Федеральное государственное автономное образовательное учреждение высшего образования

«Санкт-Петербургский политехнический университет Петра Великого»

Высшая инженерная школа

**КУРСОВОЙ ПР ОЕКТ**

**Алгоритмы Дейкстры и A\***

по дисциплине

«Структуры и алгоритмы компьютерной обработки данных»

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр. ПРГ.ИС.2.2. |  | Е.В. Приходько |
| Руководитель  доцент, к.ф.-м.н. |  | В.Г. Пак |

« » 2016

Санкт-Петербург

2016

1 Постановка задачи, решаемой алгоритмами Дейкстры и A\*

Алгоритмы Дейкстры и A\* используются для поиска оптимального пути в взвешенном графе с неотрицательными весами ребер. Алгоритм Дейкстры широко применяется в протоколах маршрутизации при обменах данными по сети, для решения логистических задач и для ориентирования систем искусственного интеллекта в некотором пространстве. Алгоритм A\* похож на алгоритм Дейкстры, но использует эвристическую функцию для определения следующей рассматриваемой вершины, что позволяет уменьшить время поиска.

Рассмотри пример применения алгоритмов поиска пути. Пусть дана карта некоторой местности (рисунок 1), где серым и черным цветами обозначены, соответственно, проходимые и непроходимые клетки.

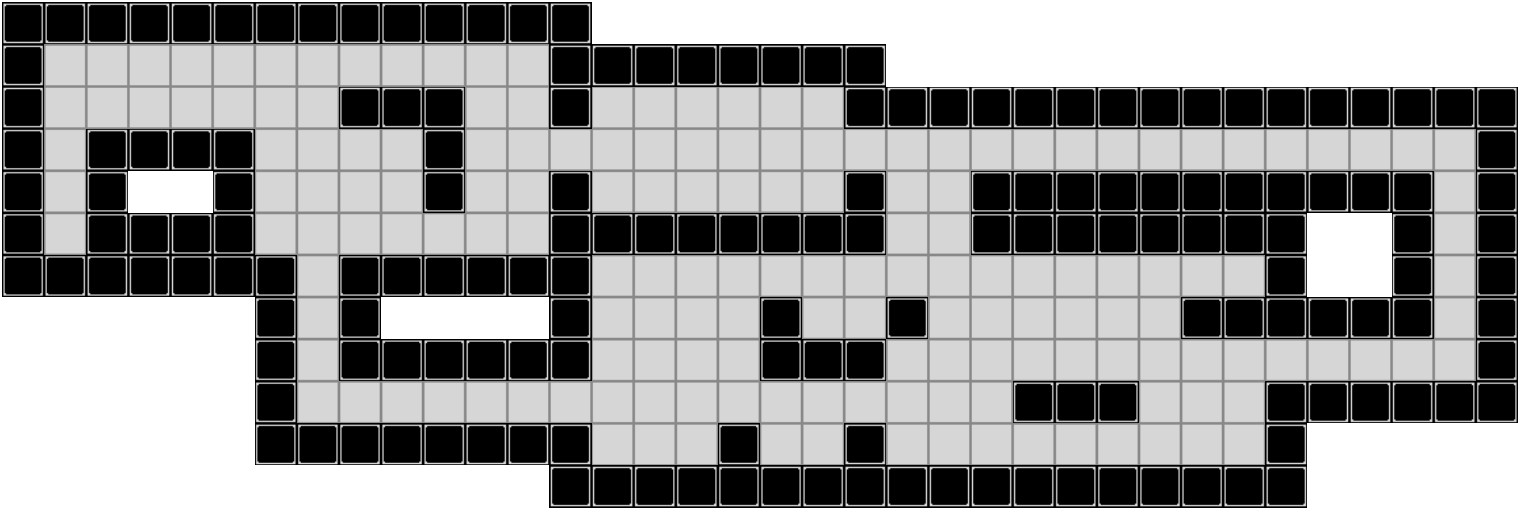


Рисунок 1. Пример топологии местности.

Сгенерировав граф возможных положений и переходов между ними, можно автоматизировать поиск пути из одной точки местности в другую (рисунок 2).

