоритмов и структур данных, используемых в решении, схожих структур и алгоритмов 3](#_Toc91456871)

[Глава 2 Практическая реализация алгоритма 6](#_Toc91456872)

[Заключение 16](#_Toc91456873)

[Список использованной литературы 17](#_Toc91456874)

[Приложение 18](#_Toc91456875)

# Глава 1. Теоретический обзор алгоритмов и структур данных, используемых в решении, схожих структур и алгоритмов

Для решения задачи о поиске пути на дискретном поле само поле удобно представлять, как граф. Есть два стандартных способа представления графа: как набор списков смежных вершин или как матрица смежности[1]. Но для данной задачи применение этих способов будет неэффективно, поскольку: заранее известно количество ребер у каждой клетки (вершины) (из каждой клетки можно переместиться на одну вверх, вниз, влево и вправо, то есть всего существует 4 ребра, также существуют 4 угловые клетки, у которых по два ребра) и количество вершин велико даже у небольших полей. В связи с этим, для экономии памяти и более наглядного представления, информацию о дискретном поле лучше хранить в массиве массивов, где в элементе будет храниться стоимость перехода на данную клетку, а индексы элемента будут одновременно служить и ссылкой, и названием вершины.

Для реализации данной структуры на языке C++ можно было бы использовать класс array, но гораздо лучше будет воспользоваться классом vector из библиотеки STL, так как vector обладает стандартными функциями (такими как size(), resize(), assign() и т.д.), которые значительно упрощают реализацию алгоритмов и функций необходимых для решения задач.

Один из алгоритмов, который можно использовать для нахождения пути в дискретном поле – это поиск в глубину (или dfs). В нём для хранения обработанных вершин используется список, а для запоминания смежных вершин – стек. Когда начальная вершина помещается в стек, начинается итерационный процесс, выталкивая последнюю добавленную вершину и обрабатывая её. Когда в стеке не остаётся вершин, процесс завершается и на выходе мы получаем список обработанных вершин. На каждом шаге применяется следующий алгоритм:

Берется вершина из стека, проверятся по списку, была ли она обработана. Если нет, то происходит её обработка, а также все её соседи добавляются в стек. Текущая вершина добавляется в список использованных. Перейти к следующей вершине из стека [2].

DFS имеет широкую область применения (поиск пути в графе, поиск наименьшего общего предка, поиск компонент связности и т.д.). Но для решения поставленной задачи поиск в глубину не подходит. DFS не может найти путь с самой низкой стоимостью, поскольку при выборе вершины на обработку первой пойдёт та, которая была добавлена последней, а не с самой низкой стоимостью прохода по ней. Но также поиск в глубину не может найти наилучший маршрут по количеству пройденных клеток из-за того, что сначала обрабатываются вершины добавленные последними, что не гарантирует самый короткий путь. Сложность данного алгоритма равна О(n^2) [2].

Следующий рассматриваемый алгоритм – это поиск в ширину (или bfs). Он представляет собой один из простейших алгоритмов для обхода графа и является основой для многих важных алгоритмов. Например, алгоритм Прима (Prim) или алгоритм Дейкстры (Dijkstra) [1]. С помощью bfs, как и с помощью dfs нельзя найти кратчайший путь в взвешенном графе. Поиск в ширину мог бы решить данную задачу, если бы порядок обработки вершин определялся весом ребра (но в таком случае, мы бы просто получили алгоритм Дейкстры). В тоже время, задачу о поиске пути наилучшего по количеству пройденных клеток, bfs выполняет. Также поиск в ширину решает задачи: поиск компонент связности, нахождение кратчайшего цикла и т.д. При этом сложность bfs равна сложности dfs - О(n^2) [2].

Ещё один алгоритм, для нахождения пути в графе – это волновой алгоритм (или алгоритм Ли). Алгоритм выполняется следующим образом:

Из начального элемента распространяется в четырёх (в случае поля) волна, проверяя проходимые эти клетки и не были ли мы в этой клетке раньше. Если оба эти условия выполняются, то в клетку записывается число шагов от стартовой вершины. Затем уже из этой клетки порождаются волны, до тех пор, пока не будет найдём путь между двумя вершинами или не произойдет перебор всех вершин.

