

# Пояснительная записка к проекту PathFinder

Подготовил: Жахонгир Ахмадалиев

Email: [jahamarsi@gmail.com](mailto:jahamarsi@gmail.com)

## Оглавление

- Введение
- Гипотеза
- Задача
- Вербальная модель решения
- Математическая модель
- Программная реализация
  - Структура проекта
  - Ключевые компоненты
    - Основной алгоритм поиска пути (BFS)
    - Чтение и парсинг файла лабиринта
    - Визуализация результата
  - Инструменты DevOps
    - Автоматизация сборки и тестирования
    - Профили сборки
    - Скрипт сборки
- Тестирование
- Анализ результатов
  - Пример 1: Существует путь
  - Пример 2: Путь не существует
- Выводы
- Список литературы

## Введение

В данной работе представлено решение задачи поиска пути в лабиринте с применением технологий DevOps. Задача заключается в разработке программы, которая находит оптимальный путь от начальной точки до конечной в заданном лабиринте. Проект реализован на языке программирования Rust

с использованием современных подходов к разработке, тестированию и развертыванию программного обеспечения.

## Гипотеза

Применение алгоритма поиска в ширину (BFS) в сочетании с современными инструментами DevOps позволит создать эффективное, надежное и легко поддерживаемое решение для нахождения оптимального пути в лабиринте. Использование языка Rust обеспечит высокую производительность и безопасность при работе с памятью, а внедрение практик DevOps улучшит качество кода и упростит процесс разработки.

## Задача

Разработать PathFinder, снабдить его инструментарием DevOps

Герой решил поехать в гости к другу. Перед ним лежит лабиринт.  
Помогите герою найти путь.

Лабиринт задан текстовым файлом в UTF-8

На первой строке задана ширина лабиринта (от 1 до 2048)

Во второй строке задана высота лабиринта (от 1 до 2048)

Далее на  $n$  строках задан лабиринт где :

'\_' -- пустое поле

'#' -- стена

1 -- Положение первого рыцаря

F -- Положение финальной точки (выхода из лабиринта)

Программа должна печатать последовательность ходов (клеток), по которым должен пройти рыцаря или писать, что прохода нет.

Формат вывода --  $m$  шагов с адресами клеток.

Допускается визуализация пути

Вход	Ответ	Вход	Ответ
5	x:1, y:2	5	Прохода нет
5	x:2, y:2	5	
#####	x:3, y:2	#####	
1#	x:4, y:2	1#	
####	x:4, y:3	####	

#_ #	x:4, y:4	##	
#F ###	x:3, y:4	#F ###	
	x:2, y:4		
	x:2, y:5		

## Вербальная модель решения

Для решения задачи поиска пути в лабиринте будем использовать алгоритм поиска в ширину (BFS). Этот алгоритм гарантирует нахождение кратчайшего пути в невзвешенном графе, каким является наш лабиринт.

Алгоритм работает следующим образом:

1. Начинаем с исходной точки (позиция '1').
2. Помещаем эту точку в очередь и отмечаем как посещенную.
3. Пока очередь не пуста, извлекаем точку из очереди.
4. Если эта точка – финальная (позиция 'F'), то путь найден.
5. Иначе, для каждой соседней клетки (вверх, вправо, вниз, влево), если она не является стеной и не посещена ранее:
  - Добавляем её в очередь.
  - Отмечаем как посещенную.
  - Запоминаем, из какой клетки мы в неё пришли.
6. Если очередь опустела, а финальная точка не найдена, то путь не существует.
7. Если путь найден, восстанавливаем его от финальной точки к начальной, используя записи о предыдущих клетках.

## Математическая модель

Представим лабиринт как двумерный массив (матрицу) символов  $M[i][j]$ , где  $i$  – номер строки,  $j$  – номер столбца. Каждый элемент может быть одним из следующих символов: '#' (стена), '\_' (пустая клетка), '1' (начальная точка), 'F' (конечная точка).