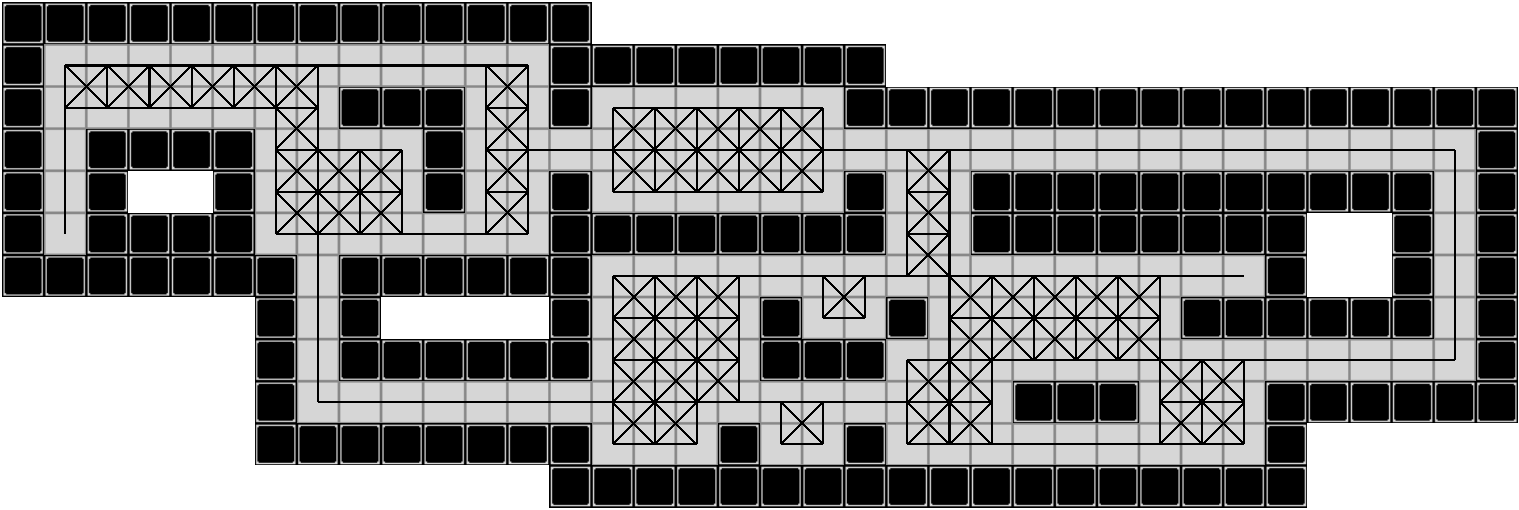


Рисунок 2. Пример местности со сгенерированным графом.

Подобный подход используется при программировании искусственного интеллекта для перемещения по реальной или виртуальной местности. В этом случае граф часто называют навигационной сеткой (navigation mesh, navmesh), пример такой сетки изображен на рисунке 3.



Рисунок 3. Пример навигационной сетки, используемой в видеоиграх.

2 Алгоритм Дейкстры

Алгоритм Дейкстры – алгоритм на графах, изобретенный нидерландским ученым Эдсгером Дейкстрой в 1959 году. В общем случае находит кратчайшие пути от выбранного узла до всех остальных [Кормен].

2.1 Неформальное описание

Рассмотрим пример работы алгоритма на графе, изображенном на рисунке 4. Найдем кратчайшие пути от вершины 1 до всех остальных. Каждой вершине сопоставляется метка со значением, равным длине пути от начальной вершины до нее. Перед началом работы алгоритма метке исходной вершины устанавливается значение ноль, а всем остальным – бесконечность или другое особое значение, обозначающее отсутствие пути (рисунок 5). В последствие в ходе работы алгоритма эти значения могут уменьшаться, что означает, что был найден более короткий путь.

