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

СОДЕРЖАНИЕ

[Введение 2](#_Toc440490127)

[1 Алгоритм Дейкстры 5](#_Toc440490128)

[1.1 Неформальное описание 5](#_Toc440490129)

[1.2 Описание абстрактных типов данных 7](#_Toc440490130)

[1.2.1 АТД графа 8](#_Toc440490131)

[1.2.2 АТД для хранения меток 8](#_Toc440490132)

[1.2.3 АТД для хранения посещенных вершин 9](#_Toc440490133)

[1.2.4 АТД для хранения результата обхода 10](#_Toc440490134)

[1.2.5 АТД для хранения пути 10](#_Toc440490135)

[1.3 Формальное описание 10](#_Toc440490136)

[1.4 Анализ алгоритма Дейкстры 11](#_Toc440490137)

[1.5 Недостатки алгоритма Дейкстры 12](#_Toc440490138)

[2 Алгоритм A\* 14](#_Toc440490139)

[2.1 Эвристические функции 14](#_Toc440490140)

[2.2 Неформальное описание 17](#_Toc440490141)

[2.3 Описание абстрактных типов данных 19](#_Toc440490142)

[2.4 Формальное описание 20](#_Toc440490143)

[2.5 Анализ алгоритма A\* 21](#_Toc440490144)

[2.6 Примеры работы алгоритма 21](#_Toc440490145)

[3 Сравнение производительности алгоритмов Дейкстры и A\* 23](#_Toc440490146)

[Заключение 25](#_Toc440490147)

[Список литературы 27](#_Toc440490148)

[ПРИЛОЖЕНИЕ А 28](#_Toc440490149)

[ПРИЛОЖЕНИЕ Б 29](#_Toc440490150)

[ПРИЛОЖЕНИЕ В 32](#_Toc440490151)

[ПРИЛОЖЕНИЕ Г 34](#_Toc440490152)

# Введение

Алгоритмы Дейкстры и A\* используются для поиска кратчайшего пути между вершинами 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 пространстве. Алгоритм A\* похож на алгоритм Дейкстры, но использует эвристическую функцию для определения следующей рассматриваемой вершины, что позволяет уменьшить время поиска.

В качестве примера в данной работе будем рассматривать класс задач поиска пути между двумя вершинами на графе, построенном на основе некоторого пространства. При этом каждая вершина графа характеризуется конкретным положением, а ребрами обозначаются переходы между этими вершинами. Вес ребра зависит от расстояния и свойств местности, например, проходимости. Пусть дана карта (рисунок 1), где серым и черным цветами обозначены, соответственно, проходимые и непроходимые клетки.

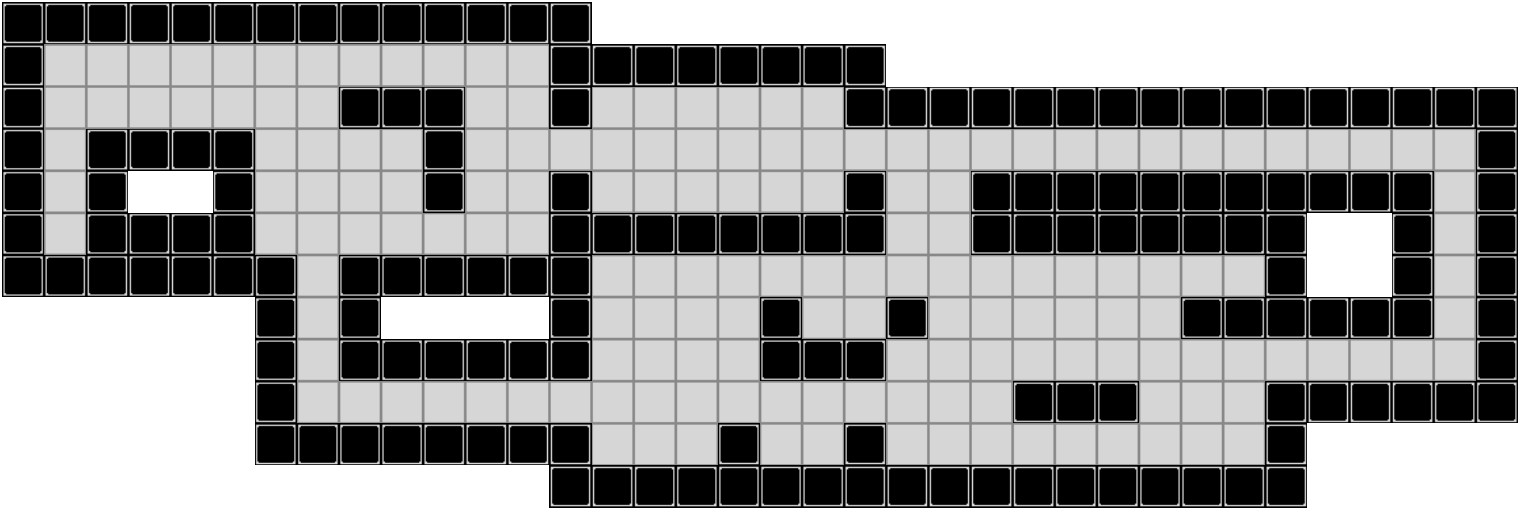


Рисунок 1. Пример карты местности.

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

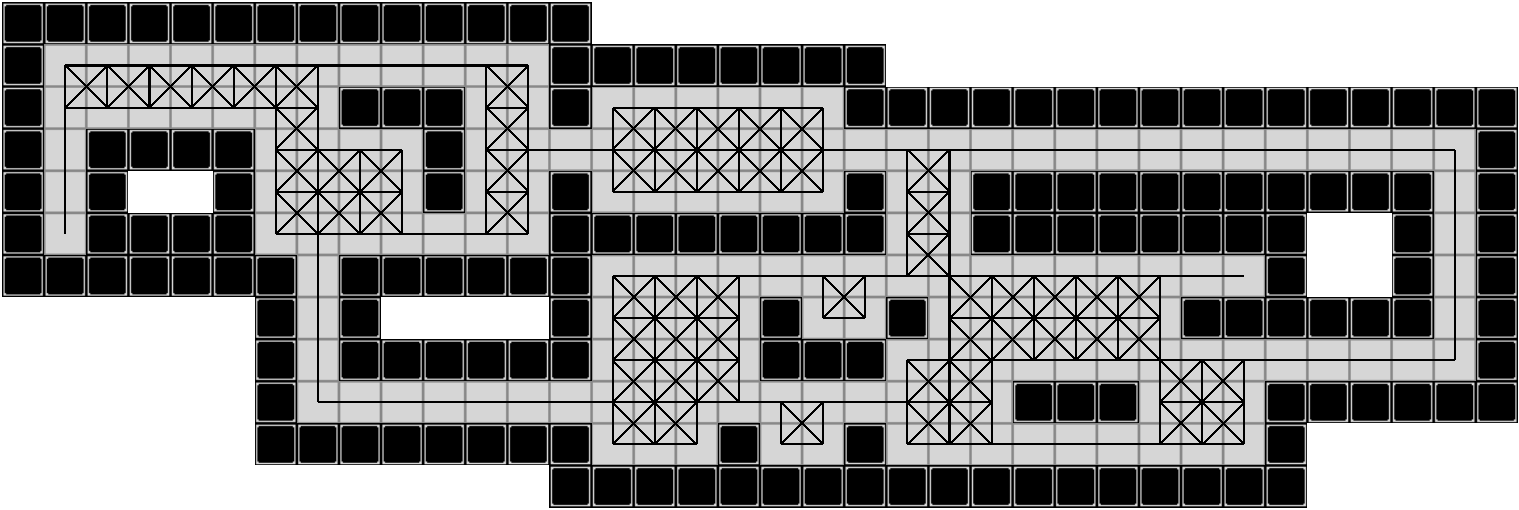


Рисунок 2. Пример местности со сгенерированным графом.

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.

# 1 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, результатом работы является дерево кратчайших путей [1].

# 1.1 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 ходе работы алгоритма эти значения могут уменьшаться, если был найден более короткий путь до вершины.

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 оно больше не будет. В данном случае выбирается вершина 1, так как ее метка равно нулю, а остальные бесконечности. 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, а в отдельную структуру сохраняется текущая вершина.

dist[2] = min{∞, 0 + 2} = 2, prev[2] = 1

dist[7] = min{∞, 0 + 2.83} = 2.83, prev[7] = 1

Результат первой итерации представлен на рисунке 6.

