Оглавление

1. Граф

Граф - это абстрактный математический объект, представляющий собой множество вершин и рёбер, соединяющих пары этих вершин.

Графы разделяют на ориентированные и неориентированные. *Ориентированный граф* — граф, рёбрам которого присвоено направление. Направленные рёбра именуются дугами. Граф, ни одному ребру которого не присвоено направление, называется *неориентированным графом*.

Также рёбрам графа могут быть присвоены «весы», то есть некоторые значения, определяющие стоимость прохождения из одной вершины в другую. Граф, каждому ребру которого поставлено в соответствие некоторое значение, называется *взвешенным*. Обычно вес ребра — это вещественное число (в таком случае его можно интерпретировать как «длину» ребра).

Графы удобно использовать для хранения карт. Так, в случае здания с несколькими этажами точки карты на одном этаже и точки перехода между этажами (лестницы) образуют множество вершин графа, а пути между ними - множество ребер. Карта не идентифицируется как игровая, поэтому не нужны «одноразовые пути», следовательно, граф будет неориентированным. Для оптимального поиска пути, учитывающего различные факторы перемещения (например, приоритет лифта над лестницей) необходимо использовать веса.

Вывод: используемый граф будет взвешенным неориентированным.

1.1. Реализации хранения

Программно хранение графа может быть реализовано несколькими способами:

1. В виде массива списков смежности («CListGraph»)

Список смежности вершины - набор вершин, в которые существуют пути (ребра) из заданной вершины.

Связи между вершинами хранятся в виде двумерного массива:

```
std::vector<std::vector<int>> out_edges;
```

Ребро добавляется по номеру начальной вершины:

```
void CListGraph::add_edge(int from, int to) {
    assert(from >= 0 && from < vertices_count());
    assert(to >= 0 && to < vertices_count());

    out_edges[from].push_back(to);
}
```

Получение смежных вершин:

```
std::vector<int> CListGraph::
get_next_vertices(int vertex) const {
    assert(vertex >= 0 &&
           vertex < vertices_count()
    );

    return out_edges[vertex];
}
```

2. В виде матрицы смежности («CMatrixGraph»)

Матрица смежности - квадратная булева матрица. В качестве номеров строк и столбцов матрицы выступают вершины графа, в качестве ее элементов - нули или единицы. Так, если существует проход из i-ой вершины в j-ую, значение в ячейке (i; j) равно единице, иначе - нулю.

Связи между вершинами хранятся в виде булевой матрицы, то есть двумерного массива (вектора) нулей и единиц:

```
std::vector<std::vector<bool>> matrix;
```

Ребро, то есть путь из одной вершины в другую, добавляется в элемент матрицы с координатами (from, to), где from - номер начальной вершины, to - номер конечной вершины:

```
void CMatrixGraph::add_edge(int from, int to) {
    assert(from >= 0 && from < vertices_count());
    assert(to >= 0 && to < vertices_count());

    matrix[from][to] = true;
}
```

Получение смежных вершин осуществляется путем поиска номеров значений, принимающих единицу, в строке матрицы с номером заданной вершины:

```
std::vector<int> CMatrixGraph::
get_next_vertices(int vertex) const {
    assert(vertex >= 0 &&
           vertex < vertices_count()
    );

    std::vector<int> next_vertices;
    for (size_t vert = 0;
         vert < vertices_count(); ++vert) {
        if (matrix[vertex][vert]) {
            next_vertices.push_back(
                static_cast<int &&>(vert)
            );
        }
    }

    return next_vertices;
}
```

3. В виде массива хэш-таблиц («CSetGraph»)

Хэш-таблица - структура, хранящая пары ключей и значений, где ключ представлен некоторой хэш-функцией. В данном случае используется массив таких структур, где номер элемента (хэш-таблицы) представляет собой номер вершины графа, а сам элемент (хэш-таблица) - набор ребер для этой вершины.