Главное достоинство алгоритма Ли – нахождение любого существующего пути. Но из-за этого возникает и главный недостаток – большое потребление памяти. Основная область применения данного алгоритма: трассировка печатных плат, проложение пути и поиск кратчайшего расстояния в стратегических играх. Сложность алгоритма Ли равна О(n^2).

И последний рассматриваемый алгоритм – это А\*(«А звезда»). «А звезда» решает задачу нахождения кратчайшего пути в взвешенном графе. Порядок обхода вершин в данном алгоритме определяется эвристической функцией. В свою очередь эта функция является суммой двух других функций, функции стоимости достижения рассматриваемой вершины из начальной и функции эвристической оценки расстояния от текущей вершины до конечной. Последняя функция может меняться в зависимости от условий. Например, если можно перемещаться только в четырёх направлениях используется манхэттенское расстояние, если можно перемещаться по диагонали, то расстояние Чебышева и т.д. Алгоритм А\* всегда найдет лучшее решение (если оно существует) за кратчайший срок. Но данный алгоритм потребляет большое количество памяти, из-за необходимости хранить всю информацию об исследуемых вершинах. Поэтому, не смотря на то что А\* является лучшим алгоритмом для поиска кратчайшего пути в взвешенном графе, в ситуациях, когда объем потребляемой памяти является критичным, нужно использовать другие алгоритмы (например, алгоритм Дейкстры или «жадный» алгоритм). Сложность алгоритма «А звезда» равна O(n∙log(n)) и совпадает со сложностью алгоритма Дейкстры.

# Глава 2 Практическая реализация алгоритма

Для реализации алгоритмов я создал один класс **Way**. У данного класса есть лишь одно поле – **field**, хранящее информацию о дискретном поле. В классе Wayесть два конструктор. Первый принимает две переменные типа int (**length** и **width**), которые обозначают размер поля. Второй принимает те же переменные, которые вновь задают размер, но также третью переменную –**walkprice**, которая задает стоимость перемещения для всех клеток. Второй конструктор используется в случае, когда необходимо задать поле с одинаковой ценой перемещения между всеми клетками.

Way(int length, int width)

{

field.assign(length, vector<T>(width));

}

Way(int length, int width, T walkprice)

{

field.assign(length, vector<T>(width));

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

{

fill(field[i].begin(), field[i].end(), walkprice);

}

}

Листинг 2.1. Конструкторы класса **Way**

В классе **Way** присутствует несколько служебных функций. Пользователь не имеет к ним доступа и не может ими воспользоваться. Они применяются только в других функциях.

Одна из них таких функций **resize**. Данная функция служит для изменения размера переменной **field**, и, соответственно, принимает два параметра типа int. Изменение размера может быть необходимо при принятии информации о размере поле и цене прохода по ячейкам из файла. Если в файле будет информация о слишком большом (или маленьком поле), то переменная fieldпросто будет изменена.

Следующие служебные функции это **answer1** и **answer2**. Эти функции принимают начальную и конечную клетку и стоимость пути между ними. Цель обоих функций одна – кратко и понятно вывести информацию о стоимости пути между клетками, но область применения различна. **answer1** используется для вывода информации о самом коротком пути и применяется, когда стоимость перехода между клетками равняется 0 или 1. А **answer2** выводит информацию о самом дешевом пути.

И последняя служебная функция – это **h\_function**. Она используется в функции **A\_star** для вычисления примерного расстояния до конечной точки.  **h\_function** вычисляет это расстояния используя формулу манхэттенского расстояния.

int h\_function(pair<int, int> start, pair<int, int> finish)

{

return(abs(start.first - finish.first) + abs(start.second - finish.second));

}

Листинг 2.2. Вычисление эвристической оценки расстояния от текущей вершины до конечной для алгоритма «А звезда»

Для вычисления расстояния на дискретном поле я рассматриваю 4 алгоритма. Для каждого алгоритма была написана своя функция.

