Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материала и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Алгоритм Декстры, A*, Lee

Выполнил студент гр. 3331506/20102 Дёгтева Д.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

Содержание

Введение

Поиск кратчайшего пути в графе — одна из фундаментальных задач в области информатики, дискретной математики и алгоритмики. Она заключается в нахождении оптимального маршрута между двумя вершинами графа с учетом структуры и весов рёбер.

Предметом данной работы являются три алгоритма: Дейкстры, A* и Ли — ключевые методы поиска кратчайшего пути, применяемые в различных типах графов и структурах данных. Алгоритм Дейкстры предназначен для графов с неотрицательными весами рёбер и обеспечивает нахождение кратчайших путей от одной вершины ко всем остальным. Алгоритм A* расширяет возможности метода Дейкстры за счёт использования эвристических оценок, что делает его особенно эффективным при наличии информации о направлении к цели. Алгоритм Ли, в свою очередь, применяется преимущественно в целочисленных решётках (сетках) и лабиринтах и гарантирует нахождение кратчайшего пути при наличии проходимых и непроходимых ячеек.

Постановка задачи заключается в сравнительном исследовании указанных алгоритмов с точки зрения принципов их работы, вычислительной сложности, применимости к различным типам графов и эффективности при решении задач поиска пути. Для этого предполагается провести как теоретический анализ, так и экспериментальное моделирование на случайных и типовых графах с различными характеристиками.

Значимость задачи обусловлена широким спектром практических применений — от построения маршрутов в GPS-устройствах и системах управления беспилотниками до вычислений в логистических системах и цифровом моделировании. Понимание различий между алгоритмами и условий их применения позволяет выбирать наиболее подходящее решение для конкретной задачи, что критично для оптимизации ресурсов и времени выполнения в программных и аппаратных системах.

Алгоритм Дейкстры

Алгоритм Дейкстры является одним из наиболее известных и широко используемых методов для нахождения кратчайших путей в графах с неотрицательными весами ребер. Он был предложен голландским ученым Эдсгером Дейкстрой в 1959 году в статье *"A note on two problems in connexion with graphs"* [1].

Применяется в различных областях, включая: навигационные системы (поиск оптимального маршрута на карте), компьютерные сети (выбор маршрутов передачи данных), робототехника (планирование движения), анализ транспортных систем и сетей.

Алгоритм Дейкстры работает по принципу жадного поиска: он последовательно находит кратчайшие пути от начальной вершины до всех остальных вершин графа. На каждом шаге выбирается вершина с минимальным известным расстоянием от начальной точки, после чего выполняется релаксация смежных вершин (обновление их расстояний, если найден более короткий путь).

На вход алгоритм получает взвешенный граф *G(V,E)*, где V — множество вершин, E — множество ребер. Пусть a некоторая вершина от которой будем искать кратчайшие пути до остальных вершин.

Алгоритм:

1. Каждой вершине из V сопоставим метку – минимальное известное расстояние от этой вершины до a. На начальном этапе метки всех вершин, кроме a, равны ∞ (это означает, что расстояние до них пока не известно). Также введем множество p(V), которое содержит предпоследнюю вершину на пути от a к любой вершине из V.
2. Все вершины графа помечаются как непосещенные;
3. Пока все вершины не посещены:
   - Выбрать еще не посещенную вершину u, метка которой минимальна.
   - Пометить вершину u как посещенную;

– Для каждого соседа u (соседи u – вершины, в которые ведут ребра из u) рассмотрим новую длину пути, вычисляемую как сумма метки u и длины ребра из u к соседу.
– Если полученное значение меньше текущей метки соседа, то заменяем его метку полученным значением пути, а также вносим вершину u в множество p.

Пример работы алгоритма представлен на рисунке 1.

```
Алгоритм Дейкстры:
Вершина        Расстояние        Путь
  1               0               1
  2               10              1->2
  3               50              1->4->3
  4               30              1->4
  5               60              1->4->3->5
```

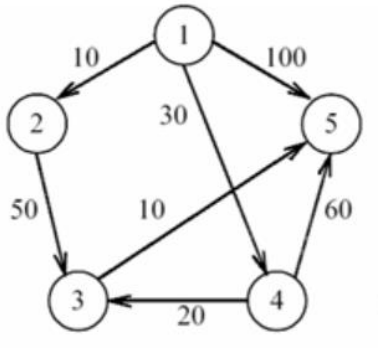

Рисунок 1 – Пример работы алгоритма Дейкстры

Эксперимент:

Для исследования временной сложности алгоритма Дейкстры были проведены эксперименты на случайных графах разного размера. Генерация графов осуществлялась с использованием псевдослучайных чисел. Измерения времени выполнялись с помощью стандартной библиотеки `<chrono>`. Алгоритм запускался на графах с числом вершин от 0 до 1000 с шагом 50.

