Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

КУРСОВАЯ РАБОТА

по курсу “Логика и основы алгоритмизации в инженерных задачах”

на тему: “Реализация алгоритма поиска путей в лабиринте”

Выполнил:

студент группы 23ВВВ4

Соснин Глеб

к.т.н., доцент Юрова О. В.

Пенза, 2024

**Содержание**

Введение……………………………………………………………. 3

1. Постановка задач……………………………………………….. 4
2. Теоретическая часть работы………………………………….... 5
3. Описание алгоритма программы………………………………. 7
4. Описание программы…………………………………………... 9
5. Тестирование…………………………………………………… 13

Заключение………………………………………………………… 19

Список литературы………………………………………………… 21

Приложение А. Листинг программы……………………………... 22

**Введение**

Цель – разработка программы для поиска путей в лабиринте.

В работе рассмотрена реализация алгоритма Ли для поиска путей в лабиринте. Установлено, что с помощью данного алгоритма можно найти кротчайший путь в лабиринте.

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

В качестве среды разработки мною была выбрана среда MicrosoftVisualStudio2022, язык программирования – С++.

Целью курсовой работы является разработка программы для автоматической генерации лабиринта и нахождения кратчайшего пути через него с использованием алгоритма волновой трассировки. Программа будет включать элементы взаимодействия с пользователем, отображение результата в консоли и функционал сохранения/чтения лабиринта из файла.

1. **Постановка задач**
2. Создать рабочее поле (лабиринт).
3. Реализовать генерацию лабиринта.
4. Включить функционал сохранения и загрузки.
5. Реализовать алгоритм поиска пути.
6. Отобразить лабиринт и путь в консоли.
7. Реализовать приветственное окно.
8. Создать обработку ошибок.
9. Обеспечить завершение работы.
10. **Теоретическая часть работы**

Лабиринт представляет собой дискретное рабочее поле (ДРП), ограниченное замкнутой линией (стены) и разбитое на прямоугольные ячейки. Задача состоит в нахождении кратчайшего пути от стартовой ячейки к финишной.

Ячейки в лабиринте делятся на следующие категории:

Проходимые – свободные ячейки, по которым алгоритм может передвигаться.

Непроходимые – стены, препятствующие движению.

Стартовая ячейка – начальная точка маршрута.

Финишная ячейка – конечная точка маршрута.

Рабочее поле (лабиринт) в данной программе представлено в виде двумерной матрицы. Каждая ячейка матрицы может принимать следующие значения:

-2 (стена) – непроходимые области.

-1 (пустое пространство) – свободные ячейки, по которым возможен путь.

0 и больше – числа, указывающие шаги алгоритма волнового распространения (временные метки).

-3 (путь) – финальная траектория, определяемая после завершения алгоритма.

Лабиринт создается случайным образом с использованием алгоритма: – Лабиринт изначально полностью заполняется стенами. – Внутри матрицы случайным образом формируются проходы. – Каждый проход строится с учетом проверки на тупиковые ситуации, чтобы в результате лабиринт оставался проходимым.

Для поиска пути используется волновой алгоритм. Этот метод также называют методом распространения волны.

Если финишная точка недостижима, программа выводит сообщение об отсутствии пути.

Лабиринт всегда генерируется таким образом, чтобы не было изолированных областей, то есть каждая часть лабиринта соединена с другими.

Алгоритмы поиска пути применяются в различных областях в реальной жизни:

Робототехника – для прокладки маршрутов роботов.

Компьютерные игры – для создания искусственного интеллекта, который может находить пути.

Логистика – для оптимизации маршрутов доставки.

Картография – в задачах навигации и планирования маршрутов.

Данный подход, несмотря на простоту, демонстрирует эффективную работу в условиях ограниченного пространства и является базовым методом для более сложных алгоритмов.