Первая функция для расчета расстояния это **bfs**. Соответственно, в ней для расчета расстояния применяется алгоритм поиска в ширину. В функцию передается два объекта класса pair <int, int> - это координаты начальной и конечной точки. В самой функции создается несколько переменных: очередь **q** (в ней хранятся данные о клетках, которые нужно обработать), вектор объектов класса bool, названный **used** (в нем помечаются уже использованные вершины), и вектор, хранящий объекты классы int, **d** (он хранит информацию о расстоянии от начальной точки до данной). Векторы **d** и **used** хранят в себе информацию о клетках из вектора векторов, поэтому для того чтобы найти номер клетки **d или used**, зная его в **field**, нужно было провести следующие вычисления: номер первого вектора в **field** умножить на число элементов во втором векторе и прибавить номер во втором векторе (например, x и y номера в **field**,тогда номер в **d** или **used** будет равен: x\*field[0].size() + y). Для того чтобы найти соседние клетки, я брал этот номер со значение +1, -1, +10, -10. У данного подхода есть существенный минус. Помимо того, что нужно проверять, что значение не вышло за границу массива, необходимо проверить, что при прибавлении или отнимании 1 значении не перешло на другую строчку (то есть, если номер равняется 60, то 59 не будет соседней клеткой и переходить туда запрещено).

if ((((v+1) % field[0].size()) !=0) &&!used[v+1])

{

used[v + 1] = true;

q.push(v + 1);

d[v + 1] = d[v] + 1;

}

if ((((v) % field[0].size()) != 0) && !used[v - 1])

{

used[v - 1] = true;

q.push(v - 1);

d[v - 1] = d[v] + 1;

}

if ((v /field[0].size() != (field.size()-1))&& !used[v + field[0].size()])

{

used[v + field[0].size()] = true;

q.push(v + field[0].size());

d[v + field[0].size()] = d[v] + 1;

}

if ((v / field[0].size() != 0) && !used[v - field[0].size()])

{

used[v - field[0].size()] = true;

q.push(v - field[0].size());

d[v - field[0].size()] = d[v] + 1;

}

Листинг 2.3. Код добавления соседних клеток в очередь **q**

Функция **bfs** способна вычислить самое короткое расстояние и самое дешевое расстояния (но только в случае если цена перехода между всеми клетками одинакова). Для вывода ответа в этих двух случаях предусмотрены разные функции **answer1** и **answer2**. Из-за этого в конце функции нужно провести проверку, чему равно значение в какой-нибудь клетке. Если они отличается от 0 или 1, значить у клеток есть, хоть и одинаковая, но цена прохода, значит нужно вывести ещё и стоимость пути.

answer1(start, finish, d[finish.first \* field[0].size() + finish.second]);

if (field[0][0] != 0 && field[0][0] != 1)

answer2(start, finish, d[finish.first \* field[0].size() + finish.second]\* field[0][0]);

Листинг 2.4. Вывод ответа в функции **bfs**

Следующая рассматриваемая функция – **dfs**, использует алгоритм поиска в глубину. Функция **dfs** очень сильно похожа на **bfs**.Единственное различия (помимо использования другого алгоритма): вместо использования очередь **q**, используется стек **s**. Но при этом результат работы алгоритмов отличается кардинально.

Следующая функция, основанная на волновом алгоритме, называется **lee**. Как и **bfs**,и **dfs**, **lee** принимает два объекта с координатами начальной и конечной точки. Расстояние от начальной точки до текущей хранится в векторе векторов **LEEfield**. Сначала мы заполняем его -1 (или любым другим числом, которое не будет встречать в расстояниях до точек). Также создаётся переменная **d** равная 0, и в **LEEfield** ячейка, соответствующая начальной точке, приравнивается к **d**. Далее я буду перебирать все ячейки **LEEfield** и обрабатывать те, значения которых равны **d**. В данной функции посещение соседних клеток будет организованно иначе, нежили чем в **bfs** и **dfs**. Было создано два массива **move1** и **move2**, в каждом из которых лежит по четыре элемента в **move1:** 1,0,-1,0; в **move2:** 0,1,0,-1. Для того чтобы посетить все соседние клетки каждой вершины с помощью цикла for я каждую итерацию буду изменять координаты текущей вершины прибавляя к ним элементы из массивов (например, x и y координаты текущей клетки, тогда координаты соседей будут равны y + move1[0] и x + move2[0], y + move1[1] и x + move2[1] и т.д.). Вывод ответа организован так же, как и в **bfs** и **dfs**.