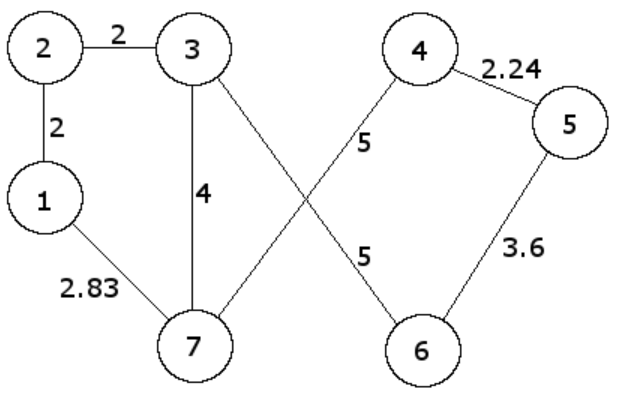
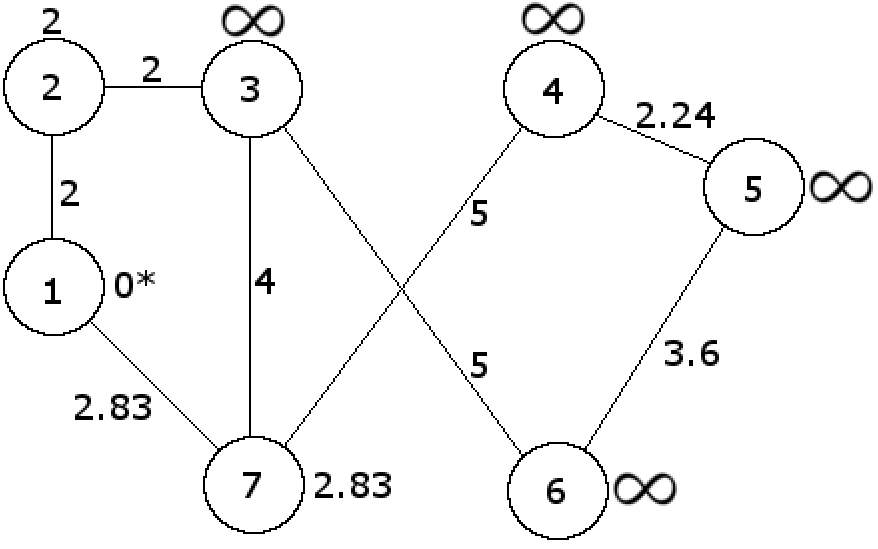
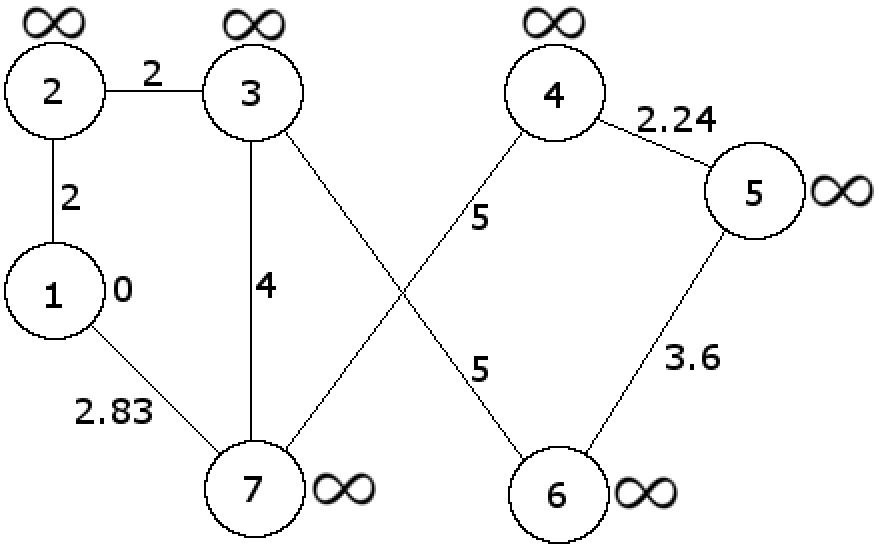
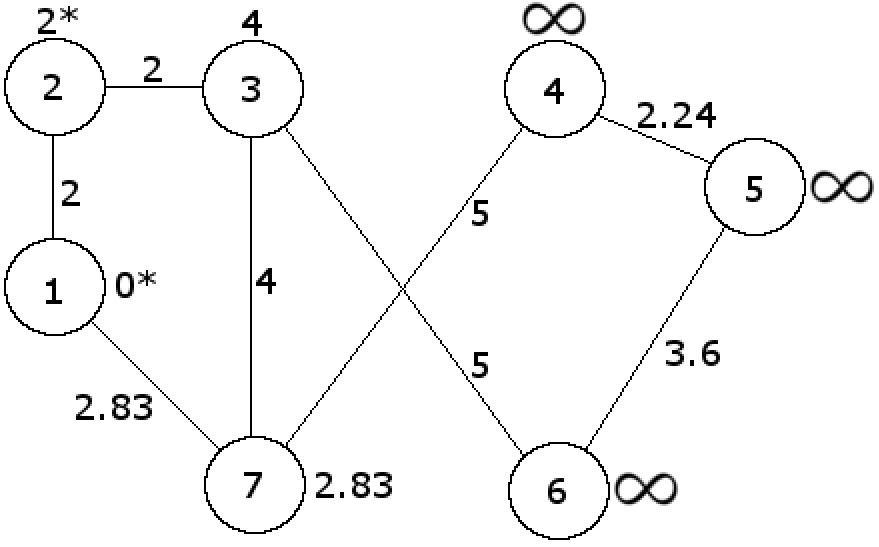


Рисунок 4. Исходный взвешенный граф.

Рисунок 6. Результат первой итерации.

Рисунок 5. Начальное состояние меток.

Рисунок 7. Результат второй итерации.

Выбирается новая вершина с минимальным значением 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.

dist[3] = min{∞, 2 + 2} = 4, prev[3] = 2

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 равным бесконечности. Второй вариант означает, что есть вершины, не достижимые из исходной. Результат обхода вершин из примера представлен на рисунке 8.

Для того, чтобы получить кратчайший путь из исходной вершины к другой, необходимо, используя структуру prev, восстановить путь, начиная с целевой вершины.



Рисунок 8. Итоговые значения меток.

Рассмотрим получение пути от вершины 1 до вершины 5:

prev[5] = 4;

prev[4] = 7;

prev[7] = 1.

Оптимальный путь из вершины 1 в вершину 5: 1-7-4-5.

Если требуется определить путь только до одной вершины, то в качестве критерия останова можно использовать достижение алгоритмом целевой вершины. При этом изменяется инициализация: нет смысла изначально помечать все вершины специальным символом или бесконечностью, так как некоторые из них могут быть не рассмотрены вовсе. Вместо этого, при пересчете вершины необходимо проверять, рассчитывается ли метка в первый раз, и инициализировать ее.

# 1.2 Описание абстрактных типов данных

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

# 1.2.1 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. getIncidentEdges() – получить список инцидентных ребер/дуг.
4. Ребра (дуги) графа.
5. getDestination() – получить вершину, в которую входит ребро.
6. getWeight() – получить вес.
7. Граф.
8. getVertex(id) – получить вершину по ее идентификатору.
9. 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. Для таких графов эффективным представлением являются список смежности и список инцидентности.

Для эффективного получения вершин по идентификатору для хранения множества вершин использовать реализацию ассоциативного массива, основанную на сбалансированном дереве или хэш-таблице. Если идентификаторы представляют собой непрерывную последовательность целых чисел, начинающуюся с нуля, можно использовать линейный массив.

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

# 1.2.2 АТД для хранения меток

Тип для хранения меток с минимальной известной длиной пути до узла должен содержать следующие операторы:

1. set(vertex, value) – установить значение для указанной вершины.
2. get(vertex) – получить значение для указанной вершины.
3. getMin() – получить вершину с наименьшим значением.
4. containsNonInfinityValue() – истина, если содержит значения меток, отличных от бесконечности.

В различных источниках, например в [1, 2], рекомендуется для этой цели использовать очередь с приоритетом, применяя в качестве параметра величину метки. В этом случае добавление элемента в очередь и извлечение минимального элемента осуществляются за O(log n). Однако, в стандартных библиотеках современных языков программирования очередь с приоритетом реализуется с помощью частично упорядоченного дерева («бинарная куча»). Такая структура данных не имеет эффективной реализации доступа к произвольному элементу и его модификации. Для решения этой проблемы в [3] рекомендуется использовать индексированную очередь с приоритетом. В этом случае помимо бинарной кучи также используются линейные или ассоциативные массивы, в которых хранятся пары <ключ объекта, номер элемента в куче с приоритетом объекта>. По ключу объекта извлекается позиция в куче, в которой хранится приоритет. Таким образом, поучается доступ к произвольному элементу за O(1). Модификация найденного элемента приводит к перебалансировке кучи за O(log n). Получаем АТД, операторы которого, необходимые для реализации алгоритма Дейкстры, выполняются за O(log n) или O(1). Реализация индексированной очереди с приоритетом представлена в приложении Б.