1. **Описание алгоритма программы**

Программа инициализирует лабиринт как массив стенок и выполняет генерацию, случайно прокладывая пути внутри массива, чтобы сформировать проходимый лабиринт. Используется метод случайного блуждания с проверкой тупиков.

Сгенерированный лабиринт сохраняется в файл (labyrinth.txt) для последующего использования.

Если файл лабиринта уже существует, программа считывает его. Это позволяет повторно использовать один и тот же лабиринт без необходимости повторной генерации.

Алгоритм распространяет "волну" от начальной точки до конечной, заполняя ячейки числовыми значениями, которые представляют расстояние от начальной точки.

После достижения конечной точки программа "отматывает" путь назад, чтобы определить кратчайший маршрут. Этот маршрут помечается символами \*.

Лабиринт отображается в консоли, где:

# – стенки лабиринта,

пробелы – проходимые области,

\* – кратчайший путь.

Итоговый лабиринт, включая кратчайший путь, сохраняется в файл для анализа.

После выполнения всех операций программа завершает свою работу.

Лабиринт представлен в виде двумерного массива N x N, где каждая ячейка может быть стеной (#) или проходом (.).

Внутренние ячейки заполняются случайным образом: стена или проход. Начальная ячейка (1,1) и конечная ячейка (N-2, N-2) всегда остаются проходами (.).

Поиск кратчайшего пути.

Псевдокод алгоритма:

1. Инициализируем очередь queue и добавляем начальную ячейку (1,1).
2. Задаем начальное расстояние dist[1][1] = 0.
3. Определяем массивы направлений:

* dx = {-1, 0, 1, 0} – движение вверх, вправо, вниз, влево по оси x.
* dy = {0, 1, 0, -1} – движение вверх, вправо, вниз, влево по оси y.

1. Пока очередь не пуста:

* Извлекаем текущую ячейку (x, y).
* Для каждой из 4 возможных сторон проверяем соседние ячейки (nx, ny):
* Если ячейка (nx, ny) доступна (.) и еще не посещена (dist[nx][ny] == -1):
* Помечаем ячейку расстоянием dist[nx][ny] = dist[x][y] + 1.
* Добавляем (nx, ny) в очередь.
* Если достигнута конечная точка (N-2, N-2), BFS завершается.

Восстановление пути

Псевдокод восстановления пути:

1. Начинаем с конечной точки (N-2, N-2).
2. Идем по ячейкам с уменьшающимися значениями dist до ячейки (1,1):

* Если ячейка (nx, ny) имеет значение на 1 меньше, чем текущая (x, y), перемещаемся в (nx, ny) и помечаем её как часть пути \*.

1. Путь сохраняется в вектор и выводится в консоль.
2. **Описание программы**

Программа написана на языке C++, который обеспечивает высокую производительность, гибкость работы с памятью и поддержку структур данных. Код реализован в виде консольного приложения с интуитивно понятным взаимодействием с пользователем.

Проект был создан в виде консольного приложения Win64 (Visual C++).

Данная программа является многомодульной, поскольку состоит из нескольких функций.

Основные функции программы:

*showWelcomeMessage*() — выводит приветственное сообщение пользователю.

*getLabyrinthSize*() — запрашивает у пользователя размер лабиринта.

*generateMaze*() — создает лабиринт с начальной и конечной точками.

*printMaze*() — отображает лабиринт на экране.

*saveMazeToFile*() — сохраняет лабиринт в текстовый файл labyrinth.txt.

*BFS*() — реализует поиск кратчайшего пути методом BFS.

*markPath*() — отмечает кратчайший путь в лабиринте.

*printPath*() — выводит последовательность координат кратчайшего пути.

Функция *generateMaze*() создает лабиринт, представленный в виде двумерного массива символов:

# — стена.

. — свободная ячейка.

Случайное размещение стен происходит с использованием генератора случайных чисел (*rand*()). Границы лабиринта всегда остаются стенами, а начальная (1,1) и конечная (N-2,N-2) ячейки всегда проходимы.