Связи между вершинами хранятся в виде массива неотсортированных хэш-таблиц:

```
std::vector<std::unordered_set<int>> vertices;
```

Ребро добавляется в хэш-таблицу, лежащую по номеру начальной вершины:

```
void CSetGraph::add_edge(int from, int to) {
    assert(from >= 0 && from < vertices_count());
    assert(to >= 0 && to < vertices_count());

    vertices[from].insert(to);
}
```

Получение смежных вершин осуществляется путем поиска массива, состоящего из хэш-таблиц, с номером заданной вершины:

```
std::vector<int> CSetGraph::
get_next_vertices(int vertex) const {
    assert(vertex >= 0 &&
           vertex < vertices_count()
    );

    return
        std::vector<int>(
            vertices[vertex].begin(),
            vertices[vertex].end()
        );
}
```

4. В виде одного массива пар («CArcGraph»)

В данном случае пара представляет собой структуру, содержащую начальную и конечную вершины ребра графа. Набор всех таких структур представляет собой множество ребер графа.

Связи между вершинами хранятся в виде массива пар номеров вершин:

```
std::vector<std::pair<int, int>> vertices_mas;
```

Ребро добавляется в конец массива:

```
void CArcGraph::add_edge(int from, int to) {
    assert(from >= 0 && from < vertices_count());
    assert(to >= 0 && to < vertices_count());

    vertices_mas.emplace_back(from, to);
}
```

Получение смежных вершин представляет собой поиск массива пар, где первому значению пары (то есть начальной точке ребра графа) соответствует номер заданной вершины:

```
std::vector<int> CArcGraph::
get_next_vertices(int vertex) const {
    assert(vertex >= 0 &&
           vertex < vertices_count()
    );

    std::vector<int> next_vertices;

    for (auto i : vertices_mas) {
        if (i.first == vertex)
            next_vertices.push_back(i.second);
    }
    return next_vertices;
}
```

Далее будут рассмотрены результаты тестирования всех способов.

Исходный граф генерируется программно по заданным количеству вершин и плотности (отношению количества вершин к количеству ребер).

Проверяются следующие методы интерфейса реализаций графа:

- добавление вершин и связей между ними
- получение вершин, доступных из текущей

Получение тестируется от трех вершин: начальной, находящейся посередине и конечной. Каждый тест проходит 1000 раз. По результатам высчитываются время прохождения теста и средний показатель «RPS» («requests per second» - количество запросов в секунду).

1. Результаты для разреженного графа с количеством вершин 1000 и количеством ребер 100000:

```
graph_size = 1000
Generation
        time: 2020 ms, (2020621 µs)
edges_count = 100000
```

Рис. 1. *Входные данные теста №1*

```
CListGraph
    CListGraph adding edges:
        time: 66 ms, (66837 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 500000.000
    get_next_vertices from middle:
        time: 0 ms, (9 µs)
        RPS = 500000.000
    get_next_vertices from back:
        time: 0 ms, (4 µs)
        RPS = 333333.333
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 5 ms, (5536 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (30 µs)
        RPS = 26315.789
    get_next_vertices from middle:
        time: 0 ms, (34 µs)
        RPS = 10638.298
    get_next_vertices from back:
        time: 0 ms, (276 µs)
        RPS = 15151.515
```

Рис. 2. *Результаты теста №1 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 81 ms, (81385 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (15 µs)
        RPS = 250000.000
    get_next_vertices from middle:
        time: 0 ms, (7 µs)
        RPS = 125000.000
    get_next_vertices from back:
        time: 0 ms, (9 µs)
        RPS = 142857.143
```

```
CArcGraph
    CArcGraph adding edges:
        time: 15 ms, (15051 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 4 ms, (4348 µs)
        RPS = 651.890
    get_next_vertices from middle:
        time: 1 ms, (1184 µs)
        RPS = 896.861
    get_next_vertices from back:
        time: 1 ms, (1143 µs)
        RPS = 943.396
```

Рис. 3. *Результаты теста №1 для реализаций «CSetGraph» и «CArcGraph»*

2. Результаты для среднезаполненного графа с количеством вершин 1000 и количеством ребер 600000:

```
graph_size = 1000
Generation
        time: 7387 ms, (7387158 µs)
edges_count = 600000
```

Рис. 4. *Входные данные теста №2*

