

# Летняя практика ФКН по path finding

Ветров Андрей

лето 2019

В этой работе я буду разбирать некоторые алгоритмы автоматического планирования траектории на плоскости. В частности, начну я с алгоритма  $A^*$ .

## 1 $A^*$

### 1.1 Постановка задачи

Допустим, у нас есть некое поле, предварительно разбитое на узлы, между которыми можно перемещаться и время перемещения между узлами известно. Также у нас имеется агент (например, робот). Ему необходимо переместиться из одного узла в другой, затратив при этом наименьшее количество времени. Иными словами, дан взвешенный граф и две вершины в нем, для которых надо явно построить путь наименьшей стоимости между ними. Стоит отметить, что очень часто необходимо решить задачу в том случае, когда поле представляет из себя прямоугольник, содержащий некоторое количество препятствий, через которые нельзя построить путь. И ходить в этом прямоугольнике можно только в восьми направлениях. Поэтому дальше я буду отталкиваться именно от этого случая.

### 1.2 Поиск кратчайшего пути

Представим, что между данными нам вершинами  $a$  и  $b$  мы нашли кратчайший путь. Получилась некая последовательность вершин  $v_0, v_1, \dots, v_n$ , где  $v_0 = a$  и  $v_n = b$ . Заметим, что для каждой вершины  $v_i$  из этой цепочки кратчайший путь от  $a$  до  $v_i$  как раз состоит из предыдущих элементов

этой последовательности. Действительно, если существует путь ещё короче, то мы можем улучшить ответ для вершины  $b$ , связав путь от  $a$  до  $v_i$ , и от  $v_i$  до  $b$ .

Это утверждение лежит в основе всех алгоритмов поиска кратчайшего пути в графах. Мы строим цепочку вершин кратчайшего пути от первой к последней. При этом кратчайший путь для точки вычисляется при помощи соседей, для которых мы уже узнали ответ. Мы перебираем кандидатов на роль родителя (сына) текущей вершины - наилучший и пойдет в ответ. Здесь и кроется самая большая проблема поиска кратчайшего пути - с каких вершин начать перебор, чтобы найти ответ как можно быстрее? Рассмотрим две парадигмы подхода к перебору.

### 1.3 Жадный перебор

Допустим, мы хотим найти кратчайший путь от точки А до точки В. Сейчас агент находится в стартовой позиции и пора решать, куда сделать первый шаг. Логично предположить, что если точка В расположена справа-выше точки А, то начать кратчайший путь может быть выгодно ходом вправо-вверх. Если же пойти в эту сторону не получается, то можно попробовать варианты пойти вправо или вверх. В противном случае переходим к вариантам вправо-вниз и влево-вверх и так далее, пока у нас не закончатся варианты. Для вершины, в которую мы переходим, повторим то же самое, с поправкой на изменение направления. Иными словами, мы идем в сторону цели, если это возможно.

Понятное дело, такой способ построения пути не найдет кратчайший путь, он всего лишь находит траекторию, близкую к правильной. Однако при малом количестве препятствий на пути между точками построенный путь почти не отличается от кратчайшего. К хорошему преимуществу этого способа можно отнести то, что при переборе мы почти не рассматриваем лишние вершины - алгоритм работает очень быстро. Почему так происходит - несколько позже.

### 1.4 Алгоритм Дейкстры

Другой подход использует алгоритм Дейкстры. Он последовательно находит все кратчайшие пути от стартовой вершины до всех остальных (или пока не встретим финиш), начиная с ближайших и заканчивая самой дальней. На каждой итерации алгоритм считает ответ для ближай-

шей к старту ещё не обработанной вершины, используя уже посчитанные величины путей до соседей.

Понятно, что алгоритм Дейкстры находит кратчайшие пути до всех вершин в компоненте связности графа. Действительно, при обработке каждой вершины все элементы кратчайшего пути, кроме последней, уже посчитаны, следовательно алгоритм найдет ответ для этой вершины не больше правильного. Так как значения длины кратчайшего пути корректны, то мы получим именно кратчайшие пути.

Однако стоит заметить, что алгоритм считает много лишней информации, а именно длину пути до всех посещенных вершин. В связи с этим время работы этого алгоритма может быть неоправданно большим, так мы можем перебрать все вершины графа, что конечно неэффективно. Давайте разберем специальный алгоритм для нахождения расстояния только для одной вершины, работающий гораздо быстрее.

## 1.5 Основная идея A\*

И состоит она в том, что мы можем объединить две описанные выше идеи в что-то более эффективное. Для начала давайте для каждой вершины введем некую функцию  $h(x)$ , ассоциированную с предполагаемым расстоянием до финиша и обладающей следующим свойством: для каждой вершины это расстояние не превосходит длину реального кратчайшего пути.