Время работы алгоритма Дейкстры складывается из двух основных составляющих: время нахождения вершины с наименьшей меткой и время изменения значения метки при необходимости. Первая операция потребует O(n) времени, а вторая операция – O(m) времени (n – количество вершин графа; m – количество ребер графа).

Первая операция выполняется O(n) раз, а вторая O(m) раз. Таким образом итоговая ассимптотика будет равна O($n^2 + m$). Слагаемое m можно отбросить в том случае, когда граф сильно разрежен.

График зависимости времени выполнения алгоритма от количества вершин графа изоыбражен на рисунке 2.

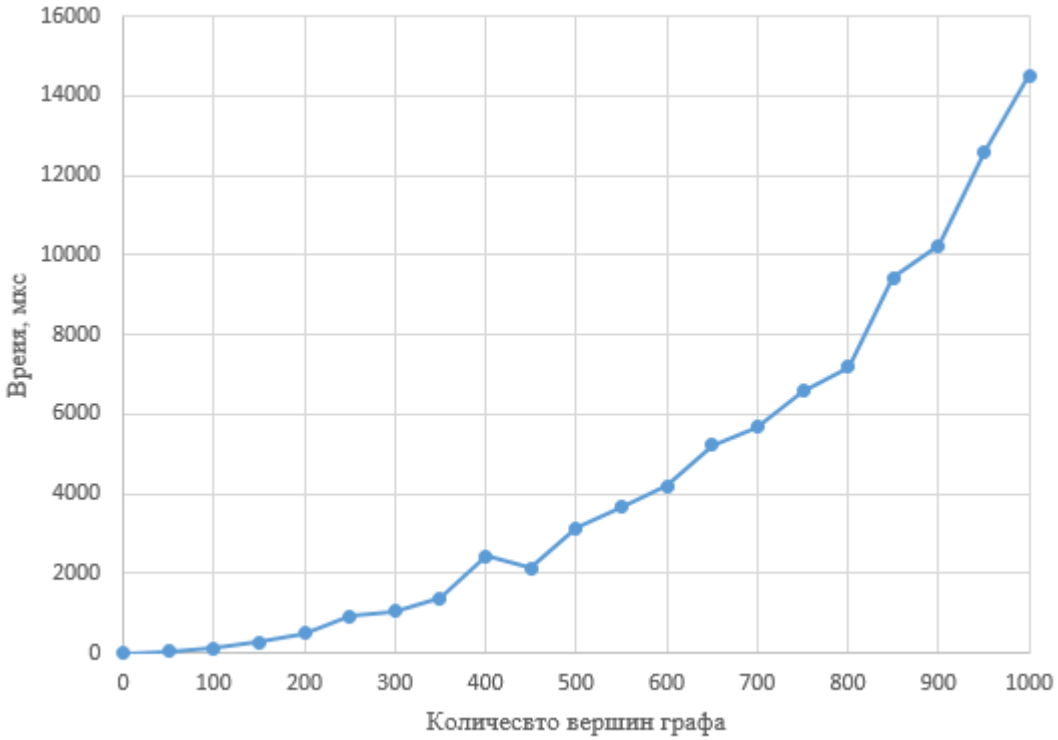

Рисунок 2 – График зависимости времени выполнения алгоритма от количества вершин графа

Алгоритм A*

Алгоритм A* (A-star) является одной из наиболее популярных модификаций алгоритма Дейкстры для задачи поиска кратчайшего пути. Он был впервые описан Питером Хартом, Нильсом Нильсоном и Бертрамом Рэфалом в 1968 году в статье «A Formal Basis for the Heuristic Determination of Minimum Cost Paths» [2]. Алгоритм A* применяется в таких областях, как робототехника (планирование маршрута движения), игровые движки (поиск пути для персонажей), логистика (оптимизация маршрутов) и другие задачи поиска на графах и картах.