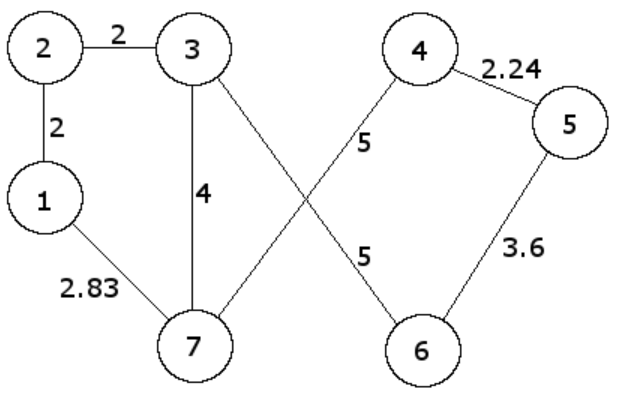


Рисунок 4. Исходный взвешенный граф.

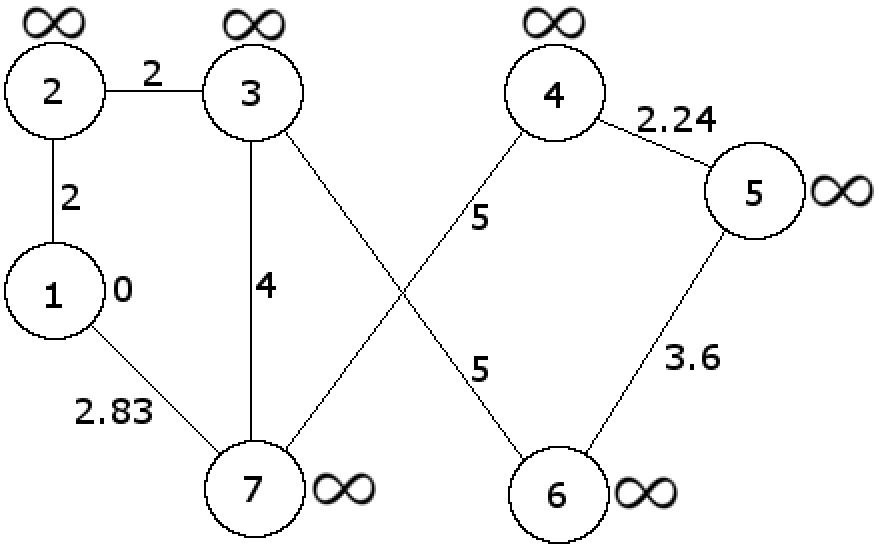


Рисунок 5. Начальное состояние меток.

На каждой итерации алгоритма вершина с минимальной меткой устанавливается в качестве текущей и помечается звездочкой, что означает, что до данной вершины найдено кратчайшее расстояние и уменьшаться она больше не будет. В данном случае выбирается вершина 1, так как ее метка равно нулю, а остальные бесконечности. Затем для всех вершин, смежных с текущей, пересчитывается значение метки: новое значение равно сумме значения текущей вершины и веса ребра между этими вершинами. Если новое значение меньше текущего, то метка обновляется. Это значит, что найден путь к этой вершине, который короче того, что был известен на данный момент.

Метка вершины 2 = min{∞, 0 + 2} = 2

Метка вершины 7 = min{∞, 0 + 2.83} = 2.83

Результат первой итерации представлен на рисунке 6.