В частности, если наше поле имеет форму прямоугольника с возможностью ходить по свободным клеткам с константной скоростью 1, то для  $h(x)$  подойдет метрика - евклидова или манхэттенское расстояние.

Теперь вернемся к алгоритму Дейкстры. Раньше на каждом шаге мы выбирали в качестве следующей вершины на обработку  $\min(g(x))$ , где  $g(x)$  - это расстояние от старта до точки  $x$ . Теперь же выберем  $\min(g(x) + h(x))$ . Заметим, что в силу свойства функции  $h(x)$ , ответ для финишной вершины будет в точности равен длине кратчайшего пути. В то же время алгоритм достигнет финишной вершины гораздо раньше, чем в алгоритме Дейкстры, так как вершины ближе к финишу имеют для него больший приоритет.

## 1.6 Оценка расстояния до финиша

В идеале нам бы хотелось уметь быстро вычислять такую функцию  $h(x)$ , которая была бы в точности равна расстоянию от вершины до финиша. В таком случае все вершины, лежащие хотя бы на одном из кратчайших путей будут иметь минимальный потенциал, а все остальные вершины будут стоять после них. К сожалению, не существует каких-то универсальных способов оценить остаточное расстояние, лучше, чем, например, евклидова метрика (за исключением, может быть, метода четырех русских - но он будет работать только на тех картах, на которых количество запросов превышает количество узлов). В то же время мы можем оценить количество препятствий на пути от вершины до финиша по "плотности" распределения препятствий в этом секторе, но опять же такие эвристики не универсальны. Потому  $A^*$  всё-таки просматривает некоторое количество неоптимальных вершин.

## 1.7 Оптимизация

Ещё одна проблема заключается в том, что от старта до финиша может вести сразу несколько кратчайших путей. Все их звенья будут иметь одинаковый потенциал  $g(x) + h(x)$  - и потому, когда мы будем по очереди доставать их из списка открытых вершин, мы можем продвигаться не вглубь, а в ширину, что нас не устраивает. Решить это можно двумя способами:

- Во-первых, можно умножать функцию  $h(x)$  на некую константу, незначительно превышающую единицу. При этом константа не должна превосходить  $\frac{1}{\max(h(x))}$ , чтобы все сравнения различных по значению элементов сохранились. В то же время равные потенциалы будут сравниваться по  $h(x)$  - и в первую очередь алгоритм выберет те, что ближе к цели. В итоге мы с высокой долей вероятности рассмотрим только один кратчайший путь.
- Другой способ - мы можем явно задать сравнение двух вершин по их  $g(x)$  и  $h(x)$ . Сначала мы сравниваем их по сумме, а затем по  $h(x)$ . Так как здесь нет проблем с выбором константы, я выбрал именно этот способ.

## 1.8 Псевдокод

Теперь можно привести псевдокод алгоритма A\*.

```
Определяем h(x)
Определяем компаратор для priority queue
Q = empty priority queue
previous = empty hash map
ready = empty set
Q.push(start)
dist[start] = 0
while not Q.empty
    <current, pot> = Q.pop
    if current = finish
        break
    ready.insert(current)
    for next : current.neighbours
        path_length = dist[current] + r(current, next)
        if next in Q and path_length < dist[next]
            Q.remove(next) \\ сейчас мы добавим его ещё раз
        if not(next in Q) and not(next in ready)
            dist[next] = path_length
            Q.push(next, dist[next] + h(next))
            previous[next] = current
\\ восстанавливаем путь
current = finish
while current != start
    path.push_front(current)
    current = previous[current]
```

Все вершины делятся на три типа. Первый тип - для них уже известен путь и его длина - это хешмап ready. Второй тип - это кандидаты на текущее расширение области посчитанных ответов - это приоритетная очередь Q. Остальные вершины находятся как минимум в двух ребрах от вершин первого типа.

## 1.9 Реализация

Пока реализация состоит из 5 файлов:

- `field.h` - здесь содержится код класса `Field`, т. е. поля, на котором мы будем искать кратчайшие пути + функция ввода для него.
- `path.h` - в процессе работы алгоритм найдет путь. Он будет храниться в виде класса, описанного здесь.
- `a-star.h` - основной класс, решающий задачу.
- `print.h` - вывод поля и найденного пути.
- `main.cpp` - здесь всё собирается и запускается.

Для запуска вроде достаточно запустить `main.cpp`. Пока весь ввод-вывод происходит из текстовых файлов. Насколько я понимаю, все корректно работает на любых картах, и путь получается действительно кратчайшим.