Главное отличие A* от алгоритма Дейкстры заключается в использовании эвристической функции, которая позволяет алгоритму выбирать наиболее перспективные вершины для обхода.

Эвристическая функция (часто обозначается как h(n)) — это оценка того, насколько "далеко" находится текущая вершина n от целевой вершины. Она помогает алгоритму A* предсказать, какая вершина быстрее приведет нас к цели, и поэтому позволяет ускорить поиск по сравнению с алгоритмом Дейкстры.

Пример эвристической функции:

Манхэттенское расстояние — это сумма по модулю разностей координат.

$$h(n) = |x_n - x| + |y_n - y|$$

Основная идея:

A* использует функцию $f(n) = g(n) + h(n)$, где:

$g(n)$ — стоимость пути от начальной вершины до вершины n,

$h(n)$ — эвристическая оценка оставшегося расстояния от n до целевой вершины.

На каждом шаге выбирается вершина с минимальным значением f(n).

На вход алгоритму подается граф G(V, E), начальная вершина s и целевая вершина t. Каждое ребро имеет неотрицательный вес. Эвристическая функция h(n) должна быть допустимой (не переоценивать реальную стоимость пути).

Алгоритм:

1. Каждой вершине присваивается значение g = ∞, кроме начальной вершины, у которой g = 0.
2. Каждой вершине вычисляется значение f = g + h.
3. Создается приоритетная очередь (открытый список), содержащая вершины для рассмотрения.
4. Пока открытый список не пуст:

– Извлекается вершина u с минимальным f(u).

– Если u — целевая вершина, найден кратчайший путь.

– Для всех соседей u обновляются значения g и f, если найден более короткий путь.

Эксперимент:

Для исследования эффективности алгоритма A* был проведён эксперимент, в котором измерялось время выполнения алгоритма при поиске кратчайшего пути на квадратной сетке различного размера.

Алгоритм запускался на пустых сетках без препятствий с размерами от 25×25 до 500×500 с шагом 25, всего было выполнено 20 запусков. Измерения производились с помощью высокоточной библиотеки <chrono>, результаты сохранялись в микросекундах, что позволило детально проанализировать время выполнения при небольших размерах входных данных.

Алгоритм A* в худшем случае имеет временную сложность O(E), аналогично алгоритму Дейкстры (если эвристика равна нулю). Однако благодаря эвристике, количество рассмотренных вершин часто значительно меньше, особенно на больших графах. На экспериментальных данных время работы росло примерно квадратично относительно размера сетки, но с лучшей практической производительностью по сравнению с обычным алгоритмом Дейкстры.

График зависимости времени выполнения алгоритма от размера сетки приведён на рисунке 3.