И последняя функция **A\_star**, основана на алгоритме «А\_звезда». Данная функция создана для расчета самого дешевого пути в случае, когда каждая клетка имеет различную стоимость прохода. Для того чтобы подобное поле было удобно задать была написана для ввода информации из файла. Она принимает объект класс **Way**,считывает из файла размер поле (если размер поля в файле и размер поля объекта не совпадают, то применяется функция resize), затем считывая из файла числа последовательно присваивает их каждой клетке.

template<class T1>

istream& operator>>(istream& s, Way<T1>& A)

{

int length, width;

s >> length >> width;

if (A.field.size() != length || A.field[0].size() != width)

A.resize(length, width);

T1 walkprice;

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

{

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

{

s >> walkprice;

A.field[i][j] = walkprice;

}

}

return s;

}

Листинг 2.5. Функция ввод данных из файла

В функции **A\_star** создаются следующие переменные: вектор пар целочисленных чисел **open** (хранит вершины, которые надо обработать), вектор векторов значений bool, названный **closed** (в нём храниться информация о уже использованных вершинах), векторы векторов **Gfield** и **Ffield**, в первом храниться стоимость перемещения из начальной клетки в данную, во втором значение эвристической функции для каждой клетки. Переход к соседним клеткам осуществляется также, как и в функции **lee**, с помощью массивов **move1** и **move2**. Данная функция используются только для поиска самого дешёвого пути, поэтому для выведения ответа используется только функция **answer2**.

Интерфейс данной программы достаточно прост. Создаётся объект класс **Way**, с заданным размером поля (возможной ценой прохождения по каждой ячейке). Затем, если нужно вычислить самый короткий путь, то выбирается функция и в неё вносятся координаты начальной и конечной точки, если нужно вычислить самый дешёвый путь, то сначала используется функция ввода данных из файла, и лишь затем выбирается алгоритм **A\_star** и в него вносятся данные о начальных и конечных клетках.

Например, вот как работает функция **bfs:**

int main()

{

setlocale(LC\_ALL, "Russian");

Way <int> A(10, 10);

Way <int> С(39, 41);

Way<int> D(10, 10, 13);

Way<int> E(17, 5, 23);

A.bfs(make\_pair(2, 8), make\_pair(9, 1));

cout << "**\n**";

С.bfs(make\_pair(1, 8), make\_pair(37, 40));

cout << "**\n**";

D.bfs(make\_pair(1, 3), make\_pair(7, 9));

cout << "**\n**";

E.bfs(make\_pair(3, 3), make\_pair(16, 4));

}

Листинг 2.6. Пример работы функции **bfs** (здесь и далее строчка setlocale(LC\_ALL, "Russian") необходима, для того чтобы в консоли могла отображаться кириллица)

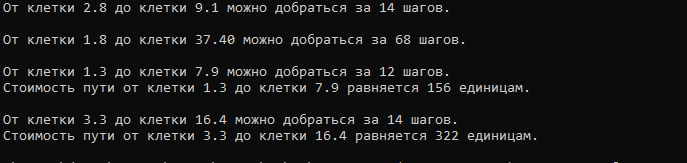


Рисунок 2.1. Результат работы функции **bfs**

Как видно из рисунка 2.1 функция находит кратчайший путь, а если у клеток задана цена перемещения по ним, то выводит и цену данного пути (который будет и самым дешевым поскольку он самый короткий, и цена перемещения одинакова для всех клеток.

