Конечные автоматы

Введение

Конечный автомат - ориентированный граф. На рёбрах находятся символы и существуют начальные (стартовые) и конечные (терминальные) вершины.

Почему *конечный* автомат? потому что количество состояний конечное

Виды автоматов

Детерминированный конечный автомат (DFA, ДКА)

- На каждый символ из каждого состояния ровно один переход
- Нет e - переходов (пустых)
- Работает за O(n) времени и O(1) по памяти на символ

пример: найти все бинарные строки, где чётное число нулей
Два состояния - чёт и нечёт
Начальное - чёт (0 нулей)
Принимающее - чёт

Недетерминированный конечный автомат (NFA, НКА)

- на один символ может быть несколько переходов
- Есть e - переходы (переход без символа)
- Может "размножаться" и проверять все пути одновременно

пример: regex сначала преобразуется в NFA, только потом превращается в DFA

Где используется?

Где	Какой атвомат	Зачем
grep -E, egrep, pcre	DFA (ДКА)	ищет регулярку за один проход по тексту

Где	Какой атвомат	Зачем
flex (лексер)	DFA	разбивает код 100 МБ на токены за 0.3 сек
Wi-Fi роумино, TCP - соединение	DFA	состояние протокола (SYN_SENT -> ESTABLISHED)
Контроллеры и микроконтроллеры	Moore/Mealy	Светофор, кофе-машина
Парсинг JSON, XML	DFA	Валидация за линейное время
Антивирусы (сигнатуры)	DFA (Ахо-Корасик)	ищет множество вирусов по сигнатурам одновременно
Граф состояний (геймдев, прогресс квеста)	DFA, NFA	Поддерживает различные состояния

regex

допустим есть regex

```
(https?://)?([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}
```

1. Thompson’s construction - строит NFA (около 100 состояний)
2. Powerset construction - превращает NFA в DFA (может быть до 2^100 состояний теоретически, но на практике 200-500)
3. Минимизация Холкрофта - сжимает DFA до минимального (+- 50 состояний)
4. Получаем таблицу переходов 50 x 256 = 12 КБ

Префиксное дерево (Бор, trie)

Удобный способ хранить множество строк
Операции:

1. Добавить строку в множество
2. Удалить строку из множества

3. Проверить что строка есть в множестве
   Все операции за O(n), где n - длина строки
   По памяти на практике меньше, чем хранить все строки

Будем поддерживать множество вершин и определим стартовую вершину. Конечными вершинами сделаем те вершины, в которых кончаются строки множества.
Введём соответствие строка (префикс строки) <-> вершина.

Пример:
aab,
abba,
bbab,
bbbab

Реализация

Если размер алфавита большой, тогда вместо массива храним unordered_map или vector переходов

```
const int k = 26;

struct Vertex {
    Vertex* to[k] = {0};
    bool terminal = 0;
};

Vertex *root = new Vertex();
```

```
void add_string(string& s) {
    v = root;
    for (char c : s) {
        c -= 'a';
        if (!v -> to[c]) // вершины ещё нету, значит создаём
            v -> to[c] = new Vertex();
        v = v -> to[c];
    }
```

```
    v -> terminal = true;
}
```

```
bool check_string(string& s) {
    v = root;
    for (char c : s) {
        c -= 'a';
        if (!v -> to[c]) // вершины нету - значит не добавляли
            return false;
        v = v -> to[c];
    }

    return v -> terminal;
}
```

Цифровое префиксное дерево

Представим каждое число в виде битовой записи одинаковой длины и будем работать с ними как со строками.

Дополнительные операции:

1. Поиск k-ого числа по возрастанию
2. Поиск суммы элементов, меньших x

Пример:

Ахо-Корасик (префиксное дерево + суфф ссылки)

Альфред Ахо и Маргарет Корасик, 1975 год.

Введение и постановка задачи

Пусть даётся множество строк S (словарь) и мы хотим уметь быстро искать эти строки в одной строке T.

Примеры такой задачи:

- grep
- Поиск запрещённых слов в тексте
- Поиск сигнатуры вирусов в программном коде

Коротко, но сухо

Задача:

пусть дан набор строк $s_1, s_2, s_3, \ldots, s_n$, называемый словарём и большой текст t. Необходимо найти все позиции, где строки словаря входят в текст. Для простоты дополнительно предположим, что строки из словаря не являются подстроками друг друга (позже мы увидим, что это требование избыточно).

Решать задачу будем следующим образом. Будем обрабатывать символы текста по одному и поддерживать наибольшую строку, являющуюся префиксом строки из словаря, и при этом суффиксом считанного на данный момент текста. Если эта строка совпадает с каким-то s_i, то отметим текущий символ - в нём заканчивается какая-то строка из словаря.

Для этой задачи нужно эффективно хранить и работать со всеми префиксами слов из словаря - для этого используем префиксное дерево.

Добавим все слова в префиксное дерево и пометим соответствующие им вершины как терминальные. Теперь наша задача состоит в том, чтобы при добавлении очередного символа быстро находить вершину в префиксном дереве, которая соответствует наидлиннейшему входящему в бор суффиксу нового выписанного префикса. Для этого давайте введём несколько вспомогательных понятий.

Анти-формализм. Давайте отождествлять вершину и соответствующую ей строку в боре.