Формально, задача состоит в нахождении последовательности клеток  $P = \{p_1, p_2, \dots, p_n\}$ , где:

- $p_1$  – начальная клетка (соответствует символу '1')
- $p_n$  – конечная клетка (соответствует символу 'F')

- Для любых соседних клеток  $p_k$  и  $p_{k+1}$  выполняется условие: они являются соседними в лабиринте по горизонтали или вертикали, и  $M[p_{k+1}]$  не является стеной.

Используя алгоритм BFS, мы будем строить дерево поиска, где:

- Вершины – клетки лабиринта
- Ребра соединяют соседние клетки (по горизонтали и вертикали)
- Корень – начальная клетка

Пусть  $S$  – множество посещенных клеток,  $Q$  – очередь клеток для обработки,  $P[i][j]$  – матрица, где для каждой клетки указана предыдущая клетка в пути.

Алгоритм можно описать следующим образом:

1. Инициализация:  $S = \{p_1\}$ ,  $Q = [p_1]$ ,  $P[i][j] = \text{null}$  для всех  $i, j$
2. Пока  $Q$  не пуста:
  - а. Извлечь клетку  $c$  из начала очереди  $Q$
  - б. Если  $c$  – конечная клетка, завершить поиск с. Для каждой соседней клетки  $n$  клетки  $c$ :
    - Если  $n$  не является стеной и  $n \notin S$ :
      - Добавить  $n$  в  $S$
      - Добавить  $n$  в конец  $Q$
      - Установить  $P[n.y][n.x] = c$
3. Если алгоритм завершился без нахождения конечной клетки, путь не существует
4. Иначе, восстановить путь от конечной клетки к начальной, используя матрицу  $P$

## Программная реализация

### Структура проекта

Проект реализован на языке Rust и имеет следующую структуру:

```
pathfinder/
├─ Cargo.toml      # Файл конфигурации Rust проекта
├─ Makefile        # Файл для автоматизации сборки и тестирования
├─ build.rs        # Скрипт для настройки сборки
└─ src/
```

```

|   |— main.rs          # Основной код программы
|   |— lib.rs           # Библиотечные функции
|   |— bin/
|       |— generate_maze.rs # Утилита для генерации лабиринтов
|— tests/                # Тесты
|— examples/              # Примеры использования

```

## Ключевые компоненты

### Основной алгоритм поиска пути (BFS)

Реализация алгоритма поиска в ширину (BFS) представлена в функции `find_path`. Она принимает лабиринт в виде двумерного массива символов и возвращает последовательность точек, представляющих путь от начальной до конечной точки.

```

pub fn find_path(maze: &Vec<Vec<char>>, width: usize, height: usize) ->
Option<Vec<Point>> {
    let (start, end) = find_start_end(maze, width, height)?;

    let mut queue = VecDeque::new();
    let mut visited = HashSet::new();
    let mut parent = vec![vec![None; width]; height];

    queue.push_back(start);
    visited.insert(start);

    while let Some(current) = queue.pop_front() {
        if current == end {
            return Some(reconstruct_path(&parent, end, start));
        }

        for next in get_adjacent_cells(&current, maze, width, height) {
            if !visited.contains(&next) {
                queue.push_back(next);
                visited.insert(next);
                parent[next.y][next.x] = Some(current);
            }
        }
    }

    None
}

```

### Чтение и парсинг файла лабиринта

Функция `parse_maze_file` отвечает за чтение файла лабиринта и его преобразование во внутреннее представление:

```
fn parse_maze_file<P: AsRef<Path>>(path: P) -> io::Result<(Vec<Vec<char>>,
    usize, usize)> {
    let file = File::open(path)?;
    let mut lines = io::BufReader::new(file).lines();

    let width: usize = lines.next()
        .ok_or_else(|| io::Error::new(io::ErrorKind::InvalidData,
            "Отсутствует информация о ширине"))?
        .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?
        .parse()
        .map_err(|_| io::Error::new(io::ErrorKind::InvalidData,
            "Некорректная ширина"))?;

    let height: usize = lines.next()
        .ok_or_else(|| io::Error::new(io::ErrorKind::InvalidData,
            "Отсутствует информация о высоте"))?
        .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?
        .parse()
        .map_err(|_| io::Error::new(io::ErrorKind::InvalidData,
            "Некорректная высота"))?;

    let maze: Vec<Vec<char>> = lines
        .map(|line| line.map(|l| l.chars().collect()))
        .collect::<Result<Vec<Vec<char>>, _>>()>;

    // Проверяем, что размеры лабиринта совпадают с указанными
    if maze.len() != height || maze.iter().any(|row| row.len() != width) {
        return Err(io::Error::new(io::ErrorKind::InvalidData, "Размеры
лабиринта не соответствуют указанным"));
    }

    Ok((maze, width, height))
}
```

