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

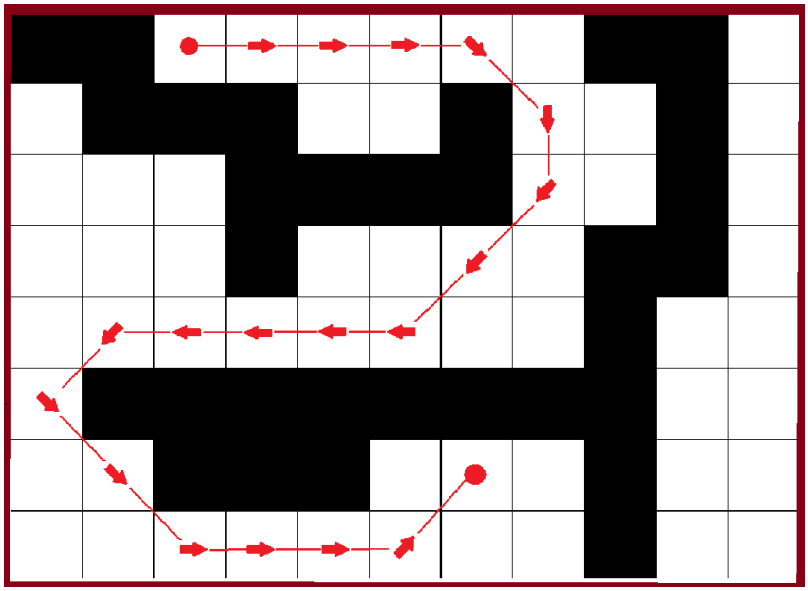

Рис. 1: Пример пути в grid-поле

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 $v_0, v_1, ...v_n$, 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 A до точки B. Сейчас агент находится в стартовой позиции и пора решать, куда сделать первый шаг. Логично предположить, что если точка B расположена справа-выше точки A, то начать кратчайший путь может быть выгодно ходом вправо-вверх. Если же пойти в эту сторону не получается, то мож-

но 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 отнсти то, что при переборе мы почти не рассматриваем лишние вершины - алгоритм работпет 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 ближайшей к старту ещё не обработанной вершины, используя уже посчитаные величины путей до соседей.
Понятно, что алгоритм Дейкстры находит кратчайшие пути до всех вершин в компоненте связанности 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 путь 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 кратчайшго 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 свойсва 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 $h(x)$, котороя была бы в точности равна расстоянию от вершины до финиша. В таком случае все вершины, лежащие хотя бы на одгом из кратчайших путей будут иметь минимальный потенциал, а все остальные вершины будут стоять после них. К сожадению, 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 может 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 будет 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 функция $h(x)$ 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
OPEN = empty priority queue of nodes
CLOSED = empty set of nodes
add start to OPEN
while not OPEN.empty
    node current = Q.top()
    remove current from OPEN
    if current = finish
        break
    add current to CLOSED
    for next : current.neighbours
        path_length = current.dist + r(current, next)
        if next in OPEN and path_length < next.dist
            remove next from OPEN // сейчас мы добавим его ещё раз
        if not next in OPEN and not next in CLOSED
            next.parent = current
            next.dist = path_length
            add next to OPEN
```

```
// восстанавливаем путь
current = finish
while current != start
    path.push_front(current)
    current = current.parent)
```

Все вершины делятся на три типа. Первый тип - для них уже известен путь и его длина - они содержатся в CLOSED. Второй тип - это кандидаты на текущее расширение области посчитанных ответов - это приоритетная очередь OPEN. Остальные вершины находятся как минимум в двух ребрах от вершин первого типа.
Алгоритм последовательно расширяет область посчитанных ответов, начиная со старта и пока не дойдет до цели. На каждой итерации выбирается наиболее подходящая вершина из кандидатов (она имеет наименьший f-value, потому является первой в приоритетной очереди OPEN). Из неё мы строим кратчайший путь до её соседей, если мы ещё не сделали этого, например, из ещё из какой-нибудь вершины). При этом мы прописываем текущую вершину, как родителя соседа.