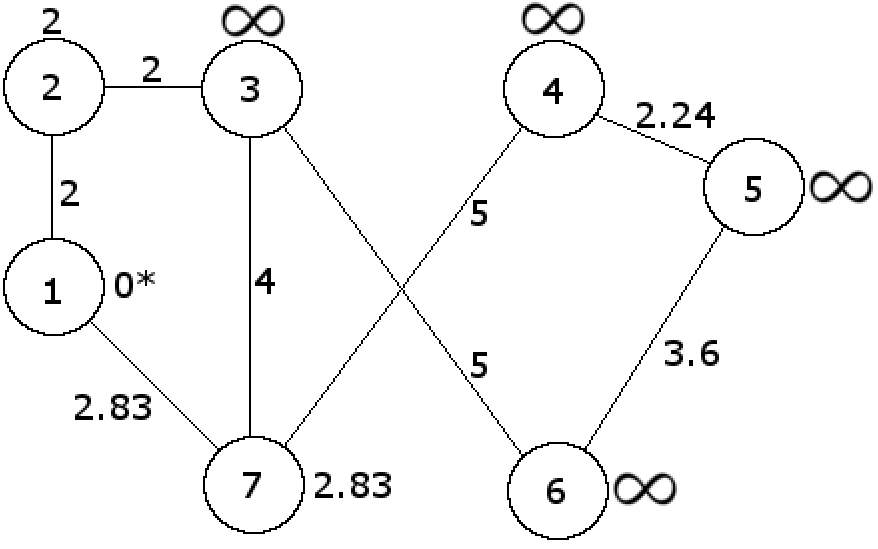


Рисунок 6. Результат первой итерации.

Выбирается новая вершина с минимальным значением метки из тех, что еще не были посещены (т.е. не отмечены звездочкой) и пересчет меток повторяется для вершин, смежных с ней. В данном примере такой вершиной является вершина 2.

Метка вершины 3 = min{∞, 2 + 2} = 4

Результат второй итерации представлен на рисунке 7.

Обход вершин продолжается, либо пока не будут посещены все вершины, либо пока не останутся вершины только со значением метки, равным бесконечности. Второй случай означает, что есть вершины, не достижимые из исходной. Результат обхода вершин из примера представлен на рисунке 8.

Для того, чтобы получить оптимальный путь из исходной вершины к другой, необходимо из целевой вершины идти в сторону начальной, каждый раз выбирая смежную вершину с минимальным значением суммы ее метки и веса ребра до текущей вершины.

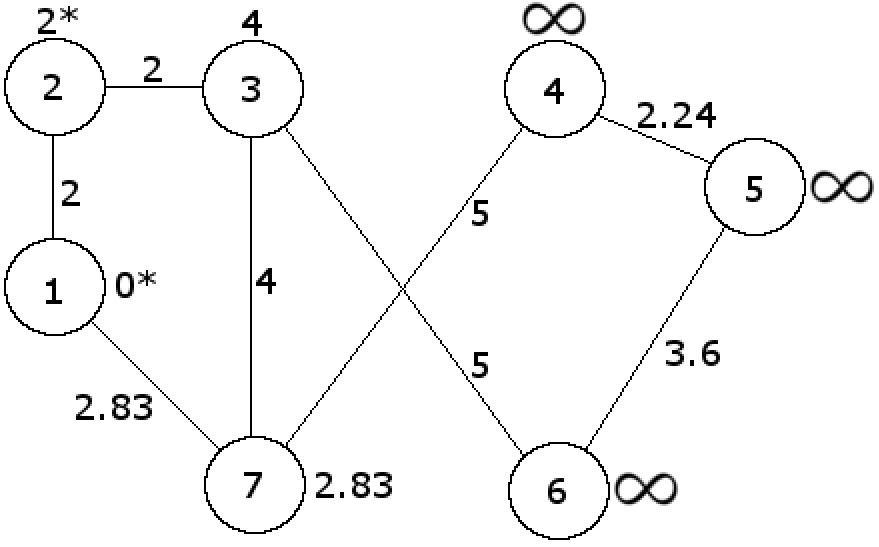


Рисунок 7. Результат второй итерации.



Рисунок 8. Итоговые значения меток.

Рассмотрим поиск оптимального пути от вершины 1 до вершины 5. Начинаем с вершины 5, перебираем все вершины, из которых можно прийти в текущую:

вершина 4: 7.83 + 2.24 = 10.07;

вершина 6: 9 + 3.6 = 12.6.

Минимальное значение у вершины 4, переходим в нее, перебираем смежные:

вершина 7: 2.83 + 5 = 5.83.

Минимальное значение у вершины 7, переходим в нее, перебираем смежные:

вершина 1: 0 + 2.83 = 2.83;

вершина 3: 4 + 4 = 8.

Минимальное значение у вершины 1, переходим в нее. Вершина 1 является исходной, значит путь найден. Таким образом, оптимальный путь из вершины 1 в вершину 5: 1-7-4-5.

