ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

ОТЧЕТ

о выполнении курсового проекта

Эвристический поиск на решётках (алгоритм A*)

по дисциплине

«Дискретный анализ»

Выполнил студент группы М8О-308Б-23:

Ибрагимов Далгат Магомедалиевич

Проверил:

Макаров Н.К.

Москва, 2025

Постановка задачи

Дано клетчатое поле размером $n \times m$ ($1 \le n, m \le 1000$), где свободные клетки обозначаются символом `.`, а препятствия — `#`. Считается, что из клетки можно перейти в одну из четырёх соседних (вверх, вниз, влево, вправо), если соседняя клетка свободна.

Требуется обработать q запросов ($1 \le q \le 200$). Каждый запрос задаётся координатами начальной и конечной клеток (x_1, y_1) и (x_2, y_2) (гарантируется, что обе клетки свободны). Для каждого запроса необходимо вывести длину кратчайшего пути между указанными клетками. Если пути не существует, вывести `-1`.

Дополнительно требуется реализовать поиск кратчайшего пути методом эвристического поиска A* (A-star) на графе решётки.

Цель работы

Освоить и реализовать эвристический алгоритм A* для поиска кратчайшего пути в графах, построенных на решётке, с корректной оценкой эвристики, а также обеспечить высокую производительность на больших размерах поля за счёт оптимизаций по времени и памяти.

Идея решения

Графовая модель

Каждая свободная клетка решётки рассматривается как вершина графа. Рёбра соединяют пары соседних по стороне свободных клеток, вес каждого ребра равен 1. Тогда задача поиска длины кратчайшего пути сводится к поиску расстояния в неориентированном невзвешенном графе.

Алгоритм A*

Алгоритм A* является модификацией алгоритма Дейкстры, использующей эвристику для направления поиска к цели.

Для вершины v:

- $g(v)$ — длина пути от старта до v (уже пройденная стоимость);
- $h(v)$ — эвристическая оценка расстояния от v до цели;
- $f(v) = g(v) + h(v)$ — приоритет вершины в очереди открытых вершин (open set).

В данной задаче используется манхэттенская эвристика:

$$h(x, y) = |x - x_2| + |y - y_2|.$$

Для четырёхсвязной решётки она является допустимой (не завышает реальное расстояние) и консистентной, что обеспечивает оптимальность результата A*.

Ускорение A* без приоритетной очереди

Классическая реализация A* использует приоритетную очередь по f. В данной реализации применяется специализированная оптимизация, основанная на свойстве манхэттенской эвристики на решётке:

при переходе к соседу g увеличивается на 1, а h меняется на ± 1, поэтому

$$f' = (g+1) + (h \pm 1) = f + 0 \quad \text{или} \quad f + 2.$$

Следовательно, значения f достижимых вершин изменяются только на 0 или 2, что позволяет заменить приоритетную очередь на две "корзины":

- `cur` — вершины с текущим значением $f =$ `curF`;
- `nxt` — вершины со следующим значением $f =$ `curF` $+ 2$.

После исчерпания `cur` производится переход к следующему слою: `curF += 2`, `cur` и `nxt` меняются местами.

Описание реализации

Реализация состоит из этапа чтения поля и обработки запросов. Для каждого запроса выполняется запуск A* с манхэттенской эвристикой и ускорением через два списка `cur/nxt`. Если достижимого пути нет, поиск исчерпывает все достижимые клетки из старта и возвращает `-1`.

Представление клеток

Клетка (x, y) кодируется одним индексом:

$$id = x \cdot m + y,$$

где используется нулевая индексация.

Маркировка посещений без очистки массивов

Поле может содержать до 10^6 клеток, поэтому очистка массивов на каждый запрос неэффективна. Используется техника "временных меток":

- `stamp` увеличивается на 1 для каждого запроса;
- `used[v] == stamp` означает, что вершина v была затронута в текущем запросе;
- `closed[v] == stamp` означает, что вершина окончательно обработана (закрыта) в текущем запросе.

Листинг программы (C++)

Вход: n, m, решётка, число запросов q, затем q строк с координатами.