1.9 Реализация

Пока реализация состоит из 6 файлов:

- Map.cpp - здесь содержится код класса Field, т. е. поля, на котором мы будем искать кратчайшие пути + функция ввода для него.
- Path.cpp - в процессе работы алгоритм найдет путь. Он будет храниться в виде клласса, описанного здесь.
- Algo-body.cpp - основной класс, решающий задачу.
- Print.cpp - вывод поля и найденного пути.
- main.cpp - здесь всё собирается и запускается.
- Collection.h - header.

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.

2 Theta*

2.1 Постановка задачи

Для начала расширим поставленную задачу. Теперь карта представляет прямоугольник размера $w \times h$. Наш агент (робот) может переходить от одной целочисленной точки к другой, если такой переход корректен, т. е. робот не пересечет никакое препятствие. Препятствия - это произвольные многоуголиники на плоскости. Задача: построить путь из точки A в точку B, близкий к кратчайшему.

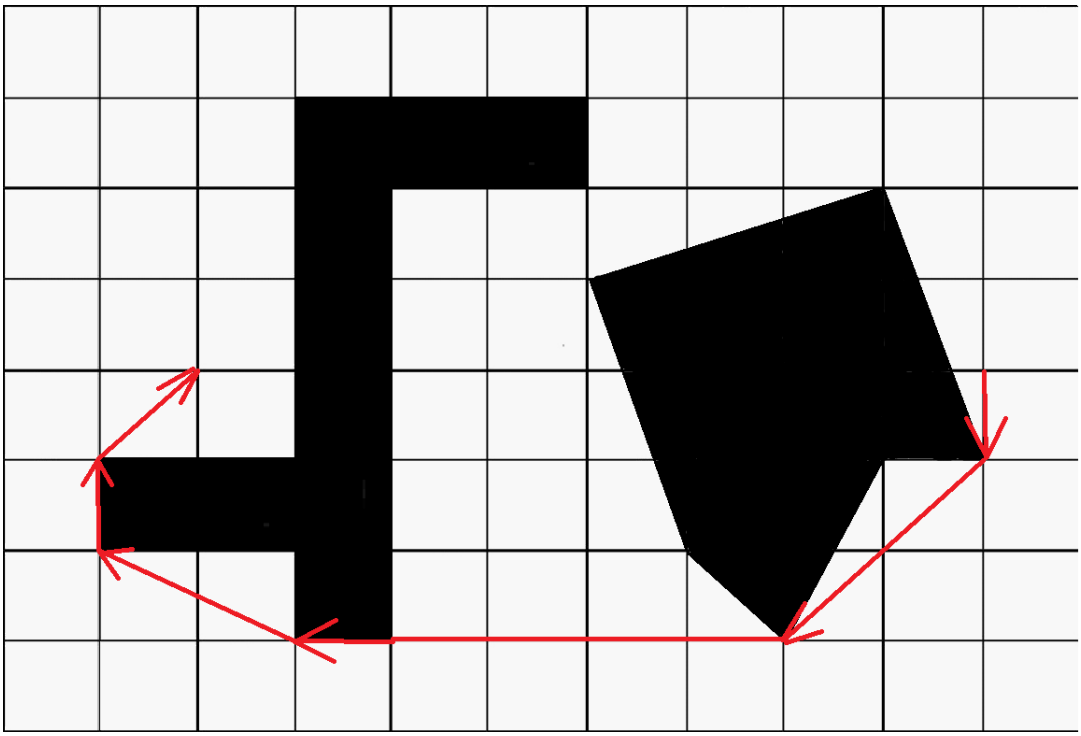

Рис. 2: Пример пути в net-поле

2.2 A*

Начнем решать эту задачу с помощью уже известного алгоритма A* на grid-поле. Заменим элементы старого поля на точки у нового. Таким образом, мы получаем некий путь из точки A в точку B, проложенный по

целочисленным координатам. Но в то же время этот путь сильно длиннее кратчайшего. По большей части эта разница заключается в том, что мы можем ходить только по 8 направлениям, потому найденная траектория получается очень угловатой. Рассмотрим пример:

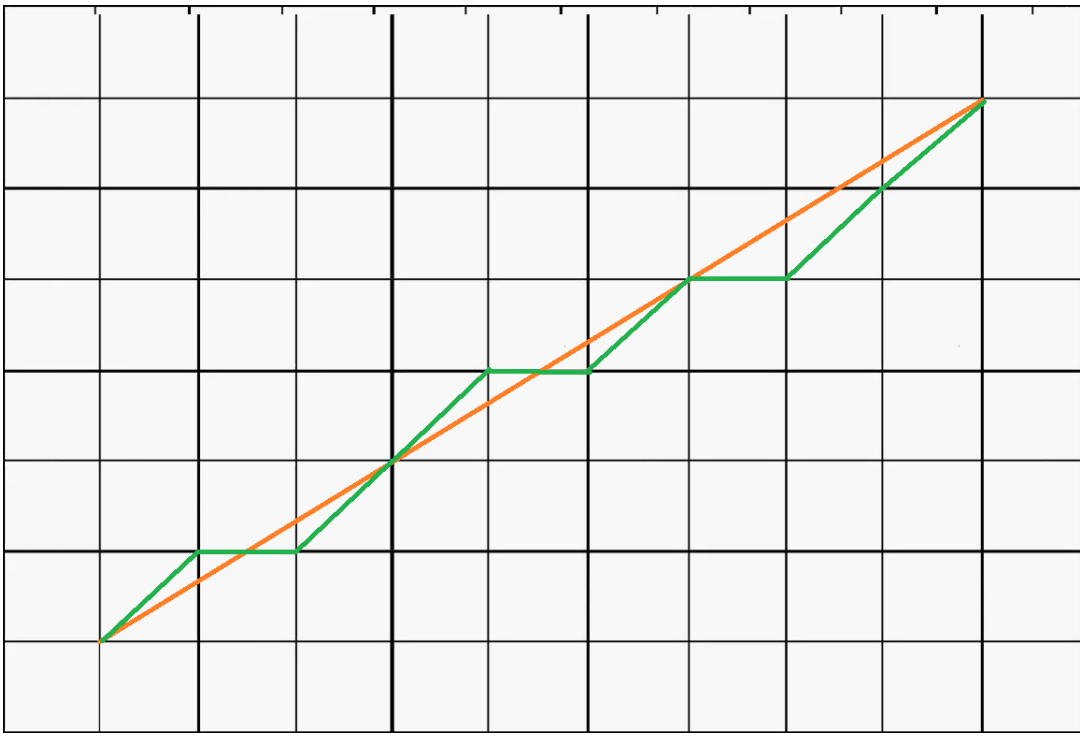

Рис. 3: A* vs Кратчайший путь

Конечно же, имеет смысл уметь спрямлять такие части траектории, чтобы приблизить ответ к идеальному.