# 1.2.3 АТД для хранения посещенных вершин

Необходимо хранить посещенные вершины в некоторой структуре и эффективно проверять их наличие в ней.

Операторы:

1. add(vertex) – добавить вершину.
2. contains(vertex) – проверить, есть ли вершина в множестве.

Для эффективного добавления и проверки можно использовать реализации множества на сбалансированном дереве или хэш-таблице. Если идентификаторы вершин представляют собой непрерывную последовательность целых чисел, начинающуюся с нуля, можно использовать линейный массив.

# 1.2.4 АТД для хранения результата обхода

Для восстановления оптимального пути после завершения обхода требуется для каждой рассмотренной вершины сохранять ту, что предшествует ей.

Операторы:

1. set(vertex, previousVertex) – установить значение для указанной вершины.
2. get(vertex) – получить значение для указанной вершины.

Для эффективного получения значения можно использовать реализацию ассоциативного массива, основанную на сбалансированном дереве или хэш-таблице. Если идентификаторы вершин представляют собой непрерывную последовательность целых чисел, начинающуюся с нуля, можно использовать линейный массив.

# 1.2.5 АТД для хранения пути

Операторы:

1. addToFront(vertex) – добавить вершину в начало

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.

# 1.3 Формальное описание

Рассмотрим псевдокод алгоритма Дейкстры [2]. Возьмем частный случай алгоритма, когда ищется расстояние только до конкретной целевой вершины.

procedure Dijkstra(graph, start, goal) {

processed = {} // просмотренные вершины

dist[start] = 0 // рассматриваемые вершины и их метки

prev[start] = null // предыдущие вершины

// обход вершин

while (dist не пусто) {

v = «извлечь из dist вершину с минимальным расстоянием от исходной вершины»

«добавить v в processed»

if (v == goal)

break // целевая вершина достигнута

for (e ∈ дуги, исходящие из v) {

v2 = e.getDestination()

if (processed содержит v2)

continue // вершина уже была посещена

d = dist[v] + e.getWeight()

if (dist не содержит v2 OR dist[v2] > d) {

dist[v2] = d

prev[v2] = v

}

}

}

}

# 1.4 Анализ алгоритма Дейкстры

Сложность алгоритма зависит от используемых реализаций АТД [1]. Проанализируем сложность выполнения псевдокода из раздела 1.3. Пусть V – это количество вершин графа, а E – количество ребер. 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·(Tdist.getMin(V) + Tprocessed.set(V)) +

+ E·(2·Tdist.get(V) + Tdist.set(V) + Tprev.set(V))

Если, следуя разделу 2.2, выбрать для их реализации сбалансированные деревья, получим:

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

~ O((V + E) · log V)

Если какой-либо из операторов имеет степень роста O(n), то по правилу умножения получаем:

TДейкстра(V, E) ~ O(V2)

Таким образом, степень роста алгоритма Дейкстры определяется самым медленным оператором используемых в нем АТД.

# 1.5 Недостатки алгоритма Дейкстры

Рассмотрим примеры работы алгоритма в задачах поиска пути между двумя точками на некоторой местности (рисунки 9-11). Темной линией показан найденный путь, светлыми линиями показаны переходы в следующую вершину при обходе графа.

Из рисунков видно, что при поиске пути алгоритм обходит большое количество вершин, которые потом не попадут в результат. Количество этих узлов зависит от графа. Алгоритм Дейкстры представляет собой обход графа в ширину, в котором используется очередь с приоритетом. 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 длины путей и проверяются во всех направлениях в порядке удаления от исходной.

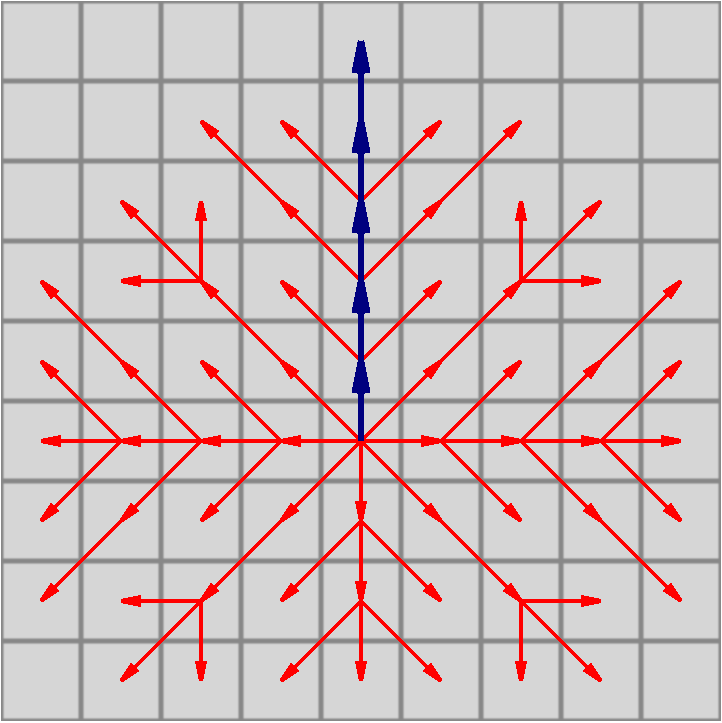


Рисунок 9. Пример поиска на карте 9x9 без препятствий.

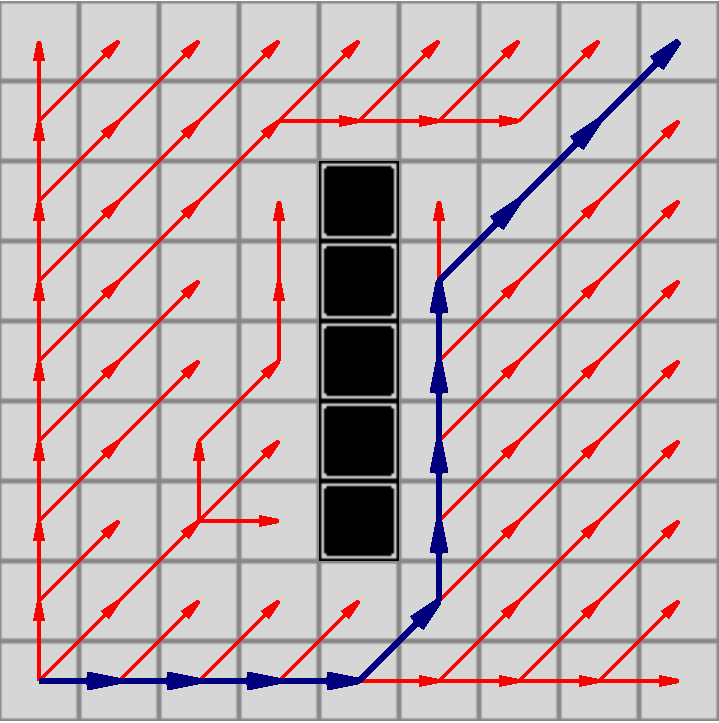


Рисунок 10. Пример поиска на карте 9х9 с препятствием.