Выход: длина кратчайшего пути для каждого запроса либо -1.

```
#include <bits/stdc++.h>
using namespace std;

struct FastScanner {
    static const int BUFSIZE = 1 << 20;
    int idx = 0, size = 0;
    char buf[BUFSIZE];

    inline char readChar() {
        if (idx >= size) {
            size = (int)fread(buf, 1, BUFSIZE, stdin);
            idx = 0;
            if (!size) return 0;
        }
        return buf[idx++];
    }

    template <class T>
    bool readInt(T &out) {
        char c;
        do {
            c = readChar();
            if (!c) return false;
        } while (c <= ' ');

        T sign = 1;
        if (c == '-') { sign = -1; c = readChar(); }

        T val = 0;
        while (c > ' ') {
            val = val * 10 + (c - '0');
            c = readChar();
        }
        out = val * sign;
        return true;
    }

    bool readString(string &s) {
        char c;
```

```
        do {
            c = readChar();
            if (!c) return false;
        } while (c <= ' ');
        s.clear();
        while (c > ' ') {
            s.push_back(c);
            c = readChar();
        }
        return true;
    }
};

static inline bool inBounds(int x, int y, int n, int m) {
    return (unsigned)x < (unsigned)n && (unsigned)y < (unsigned)m;
}

struct GridData {
    int n = 0, m = 0;
    vector<uint8_t> freeCell;
};

GridData readGrid(FastScanner &fs) {
    GridData G;
    fs.readInt(G.n);
    fs.readInt(G.m);

    vector<string> grid(G.n);
    for (int i = 0; i < G.n; i++) fs.readString(grid[i]);

    const int N = G.n * G.m;
    G.freeCell.assign(N, 0);
    for (int i = 0; i < G.n; i++) {
        for (int j = 0; j < G.m; j++) {
            G.freeCell[i * G.m + j] = (grid[i][j] == '.');
        }
    }
    return G;
}

int astarDistance(
    const GridData &G,
    int x1, int y1, int x2, int y2,
```

```
    vector<int> &dist, vector<int> &used, vector<int> &closed,
    int &stamp,
    vector<int> &cur, vector<int> &nxt
) {
    const int dx[4] = {1, -1, 0, 0};
    const int dy[4] = {0, 0, 1, -1};

    if (!inBounds(x1, y1, G.n, G.m) || !inBounds(x2, y2, G.n, G.m))
       return -1;

    int s = x1 * G.m + y1;
    int t = x2 * G.m + y2;

    if (!G.freeCell[s] || !G.freeCell[t]) return -1;
    if (s == t) return 0;

    ++stamp;

    auto H = [&](int x, int y) -> int {
        return abs(x - x2) + abs(y - y2);
    };

    cur.clear();
    nxt.clear();
    cur.reserve(1024);
    nxt.reserve(1024);

    used[s] = stamp;
    dist[s] = 0;

    int curF = H(x1, y1);
    cur.push_back(s);

    size_t headCur = 0;

    while (true) {
        while (headCur < cur.size()) {
            int v = cur[headCur++];

            if (used[v] != stamp) continue;
            if (closed[v] == stamp) continue;

            int x = v / G.m, y = v % G.m;
```

```
                int g = dist[v];

                if (g + H(x, y) != curF) continue;

                if (v == t) return g;

                closed[v] = stamp;

                for (int k = 0; k < 4; k++) {
                    int nx = x + dx[k], ny = y + dy[k];
                    if (!inBounds(nx, ny, G.n, G.m)) continue;

                    int to = nx * G.m + ny;
                    if (!G.freeCell[to]) continue;
                    if (closed[to] == stamp) continue;

                    int ng = g + 1;
                    if (used[to] != stamp || ng < dist[to]) {
                        used[to] = stamp;
                        dist[to] = ng;

                        int nf = ng + H(nx, ny);
                        if (nf == curF) cur.push_back(to);
                        else nxt.push_back(to);
                    }
                }
            }

            if (nxt.empty()) break;

            cur.swap(nxt);
            nxt.clear();
            headCur = 0;
            curF += 2;
        }

        return -1;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