2.3 Theta*

Будем спрямлять путь в процессе построения траектории алгоритмом A*. Рассмотрим одну из первых его итераций:
Алгоритм Theta* предоставляет возможность построить путь к новому соседу напрямую от родителя текущей вершины, минуя текущую, если такой переход корректен (см. рис. 4). Таким образом, к стартовой вершине могут быть привязаны в качестве детей не только её непосред-

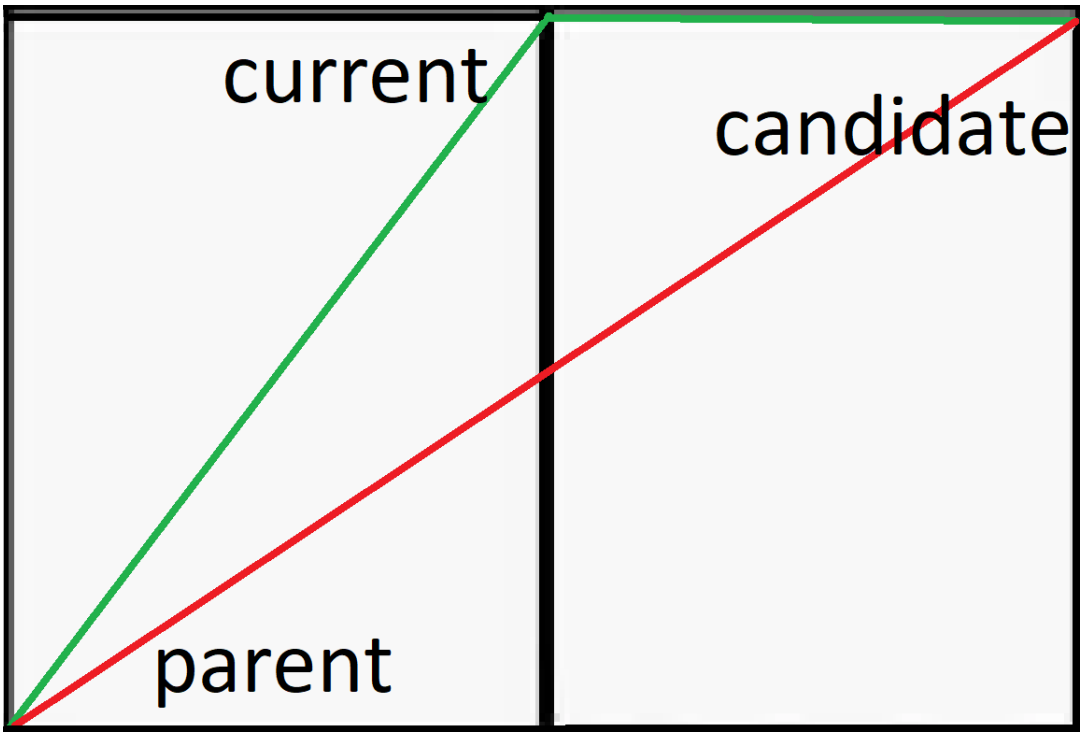

Рис. 4: Основная идея Theta*

ственные соседи, но и все вершины, достижимые одним переходом. Таким образом, путь практически полностью сглаживается, а значит, построенный путь близок к кратчайшему.
На самом деле, ситуация, при которой построенный с помощью Theta* путь не является кратчайшим, относительно редкая, а если полученный путь не идеальный, то относительная разница длины гораздо меньше 10% (при достаточно большой дляне пути, конено). Такри образом, Theta* является очень сильным, и в то же время относительно быстрым алгоритмом.

2.4 Псевдокод

Псевдокод алгоритма Theta* почти не отличается от A*. Здесь добавляется только случай перехода от родителя текущей вершины к новому соседу.

```
Определяем h(x)
Определяем компаратор для priority queue
OPEN = empty priority queue of nodes
CLOSED = empty set of nodes
add start to OPEN
while not OPEN.empty
    node current = Q.top()
    remove current from OPEN
    if current = finish
        break
    add current to CLOSED
    for next : current.neighbours
        real_parent = current
        path_length = current.g_value + r(current, next)
        if jump from current.parent to next is correct
            real_parent = current.parent
            path_length = current.g_value + r(current.parent, next)
        if next in OPEN and path_length < next.g_value
            remove next from OPEN // сейчас мы добавим его ещё раз
        if not next in OPEN and not next in CLOSED
            next.parent = real_parent
            next.g_value = path_length
            add next to OPEN
// восстанавливаем путь
current = finish
while current != start
    path.push_front(current)
    current = current.parent)
```