Взглянем на работу функции **lee**:

int main()

{

setlocale(LC\_ALL, "Russian");

Way <int> A(10, 10);

Way <int> С(39, 41);

Way<int> D(10, 10, 13);

Way<int> E(17, 5, 23);

A.lee(make\_pair(2, 8), make\_pair(9, 1));

cout << "**\n**";

С.lee(make\_pair(1, 8), make\_pair(37, 40));

cout << "**\n**";

D.lee(make\_pair(1, 3), make\_pair(7, 9));

cout << "**\n**";

E.lee(make\_pair(3, 3), make\_pair(16, 4));

}

Листинг 2.7. Проверка работы функции **lee**

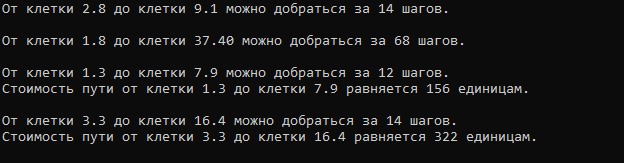


Рисунок 2.2. Результат работы функции **lee**

Как видно из рисунка 2.2 функция **lee**, запущенная с теми же данными, что и функция **bfs**, выдаёт тот же ответ.

Теперь взглянем на работу функции **dfs**, и сравним её с **bfs** и **lee:**

int main()

{

setlocale(LC\_ALL, "Russian");

Way <int> A(10, 10);

Way <int> С(39, 41);

Way<int> D(10, 10, 13);

Way<int> E(17, 5, 23);

A.dfs(make\_pair(2, 8), make\_pair(9, 1));

A.bfs(make\_pair(2, 8), make\_pair(9, 1));

A.lee(make\_pair(2, 8), make\_pair(9, 1));

cout << "**\n**";

С.dfs(make\_pair(1, 8), make\_pair(37, 40));

С.bfs(make\_pair(1, 8), make\_pair(37, 40));

С.lee(make\_pair(1, 8), make\_pair(37, 40));

cout << "**\n**";

D.dfs(make\_pair(1, 3), make\_pair(7, 9));

D.bfs(make\_pair(1, 3), make\_pair(7, 9));

D.lee(make\_pair(1, 3), make\_pair(7, 9));

cout << "**\n**";

E.dfs(make\_pair(3, 3), make\_pair(16, 4));

E.bfs(make\_pair(3, 3), make\_pair(16, 4));

E.lee(make\_pair(3, 3), make\_pair(16, 4));

}

Листинг 2.8. Сравнение работы функции **dfs** с другими функциями

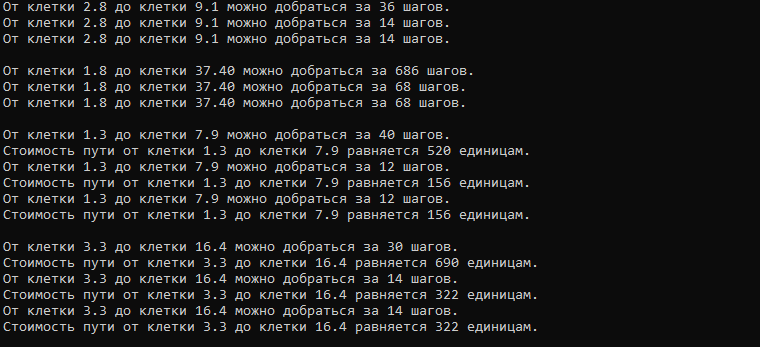


Рисунок 2.3. Сравнение результатов работы функции **dfs** с другими функциями

Как легко заметить из рисунка 2.3 ответы от функции **dfs**, выводятся самыми первыми, и они является самыми длинными. Как и было сказано ранее алгоритм поиск в глубину не способен найти ни самый короткий ни самый дешевый путь, и результаты работы программы это доказывают.