## Визуализация результата

Программа предоставляет несколько форматов вывода результата:

1. Текстовый формат: координаты точек пути
2. Визуальный формат: отображение лабиринта с отмеченным путем
3. JSON формат: представление результата в формате JSON

```
fn visualize_maze(maze: &Vec<Vec<char>>, path: &Vec<Point>) {
    let mut visual_maze = maze.clone();
```

```

// Отмечаем путь символом '*', кроме начала и конца
for &point in path.iter().skip(1).rev().skip(1) {
    visual_maze[point.y][point.x] = '*';
}

// Выводим лабиринт
for row in &visual_maze {
    for &cell in row {
        print!("{}", cell);
    }
    println!();
}
}

```

## Инструменты DevOps

### Автоматизация сборки и тестирования

Для автоматизации процессов сборки, тестирования и развертывания используется Makefile:

```

.PHONY: all build test coverage bench clean doc install package

# Основная цель - собрать проект
all: build test

# Сборка проекта
build:
    cargo build

# Сборка в production режиме
release:
    cargo build --release

# Запуск тестов
test:
    cargo test

# Тесты с детализацией
test-verbose:
    cargo test -- --nocapture

# Генерация отчета о покрытии
coverage:
    cargo install cargo-tarpaulin --force
    cargo tarpaulin --ignore-tests --out Html

# Запуск бенчмарков
bench:
    cargo bench

```

```
# Очистка результатов сборки
clean:
    cargo clean

# Генерация документации
doc:
    cargo doc --no-deps

# Установка бинарного файла
install:
    cargo install --path .

# Создание установочного пакета
package:
    cargo install cargo-deb --force
    cargo deb
```

## Профили сборки

В файле `Cargo.toml` настроены профили для разработки и производственной среды:

```
[profile.dev]
opt-level = 0
debug = true
debug-assertions = true
overflow-checks = true
lto = false

[profile.release]
opt-level = 3
debug = false
strip = true
lto = true
codegen-units = 1
panic = "abort"
```

## Скрипт сборки

Файл `build.rs` используется для настройки процесса сборки и генерации тестовых данных:

```
use std::env;
use std::fs;
use std::path::Path;

fn main() {
    println!("cargo:rerun-if-changed=build.rs");
}
```



```
println!("cargo:rerun-if-changed=src/");

// Создаем директорию для тестовых данных, если она не существует
let out_dir = env::var("OUT_DIR").unwrap();
let test_data_dir = Path::new(&out_dir).join("test_data");

if !test_data_dir.exists() {
    fs::create_dir_all(&test_data_dir).unwrap();

    // Создаем тестовый файл лабиринта для тестов
    let test_maze_path = test_data_dir.join("test_maze.txt");
    fs::write(
        &test_maze_path,
        "5\n5\n#####\n1___#\n###_#\n#___#\n#F###\n",
    ).unwrap();

    println!("cargo:warning=Test data generated in: {}",
test_data_dir.display());
}

// Определяем переменные окружения для различных сред
if env::var("PROFILE").unwrap() == "release" {
    println!("cargo:rustc-cfg=production");
} else {
    println!("cargo:rustc-cfg=development");
}
}
```

## Тестирование

Тестирование программы выполняется с использованием встроенного в Rust фреймворка для тестирования. Тесты покрывают основные функции программы:

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::fs;
    use std::path::Path;
    use std::io::Write;

    // Вспомогательная функция для создания временного файла с лабиринтом
    fn create_test_maze(content: &str) -> String {
        let temp_dir = env::temp_dir();
        let file_path = temp_dir.join("test_maze.txt");

        let mut file = fs::File::create(&file_path).unwrap();
        file.write_all(content.as_bytes()).unwrap();

        file_path.to_str().unwrap().to_string()
    }
}
```

```

#[test]
fn test_path_exists() {
    let maze = vec![
        vec!['#', '#', '#', '#', '#'],
        vec!['1', '-', '-', '-', '#'],
        vec!['#', '#', '#', '-', '#'],
        vec!['#', '-', '-', '-', '#'],
        vec!['#', 'F', '#', '#', '#'],
    ];
    let path = find_path(&maze, 5, 5).unwrap();
    assert_eq!(path.len(), 9);
    assert_eq!(path[0], Point { x: 0, y: 1 });
    assert_eq!(path[8], Point { x: 1, y: 4 });
}

#[test]
fn test_no_path() {
    let maze = vec![
        vec!['#', '#', '#', '#', '#'],
        vec!['1', '-', '-', '-', '#'],
        vec!['#', '#', '#', '#', '#'],
        vec!['#', '-', '-', '-', '#'],
        vec!['#', 'F', '#', '#', '#'],
    ];
    assert_eq!(find_path(&maze, 5, 5), None);
}

#[test]
fn test_parse_maze_file() {
    let content = "5\n5\n#####\n1__#\\n###_#\\n#___#\\n#F###\\n";
    let file_path = create_test_maze(content);

    let (maze, width, height) = parse_maze_file(&file_path).unwrap();

    assert_eq!(width, 5);
    assert_eq!(height, 5);
    assert_eq!(maze[0][0], '#');
    assert_eq!(maze[1][0], '1');
    assert_eq!(maze[4][1], 'F');
}
}

```

## Анализ результатов

Для анализа результатов работы программы были проведены тесты на различных входных данных. Рассмотрим два примера:

### Пример 1: Существует путь

Входные данные:

```
5
5
#####
1__#
###_#
#__#
#F###
```

**Ожидаемый результат:**

```
x:1, y:2
x:2, y:2
x:3, y:2
x:4, y:2
x:4, y:3
x:4, y:4
x:3, y:4
x:2, y:4
x:2, y:5
```

**Результат работы программы:**

```
x:1, y:2
x:2, y:2
x:3, y:2
x:4, y:2
x:4, y:3
x:4, y:4
x:3, y:4
x:2, y:4
x:2, y:5
```

**Визуализация пути:**

```
#####
1***#
****#
```

```
#####  
#F###
```

Результат совпадает с ожидаемым. Программа правильно нашла кратчайший путь от начальной точки до конечной.

## Пример 2: Путь не существует

Входные данные:

```
5  
5  
#####  
1__#  
#####  
#__#  
#F###
```

Ожидаемый результат:

Прохода нет

Результат работы программы:

Прохода нет

Программа корректно определила, что путь между начальной и конечной точками отсутствует.

## Выводы

В результате выполнения работы была успешно разработана программа PathFinder, которая решает задачу поиска пути в лабиринте. Программа удовлетворяет всем заданным требованиям:

1. Корректно читает и обрабатывает входные данные из файла.
2. Эффективно находит кратчайший путь в лабиринте с помощью алгоритма поиска в ширину (BFS).

3. Предоставляет результат в различных форматах: текстовый, визуальный и JSON.
4. Корректно обрабатывает случаи, когда путь отсутствует.

Использование языка Rust обеспечило высокую производительность и безопасность программы.

## Список литературы

1. Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2015. – 368 с.
2. Документация Rust. [Электронный ресурс]. – Режим доступа: <https://doc.rust-lang.org/book/> (дата обращения: 15.03.2025).
3. Документация Cargo. [Электронный ресурс]. – Режим доступа: <https://doc.rust-lang.org/cargo/> (дата обращения: 15.03.2025).