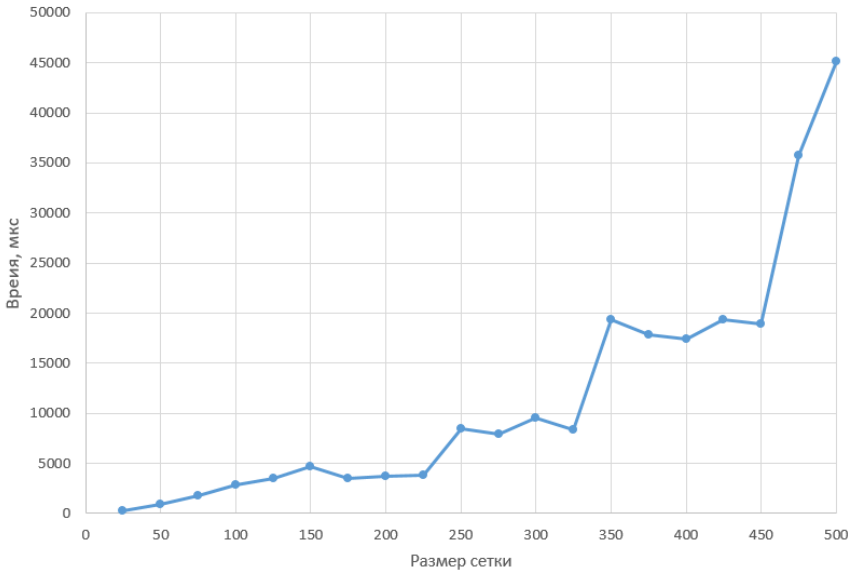

Рисунок 3 – График зависимости времени выполнения алгоритма от размера сетки

Алгоритм Ли

Алгоритм Ли (или волновой алгоритм) представляет собой один из классических методов поиска кратчайшего пути на двумерной дискретной сетке. Он был предложен Ченг-Ю Ли (Cheng Y. Lee) в 1961 году в статье *"An algorithm for path connections and its applications"* [3]. Алгоритм особенно широко применяется в задачах автоматической разводки печатных плат и в робототехнике, где требуется построение пути на сетке с учетом ограничений. Применяется в различных областях, включая: автоматизированное проектирование (CAD-системы), робототехнику (поиск пути по сетке), игровую индустрию (навигация персонажей), планирование маршрутов в ограниченном пространстве (например, склады, лабиринты), а также в системах искусственного интеллекта, где важно найти кратчайший путь без эвристик.

Алгоритм Ли основан на методе обхода в ширину (BFS) и гарантированно находит кратчайший путь, если он существует. В отличие от жадных или эвристических подходов (например, A*), он не использует оценки расстояний до цели, что делает его поведение предсказуемым и равномерным, особенно в средах без препятствий.

На вход алгоритм получает двумерное поле из клеток размером $n \times n$. Начальная клетка обозначается как $(0,0)$, целевая — как $(n-1,n-1)$. Каждая клетка может быть либо проходимой, либо непроходимой (в данном эксперименте все клетки проходимы). Расстояние между соседними клетками считается равным единице, и разрешено двигаться только по четырем направлениям: вверх, вниз, влево и вправо.

Алгоритм:

1. Каждая клетка получает метку — расстояние от начальной клетки.
2. Начальная клетка получает метку 0.
3. Далее от нее «волной» распространяются метки по соседним клеткам, увеличиваясь на единицу на каждом шаге.

4. Распространение продолжается до тех пор, пока не будет достигнута целевая клетка или пока не будут обработаны все возможные клетки.

Эксперимент:

Для исследования временной сложности алгоритма Ли были проведены эксперименты на двумерных полях различного размера. Все клетки считались проходимыми, чтобы исключить влияние случайных препятствий и обеспечить равные условия для сравнения с другими алгоритмами (Дейкстры, A*). Измерения времени производились с использованием стандартной библиотеки <chrono>. Размеры сетки варьировались от 50×50 до 1000×1000 с шагом 50.

Время выполнения алгоритма Ли в первую очередь зависит от количества ячеек, которые он должен посетить. Поскольку он выполняет полный обход сетки от начальной клетки, асимптотическая сложность составляет $O(n^2)$, где n — размер стороны квадрата. В отличие от алгоритма Дейкстры, здесь нет необходимости хранить и обновлять веса путей, что делает реализацию проще и менее затратной по памяти.

График зависимости времени выполнения алгоритма от размера сетки приведён на рисунке 4.