```
CListGraph
    CListGraph adding edges:
        time: 96 ms, (96678 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 166666.667
    get_next_vertices from middle:
        time: 0 ms, (7 µs)
        RPS = 166666.667
    get_next_vertices from back:
        time: 0 ms, (7 µs)
        RPS = 166666.667
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 20 ms, (20027 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (59 µs)
        RPS = 20000.000
    get_next_vertices from middle:
        time: 0 ms, (66 µs)
        RPS = 18518.519
    get_next_vertices from back:
        time: 0 ms, (66 µs)
        RPS = 16949.153
```

Рис.5. *Результаты теста №2 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 225 ms, (225193 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 5 ms, (5291 µs)
        RPS = 52631.579
    get_next_vertices from middle:
        time: 0 ms, (28 µs)
        RPS = 55555.556
    get_next_vertices from back:
        time: 0 ms, (23 µs)
        RPS = 58823.529
```

```
CArcGraph
    CArcGraph adding edges:
        time: 55 ms, (55633 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 6 ms, (6438 µs)
        RPS = 152.765
    get_next_vertices from middle:
        time: 6 ms, (6374 µs)
        RPS = 156.128
    get_next_vertices from back:
        time: 6 ms, (6140 µs)
        RPS = 158.178
```

Рис.6. *Результаты теста №2 для реализаций «CSetGraph» и «CArcGraph»*

3. Результаты для плотного графа с количеством вершин 1000 и количеством реберх 1000000:

```
graph_size = 1000
Generation
        time: 12355 ms, (12355933 µs)
edges_count = 1000000
```

Рис. 7. *Входные данные теста №3*

```
CListGraph
    CListGraph adding edges:
        time: 160 ms, (160745 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 90909.091
    get_next_vertices from middle:
        time: 0 ms, (26 µs)
        RPS = 83333.333
    get_next_vertices from back:
        time: 0 ms, (12 µs)
        RPS = 83333.333
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 40 ms, (40653 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (74 µs)
        RPS = 15873.016
    get_next_vertices from middle:
        time: 0 ms, (75 µs)
        RPS = 15873.016
    get_next_vertices from back:
        time: 0 ms, (63 µs)
        RPS = 15873.016
```

Рис. 8. Р*езультаты теста №3 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 330 ms, (330421 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (43 µs)
        RPS = 40000.000
    get_next_vertices from middle:
        time: 0 ms, (33 µs)
        RPS = 32258.065
    get_next_vertices from back:
        time: 0 ms, (38 µs)
        RPS = 34482.759
```