Функция *printMaze*() выводит сгенерированный лабиринт и отмеченный путь.

Функция *printPath*() выводит координаты найденного пути в виде последовательности точек.

Если путь найден, программа выводит лабиринт с помеченным путем и последовательность координат.

Если путь отсутствует, программа выводит сообщение: "Путь не найден."

**Функция генерации лабиринта.**

void generateMaze(vector<vector<char>>& maze, int N)

{

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

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

if (i == 0 || j == 0 || i == N - 1 || j == N - 1) {

maze[i][j] = '#';

}

else {

maze[i][j] = '.'; }

}

}

srand(static\_cast<unsigned>(time(0))); // Инициализация генератора случайных чисел

for (int i = 1; i < N - 1; ++i) {

for (int j = 1; j < N - 1; ++j) {

if (rand() % 3 == 0) {

maze[i][j] = '#'; // Создание случайных стен

}

}

}

maze[1][1] = '.';

maze[N - 2][N - 2] = '.';

}

// Вывод лабиринта на экран

void printMaze(const vector<vector<char>>& maze) {

for (const auto& row : maze) {

for (const auto& cell : row) {

cout << cell << " ";

}

cout << endl;

}

}

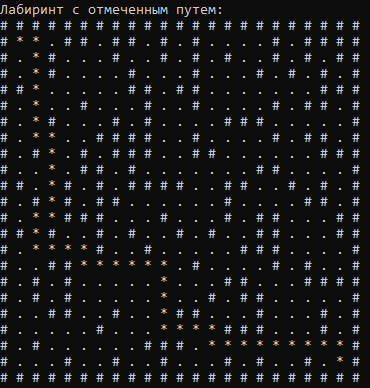


Рисунок 1 – Выведенный на экран лабиринт

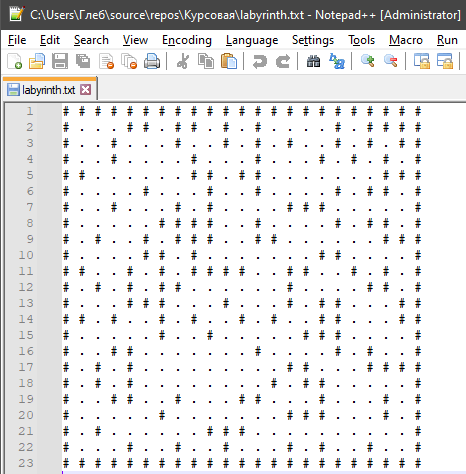


Рисунок 2 – Лабиринт, сохраненный в файле

1. **Тестирование**

Среда разработки MicrosoftVisualStudio 2019 предоставляет все средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций. Чтобы изменить размеры лабиринта мы должны поменять значение глобальной переменной N. Задали размеры лабиринта – N =17.

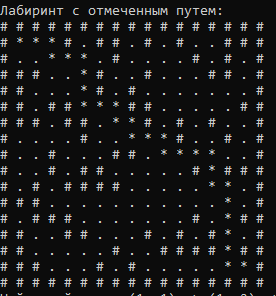


Рисунок 3 - Результат тестирования при размерах лабиринта 17x17

На рисунке 3 продемонстрирован результат тестирования при размерах лабиринта 17x17. Наш лабиринт записался в файл и теперь при следующем запуске с N = 17, наш лабиринт будет браться из файла (рисунок 4).

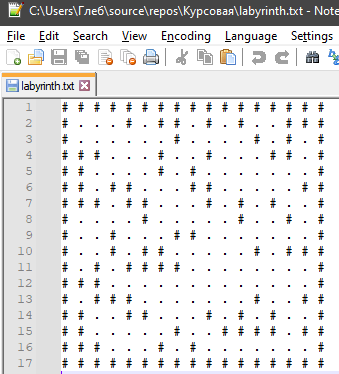


Рисунок 4 – Лабиринт, записанный в файл

Запускаем нашу программу еще раз, как видим наш лабиринт такой же как и в файле (рисунок 5).