Если требуется определить путь только до одной вершины, то тогда в качестве критерия останова можно использовать достижение целевой вершины.

2.2 Описание абстрактных типов данных

Для реализации алгоритма Дейкстры требуется определить абстрактный тип данных (АТД) графа, на котором производится поиск. Для этого требуется определить множество объектов, входящих в АТД и операторы над этим объектами.

2.2.1 АТД графа

Рассмотрим данные и операторы, которые должны содержаться в АТД, чтобы его можно было использовать в алгоритме Дейкстры.

1. Вершины графа.
2. getId() – получить идентификатор, позволяющий однозначно определить вершину среди множества всех вершин.
3. getInArcs() – получить список исходящих дуг.
4. getOutArcs() – получить список входящих дуг.

Для неориентированного графа требуется только один список инцидентных ребер: getEdges().

1. Ребра (дуги) графа.
2. getDestination() – получить вершину, в которую входит ребро.
3. getWeight() – получить вес.
4. Граф.
5. getVertex(id) – получить вершину по ее идентификатору.
6. getVertices() – получить все вершины.

В большинстве задач, в которых требуются алгоритмы поиска путей, количество всех вершин много больше значения средней степени вершин, т.е. графы являются разреженными. Поэтому для представления графа эффективно использовать список смежности или список инцидентности.

Для эффективного получения вершин по идентификатору для хранения множества вершин можно использовать дерево или хэш-таблицу. Если идентификаторы представляют собой непрерывную последовательность целых чисел, начинающуюся с нуля, то можно использовать линейный массив.

Пример реализации АТД графа на языке Java представлен в приложении А.

2.2.2 АТД для хранения меток

Для хранения меток с минимальной известной длиной пути до узла так же требуется определить структуру данных. Он должен содержать следующие операторы:

1. set(vertex, value) – установить значение метки для указанной вершины.
2. get(vertex) – получить значение метки для указанной вершины.
3. getMin() – получить значение наименьшей из меток.
4. takeMinVertex() – получить вершину с минимальным значением метки и извлечь ее.
5. containsNonInfinityValue() – истина, если содержит значения меток, отличных от бесконечности.

В различных источниках, например в [Дасгупта, Кормен], рекомендуется для этой цели использовать очередь с приоритетом, применяя в качестве ключа величину метки. В этом случае добавление элемента в очередь и извлечение минимального элемента осуществляются за O(log n). Однако, для пересчета меток, требуется найти их в очереди, извлечь, изменить и добавить в очередь. Поиск произвольного элемента в очереди с приоритетом, реализованной в стандартных библиотеках различных языков программирования, обычно не эффективен или невозможен. Например, в Java поиск осуществляется за O(n), в C++ поиск произвольного элемента не предоставляется вовсе. Поэтому для хранения вершин и их меток можно использовать сбалансированное дерево. В этом случае все требуемые операции будут выполняться за O(log n).

2.2.3 АТД для хранения посещенных вершин

Если АТД для хранения меток удаляет посещенные вершины, то требуется отдельная структура для хранения результата со следующими операторами:

1. set(vertex, value) – установить значение метки для указанной вершины.
2. get(vertex) – получить значение метки для указанной вершины.
3. contains(vertex) – проверить, содержит ли структура указанную вершину.

Для эффективного получения значения можно использовать дерево или хэш-таблицу. Если идентификаторы вершин представляют собой непрерывную последовательность целых чисел, начинающуюся с нуля, то можно использовать линейный массив.

2.2.4 АТД для хранения пути

Требуется определить следующие операторы:

1. addToFront(vertex) – добавить вершину в начало списка

Путь можно представить с помощью связного списка. Добавление вершины в голову дает последовательность вершин в порядке обратном добавлению, что и требуется, при получении оптимального пути, начиная с целевой вершины и заканчивая исходной.