И взглянем же на результаты работы последней функции - **A\_star**. Сначала нужно загрузить данные из следующего файла:

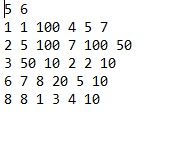


Рисунок 2.4. Данные из файла test.txt

setlocale(LC\_ALL, "Russian");

cout << "**\n**";

ifstream fin("D:**\\**test.txt");

if (fin.is\_open())

{

Way<int> T(1, 1);

fin >> T;

fin.close();

T.A\_star(make\_pair(1, 1), make\_pair(1, 3));

T.A\_star(make\_pair(2, 2), make\_pair(2, 4));

T.A\_star(make\_pair(0, 0), make\_pair(4, 5));

Листинг 2.9. Проверка ввода данных из файла и работы функции **A\_star**

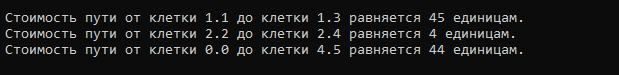


Рисунок 2.5. Результат работы функции **A\_star**

Из рисунка 2.5 можно убедится, что функция работает корректно.

# Заключение

Существует несколько алгоритмов для поиска пути на дискретном поле. Один из них, поиск в глубину, несмотря на то что, он может решить задачу в общем виде (найти путь), но не гарантируется, что данный путь будет кратчайшим или самым дешевым. У dfs широкая область применения (например, поиск наименьшего общего предка или поиск компонент связности), так что лучше использовать данный алгоритм для решения других задач, а поиск пути реализовать другими алгоритмами.

Такими как поиск в ширину и волновой алгоритм. Оба этих алгоритма могут найти кратчайший путь на дискретном поле. Для решения более простых и менее трудоёмких задачи лучше использовать dfs, поскольку он гораздо проще в реализации. Алгоритм Ли же используется в более серьёзных и трудозатратных задачах (как например трассировка плат), поскольку он более надёжен, но при всех его достоинства стоит учитывать, что для реализации данного алгоритма требуется много памяти.

И наилучший выбор для решения задач по поиску самого дешёвого пути на дискретном поле это алгоритм «А звезда». Если искомый путь вообще существует, то он найдет самый дешёвый путь быстрее всего. Но из-за главного недостатка алгоритма «А звезда» - потребления памяти, в некоторых задачах необходимо использовать альтернативные алгоритмы.

# Список использованной литературы

1. Кормен, Т. Алгоритмы: Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. [Пер. с англ.: К. Белов и др.] - М.: МЦНМО, 1999 - 955 с. ISBN 5-900916-37-5
2. Топп, У. Структуры данных в С++ / У. Топп, У. Форд. [Пер. с англ.: В. Кузьменко] - М. : ЗАО «Издательство БИНОМ», 1999. - 816 с

# Приложение

#include <iostream>

#include <fstream>

#include <queue>

#include <string>

#include <vector>

#include <cmath>

#include <stack>

#include <locale.h>

using namespace std;

template <class T>

class Way

{

protected:

vector<vector<T>> field;

void resize(int length, int width)

{

field.resize(length);

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

{

field[i].resize(width);

}

}

void answer1(pair<int, int> start, pair<int, int> finish, int aswer)

{

cout << " От клетки " << start.first << '.' << start.second << " до клетки " << finish.first << '.' << finish.second << " можно добраться за " << aswer << " шагов." << '\n';

}

void answer2(pair<int, int> start, pair<int, int> finish, T aswer)

{

cout << " Стоимость пути от клетки " << start.first << '.' << start.second << " до клетки " << finish.first << '.' << finish.second << " равняется " << aswer << " единицам." << '\n';

}

int h\_function(pair<int, int> start, pair<int, int> finish)

{

return(abs(start.first - finish.first) + abs(start.second - finish.second));

}

public:

Way(int length, int width)

{

field.assign(length, vector<T>(width));

}

Way(int length, int width, T walkprice)

{

field.assign(length, vector<T>(width));

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

{

fill(field[i].begin(), field[i].end(), walkprice);

}

}

int bfs(pair<int, int> start, pair<int, int> finish)