```
CArcGraph
    CArcGraph adding edges:
        time: 83 ms, (83110 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 11 ms, (11841 µs)
        RPS = 93.058
    get_next_vertices from middle:
        time: 10 ms, (10362 µs)
        RPS = 90.761
    get_next_vertices from back:
        time: 10 ms, (10402 µs)
        RPS = 91.971
```

Рис.9. *Результаты теста №3 для реализаций «CSetGraph» и «CArcGraph»*

В нашем случае граф будет плотным, поэтому рассмотрим результаты теста №3 (количество вершин - 1000, количество ребер - 1000000):

	CListGraph	CSetGraph	CMatrixGraph	CArcGraph
Заполнение графа, мс	160	330	40	83
Начало, запросы/с	90909	40000	15873	93
Середина, запросы/с	83333	32258	15873	91
Конец, запросы/с	83333	34482	15873	92

Таблица 1. *Характеристики графа в зависимости от реализации*

Анализируя данные результаты, можно выстроить последовательность скорости работы реализаций: 1 - «CListGraph», 2 - «CSetGraph», 3 - «CMatrixGraph», 4 - «CArcGraph». Скорость заполнения не учитывается, т.к. она происходит лишь один раз при инициализации системы. Стоит отметить, что существенная разница заметна только на больших графах с высокой плотностью и на низких объемах выбор реализации не критичен.

2. Алгоритмы поиска

Для выбора алгоритма поиска необходимо определиться, как именно представить карту в виде графа. Существуют два варианта: проставлять точки и соединять их вручную (трехмерный граф), или разделить граф на два (вертикаль и горизонталь). Для поиска оптимален первый вариант. Кроме того, его применяют во многих продуктах, доступных на рынке, которые определяют местоположение пользователя с помощью маячков. Однако такой подход влечет за собой много рутинной работы, связанной с занесением в граф данных, тогда как второму варианту понадобится наименьшее время для заполнения. Также во втором случае вся работа по отрисовке ложится на алгоритм, и из-за этого незначительно увеличивается время поиска.

Для поиска маршрута возможно использование следующих алгоритмов:

- «Поиск в ширину». Такой алгоритм выполняет поиск равномерно во всех направлениях. Это невероятно полезный алгоритм не только для обычного поиска пути, но и для процедурной генерации карт, поиска путей течения, карт расстояний и других типов анализа карт.
- «Алгоритм Дейкстры» (поиск с равномерной стоимостью). Алгоритм Дейкстры позволяет задавать приоритеты исследования путей. Вместо равномерного исследования всех возможных путей он отдаёт предпочтение путям с низкой стоимостью.
- « A*». Данный вид поиска является модификацией алгоритма Дейкстры, оптимизированной для единственной конечной точки. Алгоритм Дейкстры может находить пути ко всем точкам, тогда как A* - к одной точке. Он отдаёт приоритет путям, которые ведут ближе к цели.

2.1. Поиск в ширину

2.1.1.Описание

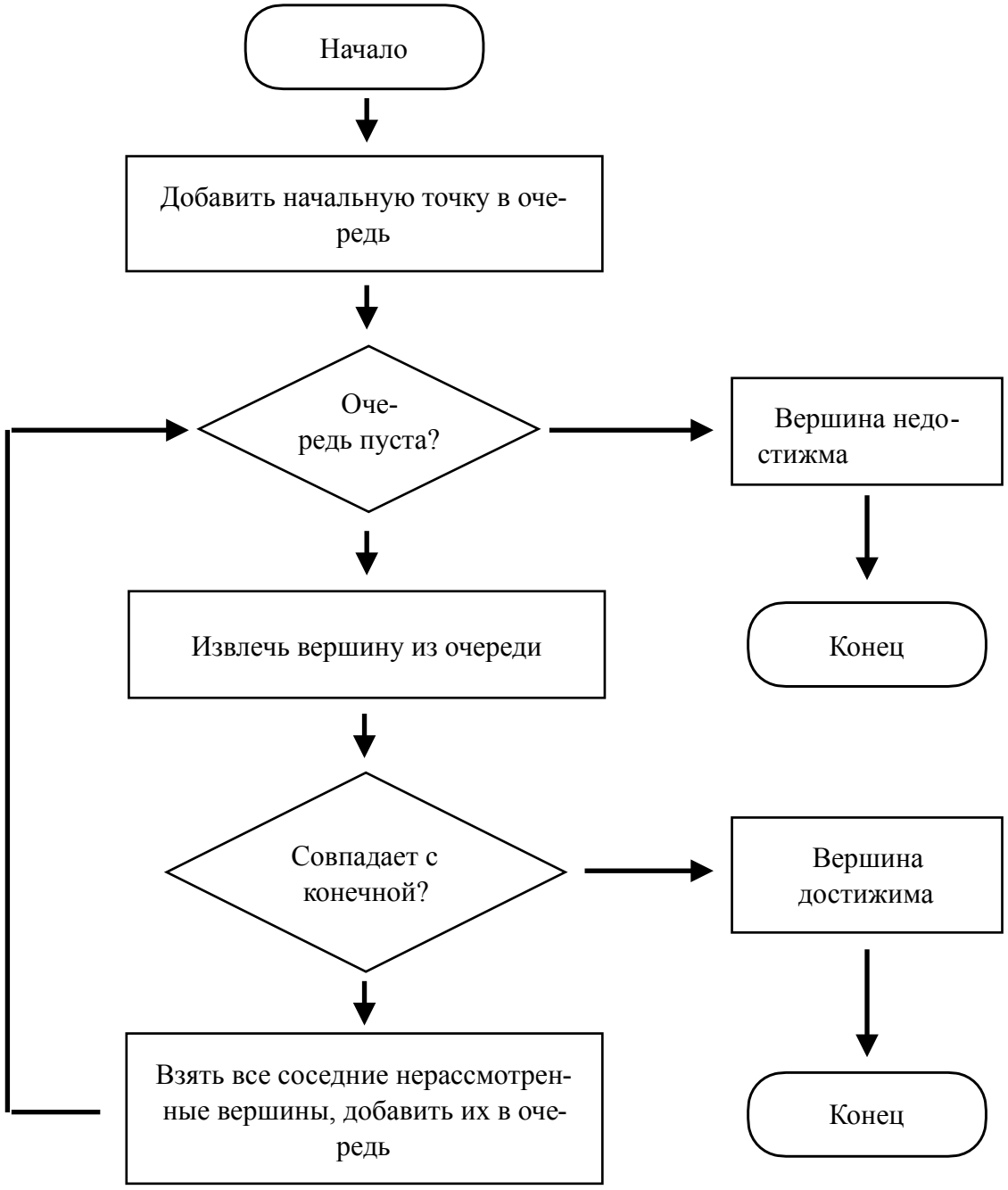

Рис. 10. *Алгоритм поиска в ширину*

Поиск в ширину подходит для поиска маршрута только в невзвешенных графах из-за особенности алгоритма. В случае взвешенного графа найденный им маршрут по числу ребер будет неоптимальным, т.к. веса ребер не учитываются.

2.1.2.Оценка сложности

Так как в памяти хранятся все развёрнутые узлы, пространственная сложность алгоритма составляет $O(|V| + |E|)$, где V, E - число вершин и рёбер в графе соответсвенно.

Сложность алгоритма составляет $O(|V| + |E|)$ для худшего случае, когда будут посещены все узлы графа.

2.2. Алгоритм Дейкстры

2.2.1.Описание

Аналогичен поиску в ширину и является его улучшением, использующим очередь с приоритетом. Подходит для работы со взвешенными графами. Для поиска оптимального пути требуется обойти весь граф, т.к. кратчайшее расстояние определяется весом ребер и первое вхождение в общем случае не является лучшим.

2.2.2.Оценка сложности

Оценка времени работы. Для реализации с двоичной кучей:

- Добавление узла: $O(logV)$. Не более V операций.
- Уменьшение значения ключа: $O(logV)$. Не более E операций.

Всего $T = O((E + V) * logV)$

Используемая память $M = O(V)$