2.3 Формальное описание

Псевдокод [Дасгупта].

procedure Dijkstra(graph, fromVertex, toVertex) {

// dist – структура для хранения меток

// result – структура для хранения посещенных вершин и

// начальная инициализация

for (u ∈ graph.getVertices())

dist.set(u, ∞)

dist.set(fromVertex, 0)

while (dist.containsNonInfinityValue()) {

value = dist.getMin()

v = dist.takeMinVertex()

result.set(v, value)

for (e ∈ v.getOutArcs()) {

v2 = e.getDestination()

oldValue = dist.get(v2)

newValue = dist.get(v) + e.getWeight()

if (newValue < oldValue)

dist.set(v2, newValue)

}

}

}

2.4 Анализ алгоритма Дейкстры

Сложность алгоритма зависит от используемых реализаций АТД [Кормен]. Проанализируем сложность выполнения псевдокода из раздела 2.3. Пусть V – это количество вершин графа, а E – количество ребер. Во время инициализации выполняется V + 1 операций dist.set, во время выполнения обхода алгоритм посещает каждую вершину максимум один раз (ни разу, если она не достижима из исходной), при этом для каждой вершины алгоритм проверяет все исходящие дуги (ребра) и пересчитывает известное расстояние для смежных вершин. Можно сформулировать выражение для оценки сложности для неизвестных АТД:

TДейкстра(V, E) = (V + 1) · Tdist.set(V) + V(Tdist.getMin(V) + Tdist.takeMinVertex(V) + Tresult.set(V)) + (1)

+ E(2 · Tdist.get(V) + Tdist.set(V))

Из выражения (1) видно, что сложность алгоритма зависит от выбранных АТД для хранения меток и результата (если это два разных объекта). Если, следуя разделу 2.2, выбрать для их реализации сбалансированные деревья, получим:

TДейкстра(V, E) = (V + 1) · O(log V) + V · (O(1) + O(log V) + O(log V)) + E(2 · O(log V) + O(log V)) ~

~ O(V · log V + V · log V + E · log V) ~ O((V + E) · log V) ~ O(E · log V)

Если использовать алгоритм для поиска пути между двумя конкретными вершинами, то время выполнения в среднем уменьшится, так как нет необходимости проверять все вершины. Тем не менее, степень роста для среднего случая не изменится.

2.5 Недостатки алгоритма Дейкстры

Рассмотрим примеры работы алгоритма в задачах поиска пути между двумя точками на некоторой местности (рисунки 9-12). Темной линией показан найденный путь, светлыми линиями показаны переходы в следующую вершину при обходе графа.

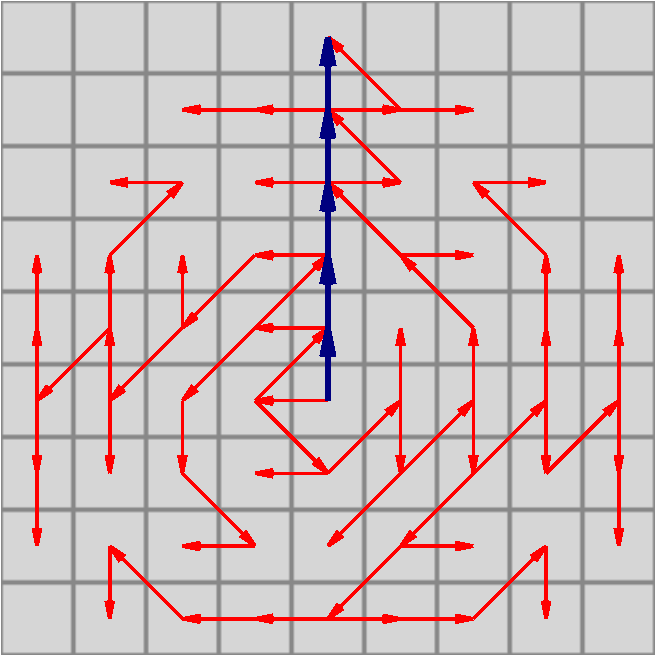


Рисунок 9. Пример поиска на карте без препятствий.