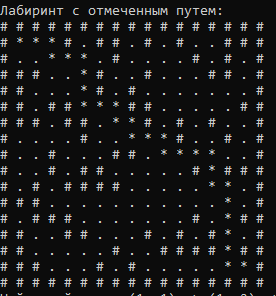


Рисунок 5 – Лабиринт, взятый из файла

Задали размеры лабиринта – N =9.

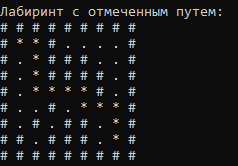


Рисунок 6 – Результат тестирования при размерах лабиринта 9х9

На рисунке 6 продемонстрирован результат тестирования при размерах лабиринта 9х9. Наш лабиринт записался в файл и теперь при следующем запуске с N = 9, наш лабиринт будет браться из файла (рисунок 7).

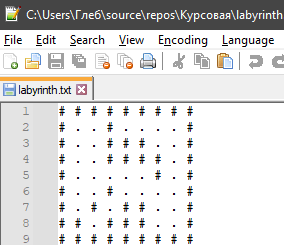


Рисунок 7 – Лабиринт 9х9, записанный в файл

Запускаем нашу программу еще раз, как видим наш лабиринт такой же как и в файле (рисунок 8).

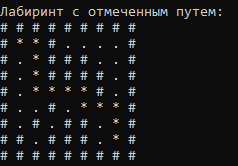


Рисунок 8 – Лабиринт 9х9

Задали размеры лабиринта – N =27.

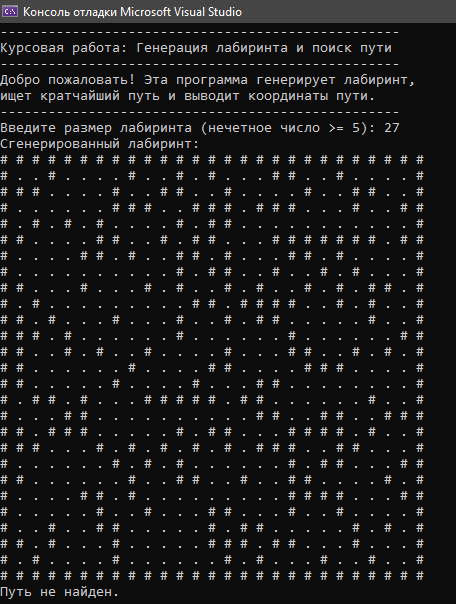


Рисунок 9 - Результат тестирования при размерах лабиринта 27х27 (путь не найден)

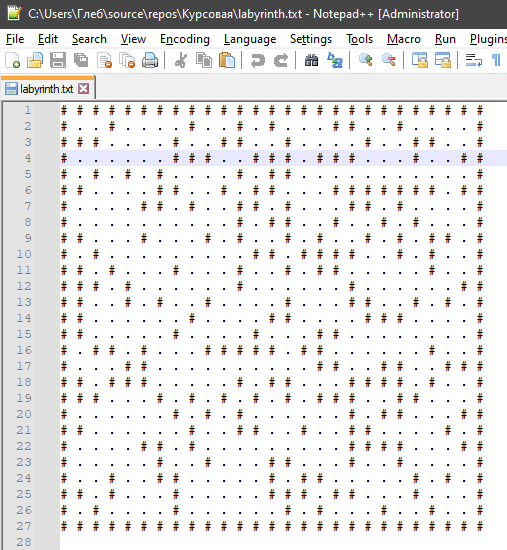


Рисунок 10 – Результат, записанный в файл

В результате тестирования было выявлено, что программа успешно генерирует лабиринт и находит кротчайший путь.