{

int s = start.first \* field[0].size() + start.second;

queue<int> q;

q.push(s);

vector<bool> used(field.size()\* field[0].size());

vector<int> d(field.size() \* field[0].size());

used[s] = true;

while (!q.empty()) {

int v = q.front();

q.pop();

if ((((v+1) % field[0].size()) !=0) &&!used[v+1])

{

used[v + 1] = true;

q.push(v + 1);

d[v + 1] = d[v] + 1;

}

if ((((v) % field[0].size()) != 0) && !used[v - 1])

{

used[v - 1] = true;

q.push(v - 1);

d[v - 1] = d[v] + 1;

}

if ((v /field[0].size() != (field.size()-1))&& !used[v + field[0].size()])

{

used[v + field[0].size()] = true;

q.push(v + field[0].size());

d[v + field[0].size()] = d[v] + 1;

}

if ((v / field[0].size() != 0) && !used[v - field[0].size()])

{

used[v - field[0].size()] = true;

q.push(v - field[0].size());

d[v - field[0].size()] = d[v] + 1;

}

}

answer1(start, finish, d[finish.first \* field[0].size() + finish.second]);

if (field[0][0] != 0 && field[0][0] != 1)

answer2(start, finish, d[finish.first \* field[0].size() + finish.second]\* field[0][0]);

return d[finish.first \* field[0].size() + finish.second];

}

int dfs(pair<int, int> start, pair<int, int> finish)

{

int f = start.first \* field.size() + start.second;

stack<int> s;

vector<int> d(field.size() \* field[0].size());

vector<bool> used(field.size() \* field[0].size());

s.push(f);

used[f] = true;

while (!s.empty())

{

int now = s.top();

s.pop();

if ((((now + 1) % field[0].size()) != 0) && !used[now + 1])

{

used[now + 1] = true;

s.push(now + 1);

d[now + 1] = d[now] + 1;

}

if ((((now) % field[0].size()) != 0) && !used[now - 1])

{

used[now - 1] = true;

s.push(now - 1);

d[now - 1] = d[now] + 1;

}

if ((now / field[0].size() != (field.size() - 1)) && !used[now + field[0].size()])

{

used[now + field[0].size()] = true;

s.push(now + field[0].size());

d[now + field[0].size()] = d[now] + 1;

}

if ((now / field[0].size() != 0) && !used[now - field[0].size()])

{

used[now - field[0].size()] = true;

s.push(now - field[0].size());

d[now - field[0].size()] = d[now] + 1;

}

}

answer1(start, finish, d[finish.first \* field[0].size() + finish.second]);

if (field[0][0] != 0 && field[0][0] != 1)

answer2(start, finish, d[finish.first \* field[0].size() + finish.second] \* field[0][0]);

return d[finish.first \* field[0].size() + finish.second];

}

int lee(pair<int, int> start, pair<int, int> finish)

{

vector<vector<T>> LEEfield = field;

for (int i = 0; i < LEEfield.size(); i++)

{

fill(LEEfield[i].begin(), LEEfield[i].end(), -1);

}

int d = 0;

LEEfield[start.first][start.second] = d;

int move1[4] = { 1,0,-1,0 };

int move2[4] = { 0,1,0,-1 };

while (LEEfield[finish.first][finish.second] == -1)

{

for (int y = 0; y < field.size(); ++y)

for (int x = 0; x < field[0].size(); ++x)

if (LEEfield[y][x] == d)

{

for (int k = 0; k < 4; ++k)

{

int iy = y + move1[k], ix = x + move2[k];

if (iy > -1 && iy < field.size() && ix > -1 && ix < field[0].size() &&

LEEfield[iy][ix] == -1)

{

LEEfield[iy][ix] = d + 1;

}

}

}

d++;

}

answer1(start, finish, LEEfield[finish.first][finish.second]);

if (field[0][0] != 0 && field[0][0] != 1)

answer2(start, finish, LEEfield[finish.first][finish.second] \* field[0][0]);

return LEEfield[finish.first][finish.second];