```

```
    FastScanner fs;

    GridData G = readGrid(fs);

    const int N = G.n * G.m;
    vector<int> dist(N, 0), used(N, 0), closed(N, 0);
    int stamp = 0;

    vector<int> cur, nxt;

    int q;
    fs.readInt(q);

    while (q--) {
        int x1, y1, x2, y2;
        fs.readInt(x1); fs.readInt(y1); fs.readInt(x2); fs.readInt(y2);
        --x1; --y1; --x2; --y2;

        int ans = astarDistance(G, x1, y1, x2, y2, dist, used, closed,
            stamp, cur, nxt);
        cout << ans << "\n";
    }

    return 0;
}
```

Доказательство корректности (эскиз)

Корректность A* с манхэттенской эвристикой

Манхэттенская эвристика $h(x, y) = |x - x_2| + |y - y_2|$ не превосходит реального кратчайшего расстояния на четырёхсвязной решётке (допустимость) и является консистентной: для любого ребра (u, v) выполняется $h(u) \leq 1 + h(v)$. Поэтому алгоритм A* при извлечении вершин по неубыванию $f = g + h$ гарантированно находит оптимальный путь: при первом достижении цели полученное значение $g(t)$ равно длине кратчайшего пути.

Если цель недостижима, то при исчерпании множества открытых вершин алгоритм обработает все вершины, достижимые из старта по свободным клеткам, и корректно вернёт `-1`.

Корректность "двухкорзинной" обработки по слоям f

В данной задаче вес ребра равен 1, а при переходе к соседу h меняется на ± 1, следовательно f меняется на 0 или 2. Это означает, что при фиксированном текущем значении `curF` все новые вершины могут попасть только в слой `curF` или `curF+2`. Таким образом, последовательная обработка слоёв `curF`, `curF+2`, `curF+4`, …эквивалентна извлечению из очереди открытых вершин по возрастанию f, что сохраняет стандартную корректность A*.

Оценка сложности

Пусть $N = n \cdot m$.

- Чтение и сохранение поля: $O(N)$ по времени и $O(N)$ по памяти.
- На один запрос A*: время $O(K)$, где K — число обработанных (раскрытых) вершин в этом запросе; в худшем случае $K = O(N)$.
- Память для A*: массивы `dist/used/closed` размера N — $O(N)$.

С учётом ограничения $q \le 200$ общая асимптотика:

$$O(N) + \sum_{i=1}^{q} O(K_i), \qquad \text{память } O(N).$$

На практике эвристика сокращает область поиска по сравнению с BFS, а использование временных меток исключает затратную очистку массивов между запросами.

Тестирование (примеры)

1. **Пример 1.**
   Вход:

```
5 5
#....
.#.#.
.#.##
.#...
...##
5
2 1 1 2
1 2 2 1
3 1 2 5
4 5 2 1
4 5 1 4
```

   Выход:

```
10
10
11
```

```
8
6
```

2. **Пример 2.**
   Вход:

   ```
   3 3
   ...
   ###
   ...
   4
   1 1 1 3
   1 1 3 3
   1 1 3 1
   3 1 3 3
   ```

   Выход:

   ```
   2
   -1
   -1
   2
   ```

3. **Граничные случаи.**
   - Старт совпадает с финишем: ответ 0.
   - Поле без препятствий: путь равен манхэттенскому расстоянию.
   - Отсутствие пути из-за препятствий: ответ `-1`.

Дневник отладки

- Исправлено хранение отметок посещения: вместо очистки массивов на N элементов используется счётчик `stamp`.
- Проверено свойство изменения f только на 0 или 2 для манхэттенской эвристики на четырёхсвязной решётке, что позволило убрать приоритетную очередь.
- Проверены индексации координат: вход 1-based, внутри программы используется 0-based.

Выводы

В рамках курсового проекта реализован эвристический алгоритм A* для поиска кратчайшего пути на решётке с препятствиями и множеством запросов. Для ускорения работы применены:

- оптимизация A* за счёт обработки слоёв f двумя списками (`cur/nxt`) вместо приоритетной очереди;

- техника временных меток для исключения очистки больших массивов между запросами.

Реализация корректно выводит длину кратчайшего пути либо `-1` при отсутствии пути и соответствует ограничениям по времени и памяти.