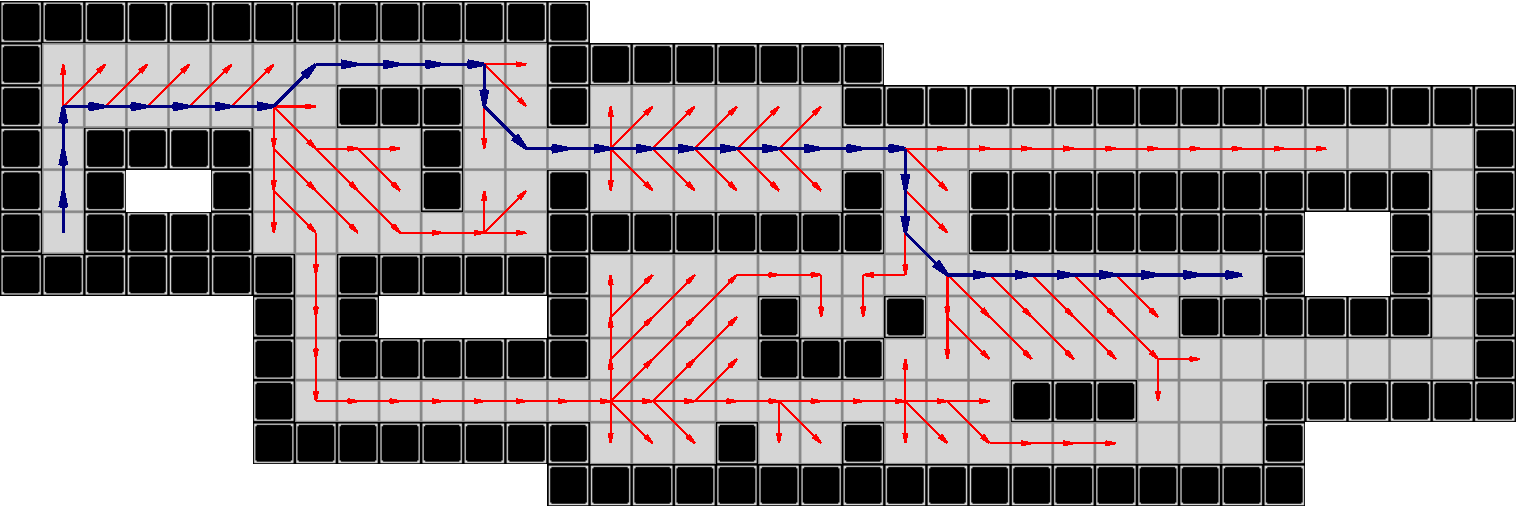


Рисунок 11. Пример поиска на относительно сложной карте.

# 2 Алгоритм A\*

Алгоритм A\* (произносится «А звезда» или «А стар», от англ. A star) – алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины к другой. В 1964 году Нильс Нильсон изобрел эвристический подход к увеличению скорости алгоритма Дейкстры. Этот алгоритм был назван А1. В 1967 году Бертрам Рафаэль сделал значительные улучшения по этому алгоритму, но ему не удалось достичь оптимальности. Он назвал этот алгоритм A2. Тогда в 1968 году Питер Э. Харт представил аргументы, которые доказывали, что A2 был оптимальным при использовании последовательной эвристики лишь с незначительными изменениями. Таким образом, он обозначил новый алгоритм в синтаксисе звездочкой, он начинается на А и включает в себя все возможные номера версий.

# 2.1 Эвристические функции

Эвристика позволяет сократить количество вершин, просмотренных алгоритмом, предлагая направление для поиска, которое позволит приблизиться к цели, но при этом не гарантирует, что полученное приближение будет верным. Эвристика может использоваться для настройки и управления алгоритмом [4]:

1. Если эвристическая функция всегда возвращает ноль, то алгоритм превращается в алгоритм Дейкстры, т.е. для выбора вершины используется только расстояние от исходной вершины.
2. Если оценка всегда меньше либо равна реальной дальности от вершины до цели, то A\* гарантированно вернет оптимальный путь.
3. Если эвристическая функция возвращает точное расстояние от вершины до цели, то алгоритм посетит только вершины, входящие в оптимальный путь.
4. Если оценка завышена, т.е. ее значение для некоторых вершин превышает реальную длину пути, алгоритм не гарантирует, что найденный путь будет оптимальным. Однако, завышенная оценка отсеивает большое количество вершин, что позволяет увеличить быстродействие за счет потери качества.
5. Если оценка много больше расстояния от исходной вершины, то поведение алгоритма близко к жадному поиску по первому наилучшему совпадению.

Эвристическая функция называется допустимой эвристической оценкой, если она не переоценивает расстояние между узлами.

Как правило, A\* используются на графах, основанных на физическом представлении, например, в прямоугольной системе координат. Используя геометрические свойства вершин графа, можно оценить минимально возможное расстояние между ними. Рассмотрим эвристические функции, которые обычно применяются с A\* [3, 4]:

1. Манхэттенское расстояние. Метрика, введённая Германом Минковским. Согласно этой метрике, расстояние между двумя точками равно сумме модулей разностей их координат (рисунок 12). Также имеет название «расстояние городских кварталов». Используется, если перемещение по клеткам допускается только в четырех направлениях.

.

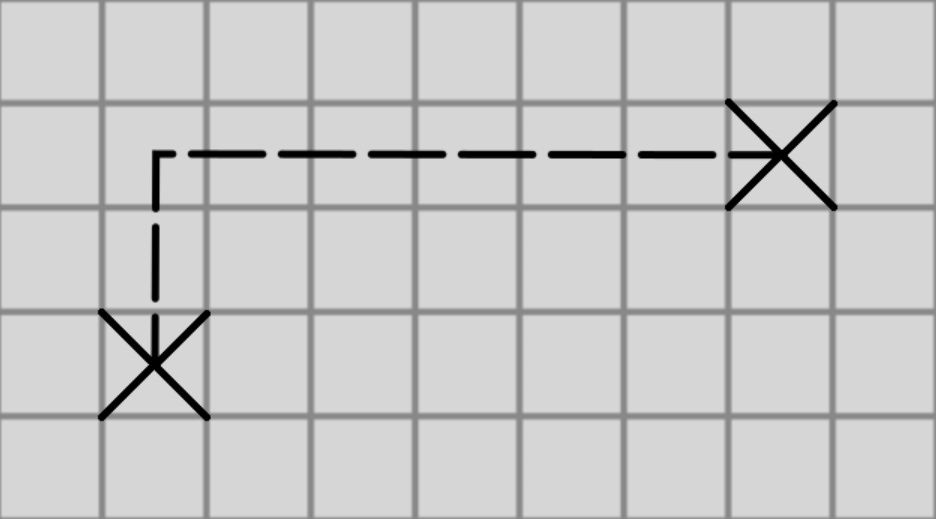


Рисунок 12. Манхэттенское расстояние между двумя узлами.

1. Диагональное расстояние. Используется, если разрешено движение между диагональными клетками. Одна из разностей координат проходится по диагонали, другая – по прямой аналогично манхэттенскому расстоянию (рисунок 13).

где .



Рисунок 13. Диагональное расстояние между двумя узлами.

1. Эвклидово расстояние. Представляет собой геометрическое расстояние между двумя точками в пространстве (рисунок 14). Гарантированно дает минимальную оценку расстояния.

.



Рисунок 14. Эвклидово расстояние между двумя узлами.

# 2.2 Неформальное описание

Аналогично алгоритму Дейкстры A\* поочередно просматривает вершины и обновляет минимальное известное расстояние от исходной вершины до текущей. Отличие заключается в получении следующей вершины из очереди. Помимо информации о расстоянии от исходной вершины, A\* также использует информацию о положении целевой вершины. Каждая следующая вершина выбирается по минимуму суммы f(v) = g(v) + h(v), где g(v) – минимальное известное расстояние от исходной вершины до вершины v, h(v) – значение эвристической функции для вершины v.

Рассмотрим пример работы алгоритма на графе, изображенном на рисунке 4. Найдем кратчайшие пути от вершины 1 до вершины 5, в качестве эвристики используем эвклидово расстояние. В A\* используется два множества: множество раскрытых вершин (open), т.е. тех, для кого алгоритм уже посчитал метку, но еще не посетил, и множество закрытых вершин (closed), т.е. тех, кого алгоритм уже посетил.

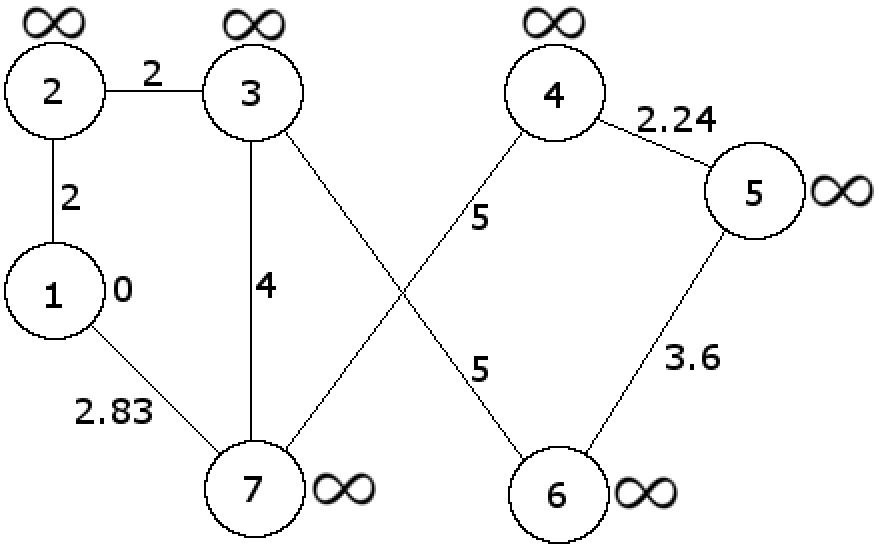


Рисунок 15. Начальное состояние меток.