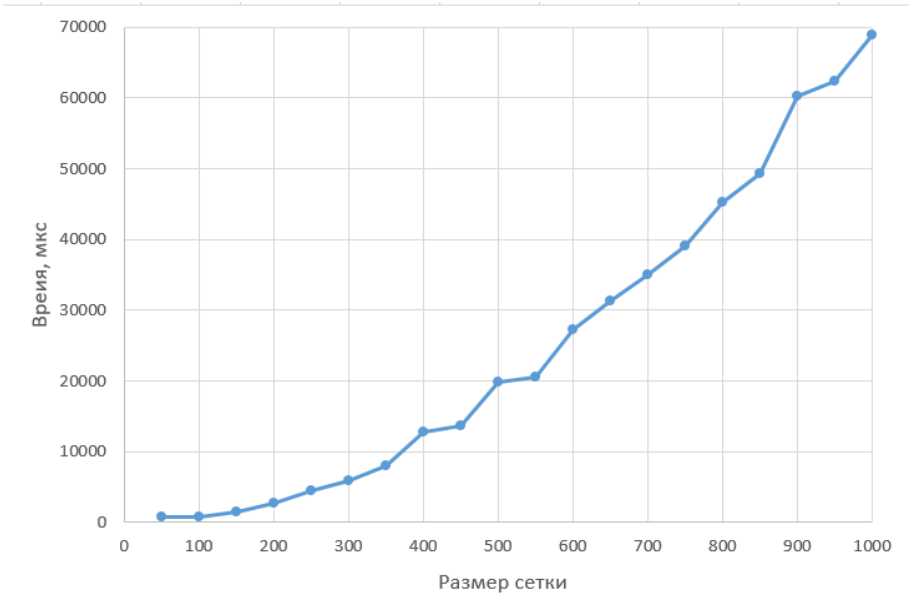

Рисунок 4 – График зависимости времени выполнения алгоритма от размера сетки

Заключение

В рамках данной курсовой работы были рассмотрены и сравнены три алгоритма поиска кратчайшего пути: алгоритм Дейкстры, алгоритм A* и волновой алгоритм Ли. Каждый из них имеет свои особенности, преимущества и ограничения, что определяет их применимость в различных задачах.

Алгоритм Дейкстры является универсальным и надёжным методом для поиска кратчайших путей во взвешенных графах с неотрицательными весами. Он не использует эвристик, что делает его пригодным для задач, где полная информация о графе доступна, и необходим точный результат для всех вершин. Однако его время работы может быть высоким на больших и плотных графах, особенно при использовании матрицы смежности.

Алгоритм A* представляет собой эвристическую модификацию алгоритма Дейкстры. Он активно использует функцию оценки (эвристику), что позволяет значительно ускорить поиск в случаях, когда известна цель и есть приближённое понимание расстояния до неё. При правильно подобранной эвристике A* работает значительно быстрее, особенно в навигационных задачах, где нужно найти путь между двумя точками на сетке или карте. Однако точность и эффективность зависят от выбора эвристики, и он не подходит для поиска путей до всех вершин одновременно.

Алгоритм Ли — это реализация поиска в ширину на двумерной сетке, применяемый в основном в задачах маршрутизации, обработки изображений, робототехники. Он гарантирует нахождение кратчайшего пути в условиях отсутствия весов, при этом не требует эвристики и работает устойчиво на сетках с препятствиями. Тем не менее, он не масштабируется хорошо на больших пространствах из-за полного обхода области.

Список литературы

1. Дейкстра Э. В. Заметка о двух задачах, связанных с графами // *Numerische Mathematik*. – 1959. – Т. 1, № 1. – С. 269–271. – DOI: 10.1007/BF01386390.
2. Харт П. Э., Нильссон Н. Дж., Рафаэль Б. Формальные основы эвристического определения пути минимальной стоимости // *IEEE Transactions on Systems Science and Cybernetics*. – 1968. – Т. 4, № 2. – С. 100–107.
3. Ли Ч. Ю. Алгоритм соединения путей и его применение // *IRE Transactions on Electronic Computers*. – 1961. – Т. EC-10, № 3. – С. 346–365.
4. Чернышев Б. А., Шевченко А. А. Методы поиска кратчайших путей в графах: сравнительный анализ // *Информационные технологии и телекоммуникации*. – 2017. – № 1. – С. 45–50.
5. Кормен Т. Х., Лейзерсон Ч. Э., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. – 3-е изд. – М.: Вильямс, 2016. – 1296 с.