Таблица 1 – Описание поведения программы при тестировании

|  |  |  |
| --- | --- | --- |
| Описание теста | Ожидаемый результат | Полученный результат |
| Запуск программы | Запуск окна | Верно |
| Генерирование лабиринта | Вывод сгенерированного лабиринта на экран | Верно |
| Поиск пути | Вывод на экран кротчайшего пути в лабиринте | Верно |

В результате тестирования было выявлено, что программа успешно выполняет необходимые требования.

**Заключение**

Таким образом, в процессе создания данного проекта разработана программа, реализующая алгоритм поиска кротчайшего пути в лабиринте в MicrosoftVisualStudio 2022.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания алгоритма, для генерации лабиринте. Приобретены навыки по осуществлению алгоритма поиска кротчайшего пути в лабиринте. Углублены знания языка программирования C++.

Преимущества работы алгоритма поиска путей лабиринта:

1. Алгоритм поиска в ширину (BFS) прост в понимании и реализации. Он не требует сложных структур данных, что делает его доступным для начинающих программистов.
2. BFS гарантирует нахождение кратчайшего пути в ненагруженных графах, таких как лабиринт с одинаковым "весом" путей, благодаря последовательному исследованию уровней.
3. Программа визуально отмечает путь в лабиринте с помощью символов \*, а также выводит последовательность координат, что удобно для проверки корректности работы алгоритма.
4. Генерация лабиринта с использованием случайных стен делает каждое прохождение уникальным, что полезно для тестирования различных сценариев и демонстрации алгоритма.
5. Пользователь может задать размер лабиринта, что позволяет адаптировать алгоритм для различных случаев — от маленьких до больших лабиринтов.
6. Сохранение лабиринта в файл упрощает повторный анализ результатов и позволяет сохранять их для последующего использования.

Недостатки работы алгоритма поиска путей лабиринта:

1. BFS использует очередь для хранения всех узлов текущего уровня, что может привести к значительному потреблению памяти при больших размерах лабиринта.
2. Алгоритм генерации стен основан на случайных числах, что иногда приводит к созданию лабиринтов с неравномерной сложностью или слишком легких/сложных путей.
3. BFS подходит только для лабиринтов с равнозначными путями. В случае графов с разным "весом" путей алгоритм не находит оптимальное решение.
4. Временная сложность алгоритма линейно зависит от количества ячеек лабиринта. При больших значениях N это может привести к значительным задержкам.
5. Алгоритм генерации лабиринта создает стены случайно, что иногда приводит к тупиковым лабиринтам или слишком легкому прохождению.
6. Алгоритм требует ввода нечетного размера лабиринта, что может быть неудобно для некоторых пользователей или приложений.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

**Список литературы**

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и анализ - М.: МЦНМО, 2001. - 960 с.
2. Кристофидес Н. «Теория графов. Алгоритмический подход» - Мир, 1978
3. Герберт Шилдт «Полный справочник по C++» - Вильямс, 2006
4. Уилсон Р. Введение в теорию графов. Пер. с анг. 1977, 208 с.
5. Харви Дейтел, Пол Дейтел. Как программировать на C/C++. 2009 г.
6. 3. Оре О. Графы и их применение: Пер. с англ. 1965, 176 с.

**Приложение А**

**Листинг программы:**

#include <iostream>

#include <queue>

#include <vector>

#include <fstream>

#include <cstdlib>

#include <ctime>

#include <stack>

using namespace std;

struct Point {

int x, y;

};

// Массивы для движения в 4 направлениях: вверх, вправо, вниз, влево

int dx[] = { -1, 0, 1, 0 };

int dy[] = { 0, 1, 0, -1 };

// Функция для отображения приветственного сообщения

void showWelcomeMessage() {

cout << "--------------------------------------------------" << endl;

cout << "Курсовая работа: Генерация лабиринта и поиск пути" << endl;

cout << "--------------------------------------------------" << endl;

cout << "Добро пожаловать! Эта программа генерирует лабиринт," << endl;

cout << "ищет кратчайший путь и выводит координаты пути." << endl;

cout << "--------------------------------------------------" << endl;