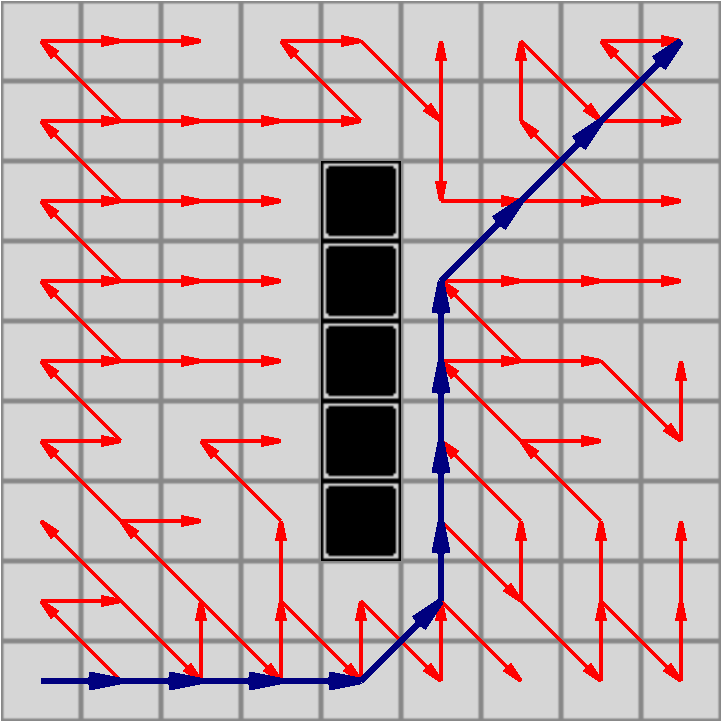


Рисунок 10. Пример поиска на карте 9х9 с препятствием.

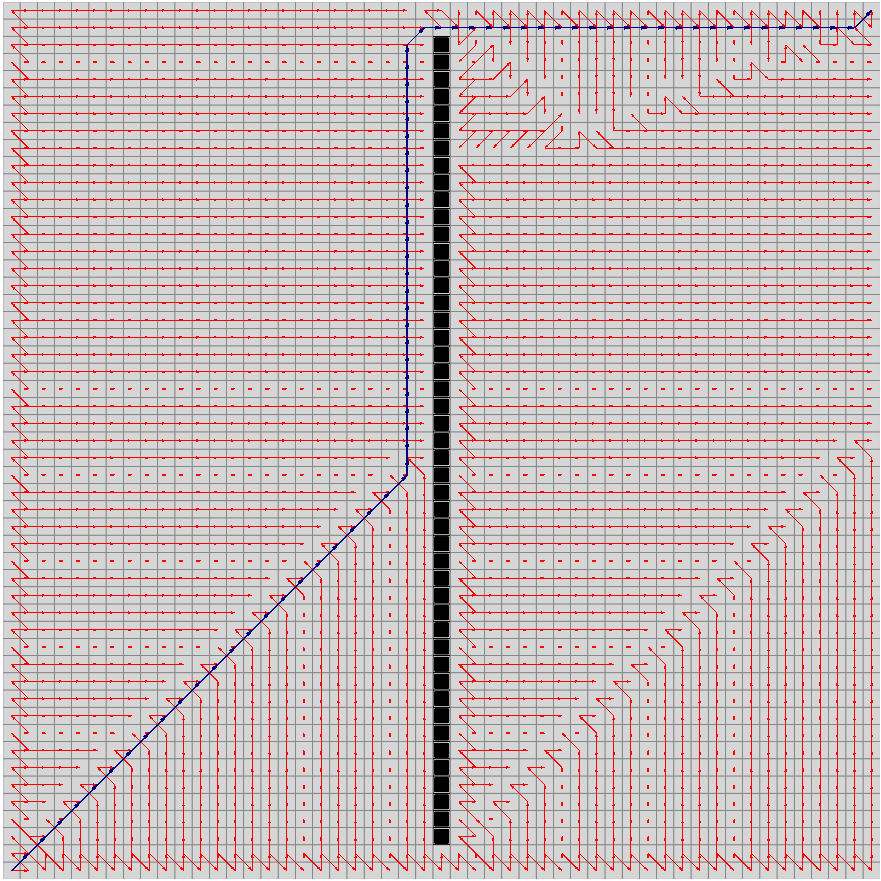


Рисунок 11. Пример поиска на карте 51х51 с препятствием.