Приложение

Алгоритм Дейкстры:

```
#include <limits.h>
#include <iostream>

constexpr auto number_of_vertices = 5;

int graph[number_of_vertices][number_of_vertices];

int minDistance(int distance[], bool vertex_checked[]) {
    int min = INT_MAX;
    int min_index = -1;

    for (int v = 0; v < number_of_vertices; v++) {
        if (!vertex_checked[v] && distance[v] <= min) {
            min = distance[v];
            min_index = v;
        }
    }
    return min_index;
}

void printSolution(int distance[], int start_vertex, int
parent[]) {
    std::cout << "Вершина      Расстояние      Путь\n";
    for (int i = 0; i < number_of_vertices; i++) {
        if (distance[i] == INT_MAX) {
                std::cout << "   " << (i + 1) << "\t\t" <<
"недостижимо\n";
            continue;
        }

         std::cout << "   " << (i + 1) << "\t\t" << distance[i]
<< "\t    ";

        int path[number_of_vertices];
        int count = 0;
        int temp = i;

        while (temp != start_vertex) {
```

```
            path[count++] = temp;
            temp = parent[temp];
        }

        std::cout << (start_vertex + 1);
        for (int j = count - 1; j >= 0; j--) {
            std::cout << "->" << (path[j] + 1);
        }
        std::cout << "\n";
    }
}

void dijkstra(int start_vertex) {
    int distance[number_of_vertices];
    bool vertex_checked[number_of_vertices];
    int parent[number_of_vertices];

    for (int i = 0; i < number_of_vertices; i++) {
        distance[i] = INT_MAX;
        vertex_checked[i] = false;
        parent[i] = -1;
    }

    distance[start_vertex] = 0;
    parent[start_vertex] = start_vertex;

    for (int count = 0; count < number_of_vertices - 1; count++)
{
        int u = minDistance(distance, vertex_checked);
        if (u == -1) break;
        vertex_checked[u] = true;

        for (int v = 0; v < number_of_vertices; v++) {
           if (!vertex_checked[v] && graph[u][v] && distance[u]
!= INT_MAX
                && distance[u] + graph[u][v] < distance[v]) {
                distance[v] = distance[u] + graph[u][v];
                parent[v] = u;
            }
        }
    }

    printSolution(distance, start_vertex, parent);
}
```

Алгоритм A*:

```
#include <iostream>
#include <queue>
```

```
#include <chrono>
#include <cmath>

const int MAX_SIZE = 1000;

struct Cell {
    int x, y;
    int g, f;
    bool operator>(const Cell& other) const {
        return f > other.f;
    }
};

int heuristic(int x1, int y1, int x2, int y2) {
    return abs(x1 - x2) + abs(y1 - y2);
}

bool isValid(int x, int y, int size) {
    return x >= 0 && x < size && y >= 0 && y < size;
}

void astar(int size) {
    bool visited[MAX_SIZE][MAX_SIZE] = { false };
              std::priority_queue<Cell,      std::vector<Cell>,
std::greater<Cell>> open;

    Cell start;
    start.x = 0;
    start.y = 0;
    start.g = 0;
    start.f = heuristic(0, 0, size - 1, size - 1);
    open.push(start);

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

    while (!open.empty()) {
        Cell current = open.top();
        open.pop();

        int x = current.x, y = current.y;
        if (visited[x][y]) continue;
        visited[x][y] = true;

        if (x == size - 1 && y == size - 1) return;

        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (isValid(nx, ny, size) && !visited[nx][ny]) {
                Cell next;
```

```
                next.x = nx;
                next.y = ny;
                next.g = current.g + 1;
                 next.f = next.g + heuristic(nx, ny, size - 1,
size - 1);
                open.push(next);
            }
        }
    }
}
```

Алгоритм Ли:

```
#include <iostream>
#include <queue>
#include <chrono>
#include <cmath>

const int MAX_SIZE = 1000;

struct Cell {
    int x, y;
    int g;
    int f;

    bool operator>(const Cell& other) const {
        return f > other.f;
    }
};

int heuristic(int x1, int y1, int x2, int y2) {
    return abs(x1 - x2) + abs(y1 - y2);
}

bool isValid(int x, int y, int size) {
    return x >= 0 && x < size && y >= 0 && y < size;
}

void astar(int size) {
    bool visited[MAX_SIZE][MAX_SIZE] = { false };

             std::priority_queue<Cell,      std::vector<Cell>,
std::greater<Cell>> open;

    Cell start;
    start.x = 0;
    start.y = 0;
    start.g = 0;
    start.f = heuristic(0, 0, size - 1, size - 1);
    open.push(start);

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

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

    while (!open.empty()) {
        Cell current = open.top();
        open.pop();

        int x = current.x;
        int y = current.y;

        if (visited[x][y]) continue;
        visited[x][y] = true;

        if (x == size - 1 && y == size - 1) return;

        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if (isValid(nx, ny, size) && !visited[nx][ny]) {
                Cell next;
                next.x = nx;
                next.y = ny;
                next.g = current.g + 1;
                 next.f = next.g + heuristic(nx, ny, size - 1,
size - 1);
                open.push(next);
            }
        }
    }
}
```