}

// Проверка корректности входного размера лабиринта

int getLabyrinthSize() {

int N;

while (true) {

cout << "Введите размер лабиринта (нечетное число >= 5): ";

cin >> N;

if (cin.fail() || N < 5 || N % 2 == 0) {

cout << "Ошибка: размер должен быть нечетным числом >= 5." << endl;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

else {

break;

}

}

return N;

}

// Генерация лабиринта

void generateMaze(vector<vector<char>>& maze, int N) {

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

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

if (i == 0 || j == 0 || i == N - 1 || j == N - 1) {

maze[i][j] = '#';

}

else {

maze[i][j] = '.';

}

}

}

srand(static\_cast<unsigned>(time(0)));

for (int i = 1; i < N - 1; ++i) {

for (int j = 1; j < N - 1; ++j) {

if (rand() % 3 == 0) {

maze[i][j] = '#';

}

}

}

maze[1][1] = '.';

maze[N - 2][N - 2] = '.';

}

// Вывод лабиринта на экран

void printMaze(const vector<vector<char>>& maze) {

for (const auto& row : maze) {

for (const auto& cell : row) {

cout << cell << " ";

}

cout << endl;

}

}

// Сохранение лабиринта в блокнот

void saveMazeToFile(const vector<vector<char>>& maze, const string& filename) {

ofstream outFile(filename);

for (const auto& row : maze) {

for (const auto& cell : row) {

outFile << cell << " ";

}

outFile << endl;

}

outFile.close();

}

// Поиск пути методом BFS (поиск в ширину)

bool BFS(vector<vector<char>>& maze, int N, vector<vector<int>>& dist, vector<vector<Point>>& parent) {

queue<Point> q;

q.push({ 1, 1 });

dist[1][1] = 0;

while (!q.empty()) {

Point p = q.front();

q.pop();

for (int i = 0; i < 4; ++i) {

int nx = p.x + dx[i];

int ny = p.y + dy[i];

if (nx >= 0 && ny >= 0 && nx < N && ny < N && maze[nx][ny] == '.' && dist[nx][ny] == -1) {

dist[nx][ny] = dist[p.x][p.y] + 1;

parent[nx][ny] = p;

q.push({ nx, ny });

if (nx == N - 2 && ny == N - 2) {

return true;

}

}

}

}

return false;

}

void markPath(vector<vector<char>>& maze, vector<vector<Point>>& parent, int N) {

Point p = { N - 2, N - 2 };

while (!(p.x == 1 && p.y == 1)) {

maze[p.x][p.y] = '\*';

p = parent[p.x][p.y];

}

maze[1][1] = '\*';

}

// Вывод координат найденного пути

void printPath(const vector<vector<Point>>& parent, int N) {

vector<Point> path;

Point p = { N - 2, N - 2 };

while (!(p.x == 1 && p.y == 1)) {

path.push\_back(p);

p = parent[p.x][p.y];

}

path.push\_back({ 1, 1 });

cout << "Найденный путь: ";

for (int i = path.size() - 1; i >= 0; --i) {

cout << "(" << path[i].x << ", " << path[i].y << ")";

if (i > 0) cout << " -> ";

}

cout << endl;

}

int main() {

setlocale(LC\_ALL, "Russian");

showWelcomeMessage();

int N = getLabyrinthSize();

vector<vector<char>> maze(N, vector<char>(N));

generateMaze(maze, N);

cout << "Сгенерированный лабиринт:" << endl;

printMaze(maze);

saveMazeToFile(maze, "labyrinth.txt");

vector<vector<int>> dist(N, vector<int>(N, -1));

vector<vector<Point>> parent(N, vector<Point>(N));

if (BFS(maze, N, dist, parent)) {

markPath(maze, parent, N);

cout << "Лабиринт с отмеченным путем:" << endl;

printMaze(maze);

printPath(parent, N);

}

else {

cout << "Путь не найден." << endl;

}

return 0;

}