2.3. Алгоритм A*

2.3.1.Описание

Также является улучшенной версией предыдущего алгоритма. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Существуют различные эвристики, использующиеся в различных ситуациях. Например:

- Манхэттенское расстояние для перемещения в 4 направлениях плоскости

$$h(v) = |v.x - goal.x| + |v.y - goal.y| \quad (1)$$

- Расстояние Чебышева применяется, когда к четырем направлениям добавляются диагонали

$$h(v) = max(|v.x - goal.x|, |v.y - goal.y|) \quad (2)$$

- Если передвижение не ограничено сеткой, то можно использовать евклидово расстояние по прямой

$$h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2} \quad (3)$$

В проекте были протестированы разные эвристики, например, для махэттенского расстояния поиск на сетке размером 1280x1080 недалеко стоящих точек занял 575 итераций, при спрямлении пути - 2141, а при евклидовом расстоянии уже 31918. Также усложнение эврестической функции на каждой итерации сильно отразилось на поиске маршрута в худшую сторону.

2.3.2.Оценка сложности

Сложность алгоритма зависит от используемой эвристики. В худшем случае, число вершин, посещенных алгоритмом, растет экспоненциально.

3. Поиск маршрута

В работе используется взвешенный неориентированный граф. Вся карта делится на две плоскости - горизонтальную и вертикальную. Вертикальный граф состоит из графа лестниц и связей между ними, поиск по нему осуществляется с использованием алгоритма Дейкстры. Горизонтальный представляет собой сетку, где каждая вершина - это пиксель, связанный с другими ближайшими. Размер горизонтального графа равен размеру изображения плана этажа и составляет 1280x1080 пикселей. Поиск на этаже осуществляется алгоритмом A* с манхэтаннской эвристикой.

Таким образом, алгоритм поиска состоит из комбинации двух алгоритмов, и позволяет прокладывать маршруты в ограниченных пространствах.

Блок-схема работы алгоритма представлена на рис. 11.

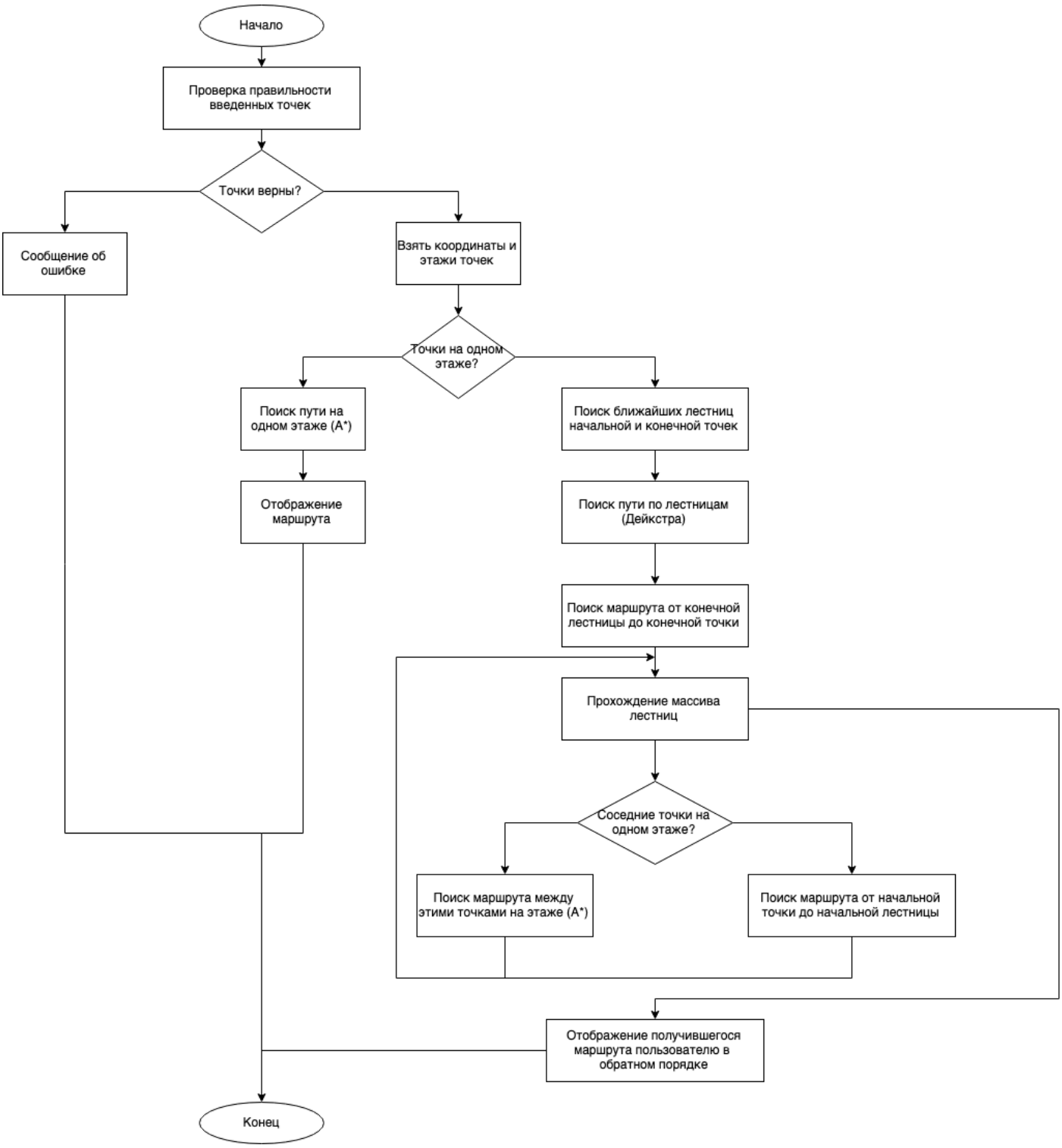

Рис. 11. *Алгоритм поиска пути внутри здания*