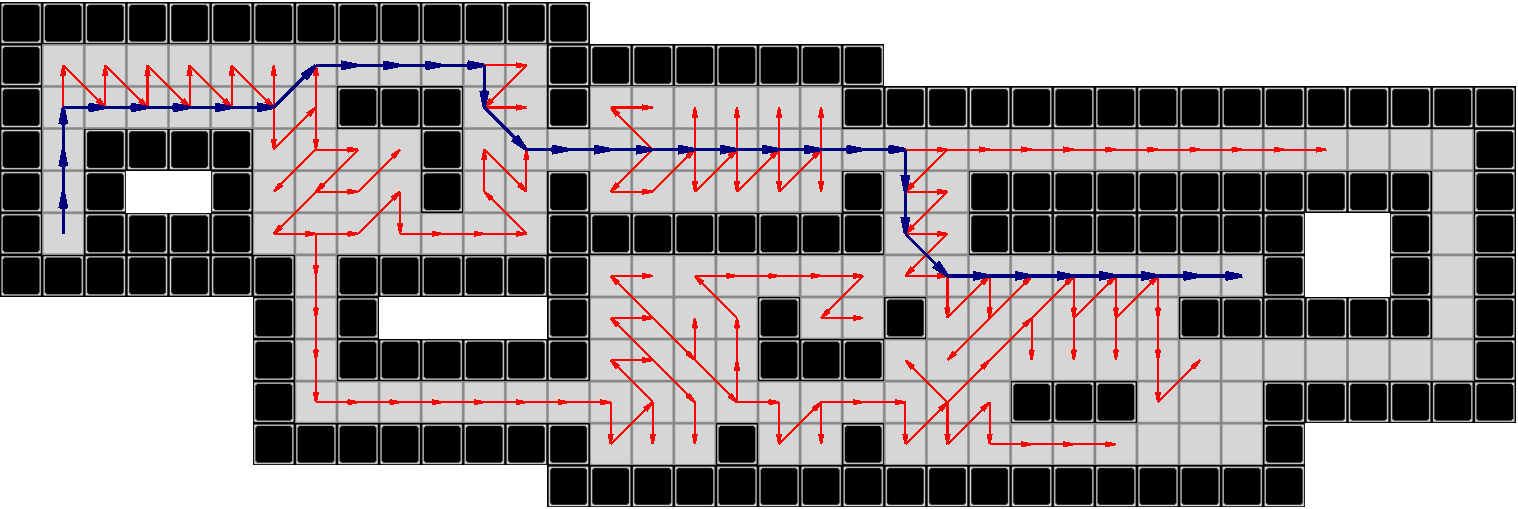


Рисунок 12. Пример поиска на относительно сложной карте.

Из рисунков видно, что при поиске пути алгоритм обходит большое количество вершин, которые потом не попадут в результат. Количество этих узлов зависит от графа. Если для построения графа используется регулярная сетка, как в рассматриваемых примерах, то в графе присутствует большое количество ребер с одинаковыми весами. Это приводит к тому, что вершины, равноудаленные от исходной, имеют одинаковые значения меток, и выбор вершины с минимальным значением метки становится неоднозначным. Это и является причиной отклонения алгоритма от оптимального пути.

3 Алгоритм A\*

ПРИЛОЖЕНИЕ А

Реализация АТД графа на языке Java.

public class Graph {

public static class Vertex {

Tile.Index index;

public float x;

public float y;

public Array<Edge> incidentEdges;

public Vertex(Tile.Index index, float x, float y) {

this.index = index;

this.x = x;

this.y = y;

incidentEdges = new Array<Edge>();

}

}

public static class Edge {

public Vertex destination;

public float weight;

}

private Map<Tile.Index, Vertex> m\_nodes

= new HashMap<Tile.Index, Vertex>();

public Map<Tile.Index, Vertex> getVertices() {

return m\_nodes;

}

public void addVertex(Vertex vertex) {

m\_nodes.put(vertex.index, vertex);

}

public Vertex getVertex(Tile.Index index) {

return m\_nodes.get(index);

}

}