1) Выбираем вершину 1, для смежных ей вершин рассчитываем расстояния от начала и эвристики (рисунок 16) и помещаем ее в множество open. В массив prev помещаем предшествующие им вершины.

g(2) = min(∞, 0 + 2) = 2; h(2) = 7.07; f(2) = 9.07, prev[2] = 1;

g(7) = min(∞, 0 + 2.83) = 2.83; h(7) = 5.83; f(7) = 8.66, prev[7] = 1;

2) Переходим в вершину 7 (рисунок 17).

g(3) = min(∞, 2.83 + 4) = 6.83; h(3) = 5.1; f(3) = 11.93, prev[3] = 7;

g(4) = min(∞, 2.83 + 5) = 7.83; h(4) = 2.24; f(4) = 10.07, prev[4] = 7 ;

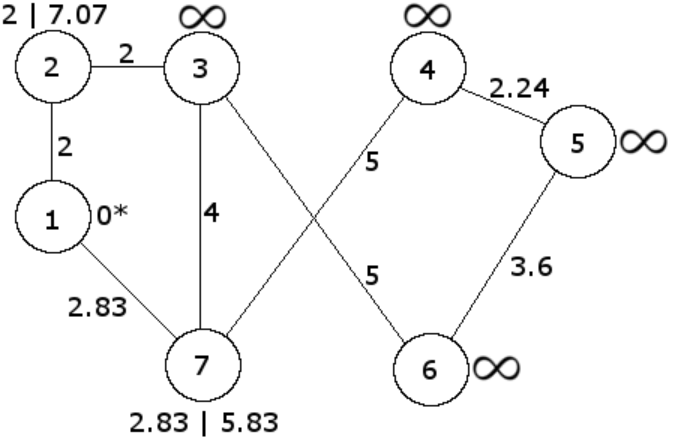


Рисунок 16. Результат первой итерации.

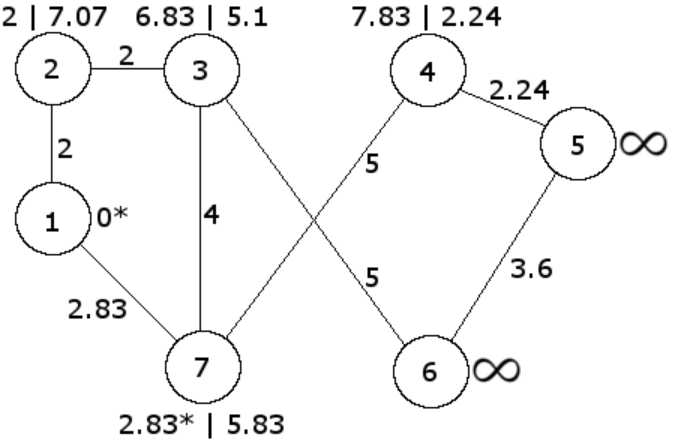


Рисунок 17. Результат второй итерации.

3) Переходим в вершину 2 (рисунок 18).

g(3) = min(6.83, 2 + 2) = 4; h(3) = 5.1; f(3) = 9.1, prev[3] = 2;

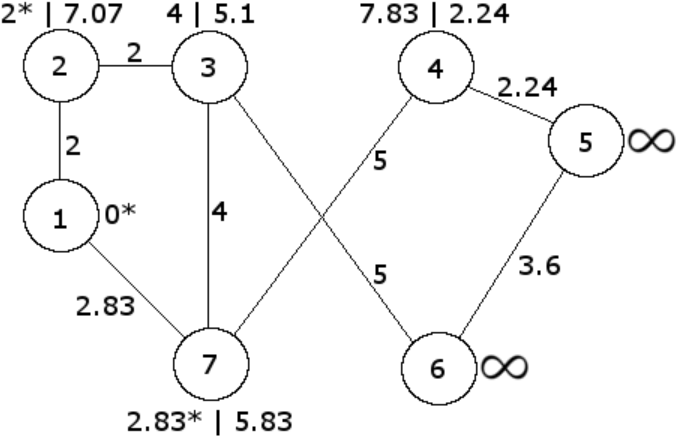


Рисунок 18. Результат третьей

итерации.

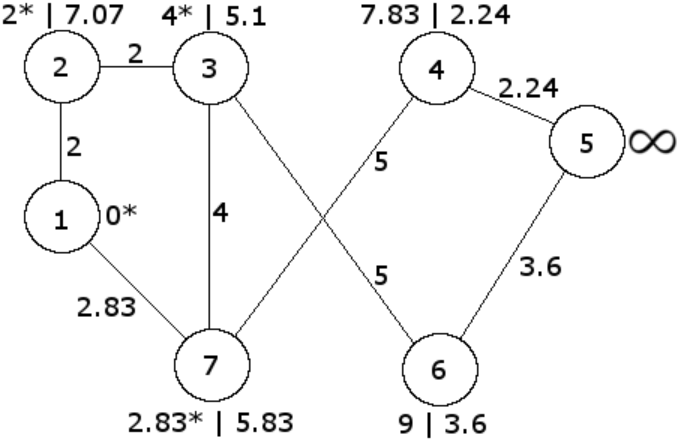


Рисунок 19. Результат четвертой итерации.

4) Переходим в вершину 3 (рисунок 19).

g(6) = min(∞, 4 + 5) = 9; h(6) = 3.6; f(6) = 12.6, prev[6] = 3;

5) Переходим в вершину 4 (рисунок 20).

g(5) = min(∞, 7.83 + 2.24) = 10.07; h(5) = 0; f(5) = 10.07, prev[5] = 4;

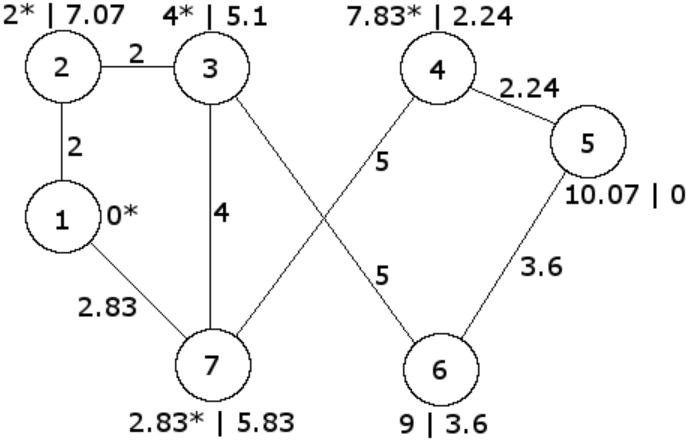


Рисунок 20. Результат пятой итерации.

6) Вершина с минимальным f – 5, извлекаем ее из open и помещаем в closed. Достигнута целевая вершина, конец работы алгоритма. Получение оптимального пути по полученным результатам выполняется аналогично алгоритму Дейкстры.

# 2.3 Описание абстрактных типов данных

АТД аналогичны описанном в разделе 1.2, за исключением того, что АТД для хранения меток должен учитывать эвристику при поиске минимального элемента. Для этого можно использовать составные метки, состоящие из расстояния от исходной вершины и оценки расстояния до целевой вершины, и определить для них оператор сравнения, учитывающий сумму двух величин.

# 2.4 Формальное описание

procedure AStar(graph, start, goal, heuristic) {

closed = {} // посещенные вершины

open = {start} // рассматриваемые вершины и их метки

prev[start] = null // структура для хранения предшествующих вершин

while (open не пусто) {

v = «извлечь из open вершину с минимальным значением f = g + h»

«добавить v в closed»

if (v == goal)

break // целевая вершина достигнута

for (e ∈ дуги, исходящие из v) {

v2 = e.getDestination()

if (closed содержит v2)

continue // вершина уже была посещена

newG = g[v] + e.getWeight()

if (open не содержит v2) {

h[v2] = heuristic(v2, goal)

g[v2] = newG

prev[v2] = v

} else {

if (g[v2] > newG) {

g[v2] = newG

prev[v2] = v

}

}

}

}

}

# 2.5 Анализ алгоритма A\*

Временная сложность алгоритма зависит от эвристики. В худшем случае число вершин, исследуемых алгоритмом растет экспоненциально по сравнению с длиной оптимального пути: O(bd), где b – коэффициент ветвления (средняя степень вершины), d – длина оптимального пути. Сложность становится полиномиальной, если эвристическая функция удовлетворяет следующему условию:

, где – это оптимальная эвристика, то есть точное расстояние от x до цели.