}

T A\_star(pair<int, int> start, pair<int, int> finish)

{

vector<pair<int, int>> open;

vector<vector<bool>> closed;

closed.assign(field.size(), vector<bool>(field[0].size()));

vector<vector<T>> Gfield = field;

vector<vector<T>> Ffield = field;

int move1[4] = { 1,0,-1,0 };

int move2[4] = { 0,1,0,-1 };

for (int i = 0; i < field.size(); i++)

{

fill(closed[i].begin(), closed[i].end(), false);

}

open.push\_back(start);

Gfield[start.first][start.second] = 0;

Ffield[start.first][start.second] = Gfield[start.first][start.second] + h\_function(start, finish);

while (open.size() != 0)

{

bool tentative\_is\_better;

auto curent = open.back();

for (auto i : open)

{

if (Ffield[i.first][i.second] < Ffield[curent.first][curent.second])

curent = i;

}

if (curent == finish)

{

answer2(start, finish, Ffield[finish.first][finish.second]);

return Ffield[finish.first][finish.second];

}

open.erase(remove(open.begin(), open.end(), curent), open.end());

closed[curent.first][curent.second] = true;

for (int k = 0; k < 4; ++k)

{

int iy = curent.first + move1[k], ix = curent.second + move2[k];

if((iy>-1 && ix>-1)&& iy< closed.size() && ix< closed[0].size())

{

if (closed[iy][ix] == true)

continue;

auto tentative\_g\_score = Gfield[curent.first][curent.second] + field[iy][ix];

if (find(open.begin(), open.end(), make\_pair(iy, ix)) == open.end())

{

open.push\_back(make\_pair(iy, ix));

tentative\_is\_better = true;

}

else

{

if (tentative\_g\_score < Gfield[iy][ix])

tentative\_is\_better = true;

else

tentative\_is\_better = false;

}

if (tentative\_is\_better == true)

{

Gfield[iy][ix] = tentative\_g\_score;

Ffield[iy][ix] = Gfield[iy][ix] + h\_function(make\_pair(iy, ix), finish);

}

}

}

}

return -1;

}

template<class T1>

friend istream& operator>>(istream& s, Way<T1>& A);

};

template<class T1>

istream& operator>>(istream& s, Way<T1>& A)

{

int length, width;

s >> length >> width;

if (A.field.size() != length || A.field[0].size() != width)

A.resize(length, width);

T1 walkprice;

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

{

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

{

s >> walkprice;

A.field[i][j] = walkprice;

}

}

return s;

}

int main()

{

setlocale(LC\_ALL, "Russian");

cout << "\n";

ifstream fin("D:\\test.txt");

if (fin.is\_open())

{

Way<int> T(1, 1);

fin >> T;

fin.close();

T.A\_star(make\_pair(1, 1), make\_pair(1, 3));

T.A\_star(make\_pair(2, 2), make\_pair(2, 4));

T.A\_star(make\_pair(0, 0), make\_pair(4, 5));

}

Way <int> A(10, 10);

Way <int> С(39, 41);

Way<int> D(10, 10, 13);

Way<int> E(17, 5, 23);

A.dfs(make\_pair(2, 8), make\_pair(9, 1));

A.bfs(make\_pair(2, 8), make\_pair(9, 1));

A.lee(make\_pair(2, 8), make\_pair(9, 1));

cout << "\n";

С.dfs(make\_pair(1, 8), make\_pair(37, 40));

С.bfs(make\_pair(1, 8), make\_pair(37, 40));

С.lee(make\_pair(1, 8), make\_pair(37, 40));

cout << "\n";

D.dfs(make\_pair(1, 3), make\_pair(7, 9));

D.bfs(make\_pair(1, 3), make\_pair(7, 9));

D.lee(make\_pair(1, 3), make\_pair(7, 9));

cout << "\n";

E.dfs(make\_pair(3, 3), make\_pair(16, 4));

E.bfs(make\_pair(3, 3), make\_pair(16, 4));

E.lee(make\_pair(3, 3), make\_pair(16, 4));

}