# 2.6 Примеры работы алгоритма

На рисунках 21 – 25 изображены примеры работы алгоритма A\* на разных графах. Графы основаны на регулярных решетках, в качестве эвристики используется диагональное расстояние. Из примеров видно, что по сравнению с алгоритмом Дейкстры A\* рассматривает меньшее число вершин.

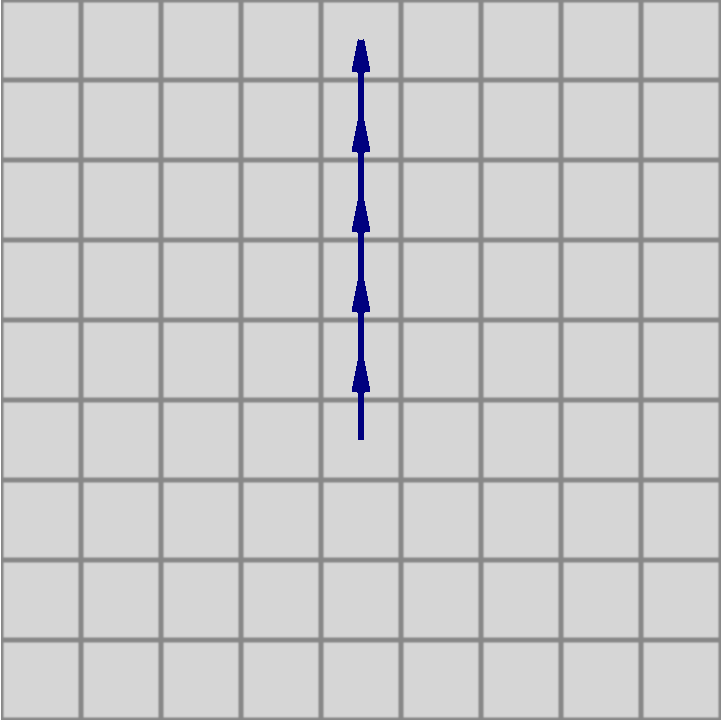


Рисунок 21. Пример поиска на карте 9x9 без препятствий.

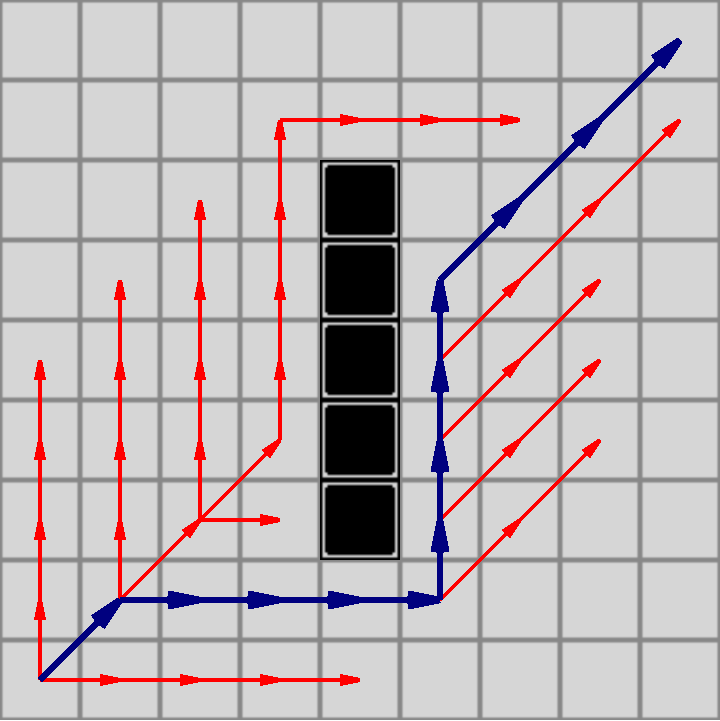


Рисунок 22. Пример поиска на карте 9х9 с препятствием.

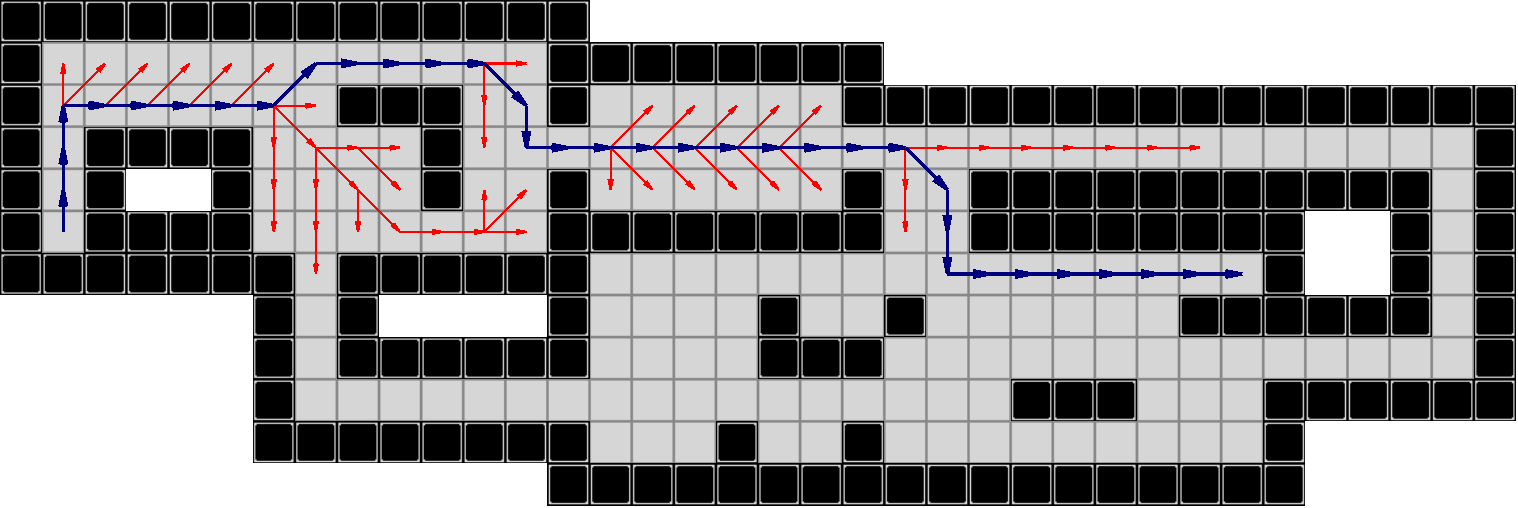


Рисунок 23. Пример поиска на относительно сложной карте.

На рисунках 24 и 25 изображен пример работы алгоритма с допустимой и завышенной оценками соответственно. Из них видно, что завышенная оценка уменьшает количество просмотренных вершин, но в результате дает не оптимальный путь (длиной 12.23 при оптимальном значении 11.41).

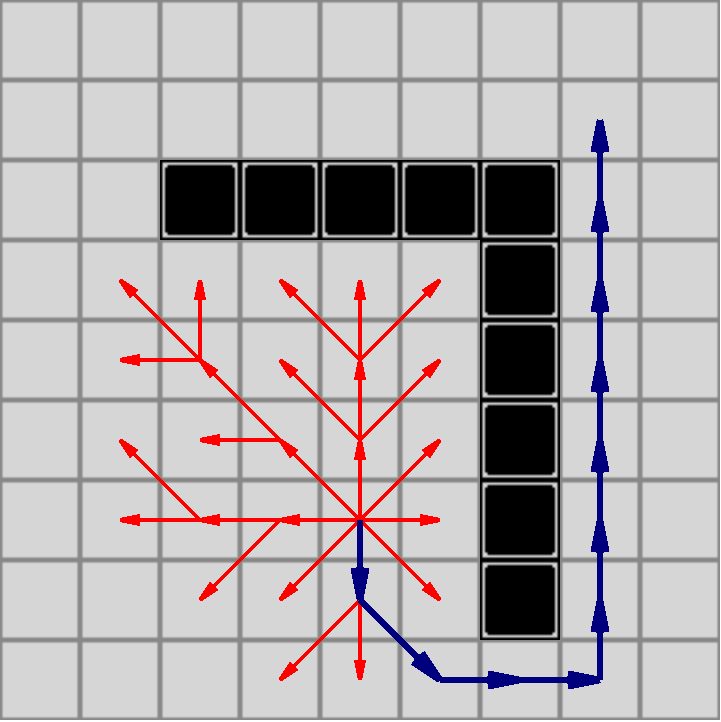


Рисунок 24. Пример поиска с допустимой оценкой.

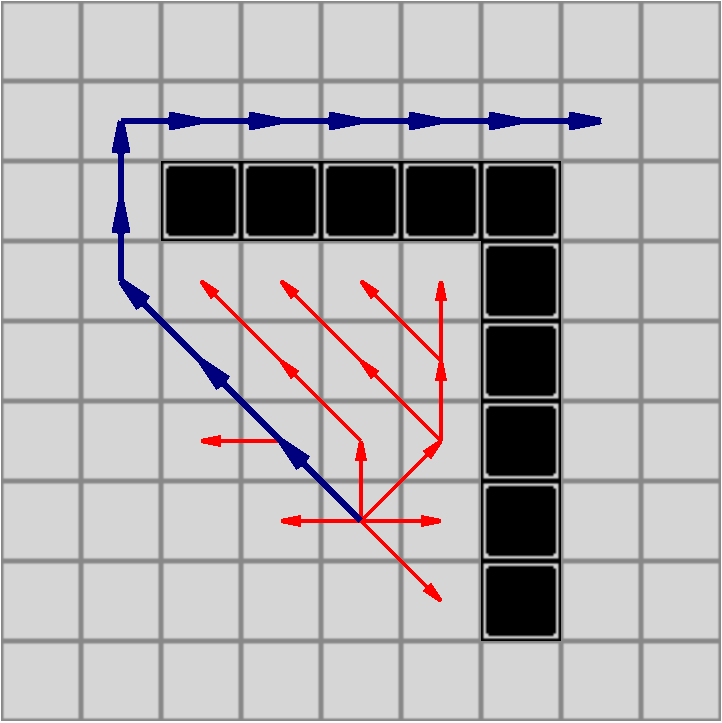


Рисунок 25. Пример поиска с завышенной оценкой.

# 3 Сравнение производительности алгоритмов Дейкстры и A\*

Было произведено сравнение времени выполнения программных реализаций алгоритмов. Реализации алгоритмов на языке Java представлены в приложениях В и Г.

Для тестирования использовалась карта из квадратных клеток с препятствием в центре. Размер карты варьировался от 5x5 до 801x801, высота препятствия равна высоте карты минус 4 (отступы в две клетки сверху и снизу). Для каждой карты и каждого алгоритма производилось 100 запусков поиска пути между вершиной в нижнем левом углу и вершиной в верхнем правом углу. Графики зависимостей времени выполнения от порядка графа представлены на рисунках 28 и 29. На рисунке 28 показы измерения для карт размером от 5x5 до 377x377 с шагом 4 (количество вершин от 24 до 141756). На рисунке 29 добавлены измерения для карт размером от 401x401 до 801x801 с шагом 100 (количество вершин от 160404 до 640804).

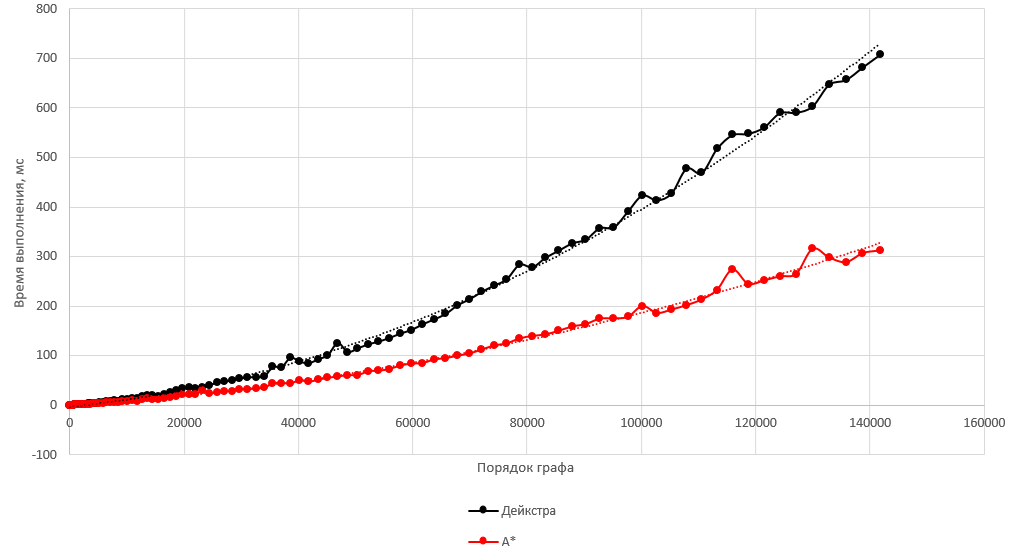


Рисунок 26. Графики зависимости времени выполнения от порядка графа (до 141756).

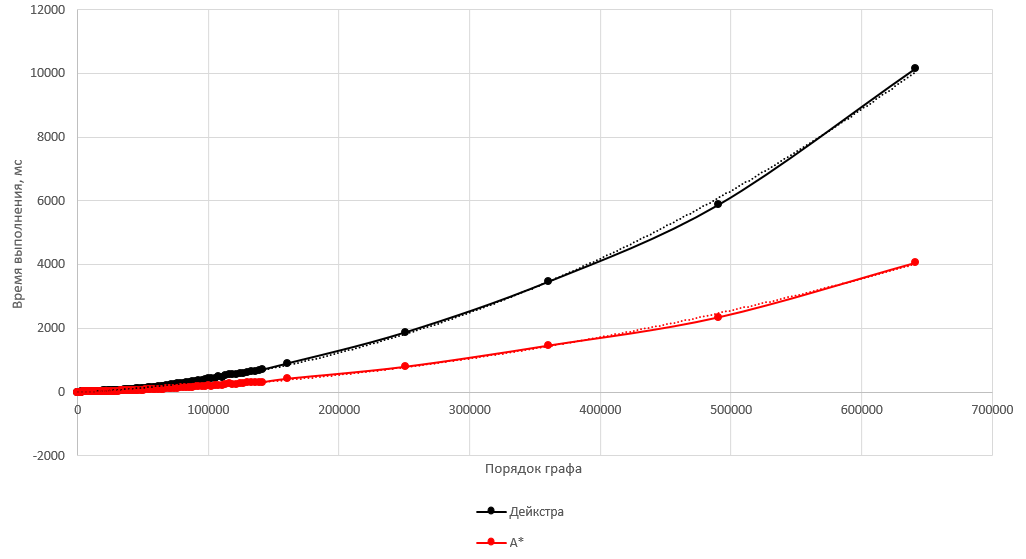


Рисунок 27. Графики зависимости времени выполнения от порядка графа (до 640804).

Из графиков видно, что степень роста времени выполнения для данной реализации алгоритмов близко к полиномиальной.

Для рисунка 26 аппроксимации следующие:

|  |  |
| --- | --- |
| Дейкстра: | y = 3E-08x2 + 0,0011x - 3,1272 |
| A\*: | y = 1E-08x2 + 0,0008x - 1,4539 |

Для рисунка 27:

|  |  |
| --- | --- |
| Дейкстра: | y = 2E-08x2 + 0,0019x - 11,105 |
| A\*: | y = 8E-09x2 + 0,0011x - 4,3626 |

Таким образом, получена квадратичная степень роста времени выполнения, что отличается от теоретического значения O((V + E) log V). Возможной причиной расхождений являются затраты времени на выделение памяти и управление ресурсами.

# Заключение

В данном курсовом проекте были рассмотрены и проанализированы алгоритмы Дейкстры и A\*: пошагово показан пример работы, рассмотрены структуры данных для представления графа и вспомогательных данных алгоритмов, приведены примеры работы реализованных алгоритмов, а также произведено сравнение времени выполнения.

Алгоритм Дейкстры – это алгоритм поиска оптимального пути во взвешенном графе с неотрицательными весами, который используется в телекоммуникационной технике, логистике и других областях. Алгоритм поочередно обходит вершины, каждый раз обновляя информацию о минимальном пути от исходной вершины до текущей. Он имеет степень роста, зависящую от структур данных, используемых в его реализации. При использовании эффективных структур данных теоретическая степень роста равна O((E+V) log V).

Алгоритм Дейкстры похож на обход графа в ширину за исключением того, что вершины извлекаются не в порядке их попадания туда, а согласно приоритету. Это приводит к тому, что при поиске пути из одной вершины к другой, алгоритм просматривает большое количество вершин, равноудаленных от исходной, которые не входят в оптимальный путь.

Для устранения этого недостатка был разработан алгоритм A\*, использующий эвристику для определения направления поиска. A\* обычно применяется в робототехнике, видеоиграх, в железнодорожном транспорте и других областях, где графы используются для представления пространственной структуры местности. В качестве эвристики используется оценка минимального расстояния от текущей вершины до целевой, каждая следующая вершина выбирается по минимуму значения f(v) = g(v) + h(v), где g(v) – расстояние между исходной вершиной и v, используемое в алгоритме Дейкстры, а h(v) – значение эвристической функции для вершины v. Благодаря этому алгоритм A\* в первую очередь проверяет вершины, которые приближают его к цели, в отличие от алгоритма Дейкстры, который проверяет все вершины.

# Список литературы

1 Кормен T., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 3-е издание. –М.: ООО «И.Д. Вильямс», 2013. – 1328 с. : ил.

2 Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы; Пер. с англ. под ред. А. Шеня. –М.: МЦНМО, 2014. – 320 с.

3 Buckland M. Programming game AI by example. – Wordwire Publishing Inc, 2005. – 520 c.

4 Amit Patel. Amit’s A\* Pages / Stanford University – URL: <http://theory.stanford.edu/~amitp/GameProgramming/index.html> – (дата обращения: 10.01.2016).

# 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);

}

}

# ПРИЛОЖЕНИЕ Б

Реализация индексированной очереди с приоритетом на языке Java

public class IndexedPriorityQueue<K, V extends Comparable<V>> {

public static class Result<K, V> {

public K key;

public V priority;

public Result(K key, V priority) {

this.key = key;

this.priority = priority;

}

}

private int m\_capacity;

private int m\_size;

private V[] m\_heap;

private HashMap<K, Integer> m\_keyToHeapPos;

private K[] m\_heapPosToKey;

public IndexedPriorityQueue(int capacity) {

m\_capacity = capacity;

m\_size = 0;

m\_heap = (V[])new Comparable[m\_capacity];

m\_keyToHeapPos = new HashMap<K, Integer>(m\_capacity);

m\_heapPosToKey = (K[])new Object[m\_capacity];

}

public void insert(K key, V priority) {

if (m\_size >= m\_capacity)

throw new IndexOutOfBoundsException("No more capacity");

m\_heap[m\_size] = priority;

m\_keyToHeapPos.put(key, m\_size);

m\_heapPosToKey[m\_size] = key;

if (m\_size > 1)

goUp(m\_size);

++m\_size;

}

public V getPriority(K key) {

int pos = m\_keyToHeapPos.get(key);

return m\_heap[pos];

}

public void changePriority(K key, V value) {

int pos = m\_keyToHeapPos.get(key);

if (m\_heap[pos].compareTo(value) > 0)

goDown(pos);

else

goUp(pos);

}

public Result<K, V> deleteMin() {

if (isEmpty())

throw new IndexOutOfBoundsException("Queue is empty");

K key = m\_heapPosToKey[0];

V priority = m\_heap[0];

--m\_size;

if (!isEmpty()) {

swap(0, m\_size);

if (m\_size > 1)

goDown(0);

}

return new Result<K, V>(key, priority);

}

public boolean isEmpty() {

return (m\_size == 0);

}

public boolean contains(K key) {

return m\_keyToHeapPos.containsKey(key);

}

private void swap(int heapPos1, int heapPos2) {

V tmp = m\_heap[heapPos1];

m\_heap[heapPos1] = m\_heap[heapPos2];

m\_heap[heapPos2] = tmp;

K key1 = m\_heapPosToKey[heapPos1];

K key2 = m\_heapPosToKey[heapPos2];

m\_keyToHeapPos.put(key1, heapPos2);

m\_keyToHeapPos.put(key2, heapPos1);

m\_heapPosToKey[heapPos1] = key2;

m\_heapPosToKey[heapPos2] = key1;

}

private void goUp(int pos) {

do {

int parentPos = (pos - 1) / 2;

if (m\_heap[pos].compareTo(m\_heap[parentPos]) >= 0)

break;

swap(pos, parentPos);

pos = parentPos;

} while (pos != 0);

}

private void goDown(int pos) {

do {

int rightChild = (pos + 1) \* 2;

int leftChild = rightChild - 1;

int minChild;

if (leftChild >= m\_size)

break; // pos is leaf

if (rightChild >= m\_size)

minChild = leftChild; // no right child

else

minChild = (m\_heap[leftChild].compareTo(m\_heap[rightChild]) <= 0)

? leftChild : rightChild;

if (m\_heap[pos].compareTo(m\_heap[minChild]) <= 0)

break;

swap(pos, minChild);

pos = minChild;

} while (true);

}

}

# ПРИЛОЖЕНИЕ В

Реализация алгоритма Дейкстры на языке Java

public class DijkstraAlgorithm extends Pathfinder {

private Graph m\_graph;

public DijkstraAlgorithm(Graph graph) {

m\_graph = graph;

}

@Override

public List<Graph.Vertex> findPath(Graph.Vertex start, Graph.Vertex goal) {

IndexedPriorityQueue<Tile.Index, Float> dist

= new IndexedPriorityQueue<Tile.Index, Float>(m\_graph.getOrder());

Set<Tile.Index> processed

= new HashSet<Tile.Index>();

HashMap<Tile.Index, Graph.Vertex> previous

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

// calculate path costs

boolean goalAchieved = false;

dist.insert(start.index, 0.0f);

while (!dist.isEmpty()) {

IndexedPriorityQueue.Result<Tile.Index, Float> nextVertex

= dist.deleteMin();

Tile.Index index = nextVertex.key;

float distance = nextVertex.priority;

Graph.Vertex v = m\_graph.getVertex(index);

processed.add(index);

if (v == goal) {

goalAchieved = true;

break;

}

for (Graph.Edge e : v.incidentEdges) {

Graph.Vertex v2 = e.destination;

if (processed.contains(v2.index))

continue;

float d = distance + e.weight;

if (!dist.contains(v2.index)) {

dist.insert(v2.index, d);

previous.put(v2.index, v);

} else {

float oldD = dist.getPriority(v2.index);

if (oldD > d) {

dist.changePriority(v2.index, d);

previous.put(v2.index, v);

}

}

}

}

if (!goalAchieved)

return null;

// get optimal path

Graph.Vertex v = goal;

List<Graph.Vertex> path = new ArrayList<Graph.Vertex>();

path.add(goal);

while (v.index != start.index) {

v = previous.get(v.index);

path.add(v);

}

Collections.reverse(path);

return path;

}

}

# ПРИЛОЖЕНИЕ Г

Реализация алгоритма A\* на языке Java

public class AStarAlgorithm extends Pathfinder {

public interface Heuristic {

float calculate(Graph.Vertex vertex, Graph.Vertex targetVertex);

}

private Graph m\_graph;

private Heuristic m\_heuristic;

public AStarAlgorithm(Graph graph, Heuristic heuristic) {

m\_graph = graph;

m\_heuristic = heuristic;

}

@Override

public List<Graph.Vertex> findPath(Graph.Vertex start, Graph.Vertex goal) {

class Info implements Comparable<Info> {

float distance;

float heuristic;

Info(float distance, float heuristic) {

this.distance = distance;

this.heuristic = heuristic;

}

@Override

public int compareTo(Info o) {

return (int)Math.signum((distance + heuristic) - (o.distance + o.heuristic));

}

}

IndexedPriorityQueue<Tile.Index, Info> dist

= new IndexedPriorityQueue<Tile.Index, Info>(m\_graph.getOrder());

Set<Tile.Index> processed

= new HashSet<Tile.Index>();

HashMap<Tile.Index, Graph.Vertex> previous

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

// calculate path costs

boolean goalAchieved = false;

dist.insert(start.index, new Info(0.0f, 0.0f));

while (!dist.isEmpty()) {

IndexedPriorityQueue.Result<Tile.Index, Info> nextVertex = dist.deleteMin();

Tile.Index index = nextVertex.key;

Info info = nextVertex.priority;

Graph.Vertex v = m\_graph.getVertex(index);

processed.add(index);

if (v == goal) {

goalAchieved = true;

break;

}

for (Graph.Edge e : v.incidentEdges) {

Graph.Vertex v2 = e.destination;

if (processed.contains(v2.index))

continue;

float d = info.distance + e.weight;

if (!dist.contains(v2.index)) {

float h = m\_heuristic.calculate(v2, goal);

dist.insert(v2.index, new Info(d, h));

previous.put(v2.index, v);

} else {

Info oldInfo = dist.getPriority(v2.index);

if (oldInfo.distance > d) {

oldInfo.distance = d;

dist.changePriority(v2.index, oldInfo);

previous.put(v2.index, v);

}

}

}

}

if (!goalAchieved)

return null;

// get optimal path

Graph.Vertex v = goal;

List<Graph.Vertex> path = new ArrayList<Graph.Vertex>();

path.add(goal);

while (v.index != start.index) {

v = previous.get(v.index);

path.add(v);

}

Collections.reverse(path);

return path;

}

}