Определение. *Суффиксная ссылка* $l(v)$ ведёт в вершину $u \neq v$, которая соответствует наидлиннейшему применяемому бором суффиксу v. Часто можно встретить название "fail - ссылка" или "fail-link"

Определение. Автоматный переход $\delta(u, c)$ ведёт в вершину, соответсвующую минимальному применяемому бором суффиксу строки $u + c$. Если переход и

так существует в боре, то автоматный переход буде твести туда же.

Автоматные переходы - это именно то, что нам и надо в задаче: они ведут ровно в те вершины, которые соответсвуют самому длинному "сматченному" суффиксу.

Заметим следующую связь суффиксных ссылок и автоматных переходов:

- $l(s_{:n}) = \delta(l(s_{:n-1}), s_n)$
- Если прямого перехода $v \to_c u$ не существует, то $\delta(v, c) = \delta(l(v), c)$

Мы выразили l и δ от строки длины n через l и δ от строк размера $(n-1)$. Значит суффиксные ссылки и автоматные переходны можно найти динамическим программированием.

Реализация

По сравнению с префиксным деревом, нам помимо массива переходов в боре `to` нужно будет хранить некоторую дополнительную информацию:

- Сам массив автоматных переходов `go`.
- Суффиксную ссылку `link`
- "Родительский" (последний) символ `pch`, который используется в формуле для суффиксной ссылке

Длинно, но понятно

Хотим построить автомат, который принимает только строки s_i и ничего больше.
S:
aab,
abba,
bbab,
bbbab

План:
Будем идти по строке T слева направо и искать вхождения строк из словаря S

Допустим, считали i символ и уже обработали все вхождения которые заканчиваются левее (конец вхождения $< i$. Пока что не думаем о вхождениях, которые начинаются правее (начало вхождения $> i$).

Пример:
T = aabbabbabab.....

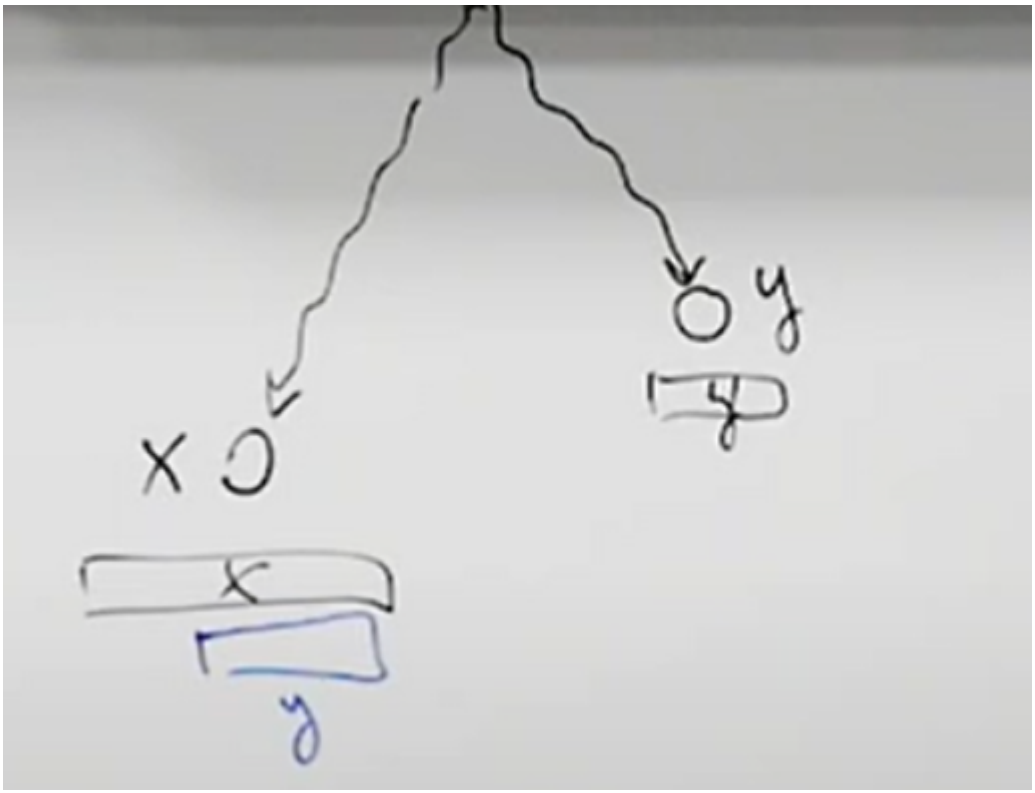
Обработали все вхождения левее последней b. Все ли символы нам нужно помнить из этой строки? на самом деле нет. Только последние 2 - ab. именно с них начинаются какие-то строки из словаря.
Если дальше идёт b, то нас будет интересовать abb (это префикс abba из словаря)
Если дальше a - нас будет интересовать только последняя a

Не стоит хранить эти символы в виде строки. Интересующий нас суффикс считанного текста - префикс какого-то слова из множества S. Этот префикс легко поддерживать в префиксном дереве.

Есть текущее состояние в префиксном дереве. Если по следующей букве можно сделать переход - переходим. Если нет - надо отрезать какое-то начало от текущего суффикса.

допустим в конце aba
Отрезаем a, но вершины ba нету. Отрезаем b, строка a есть - это именно та вершина, куда мы хотели бы попасть.

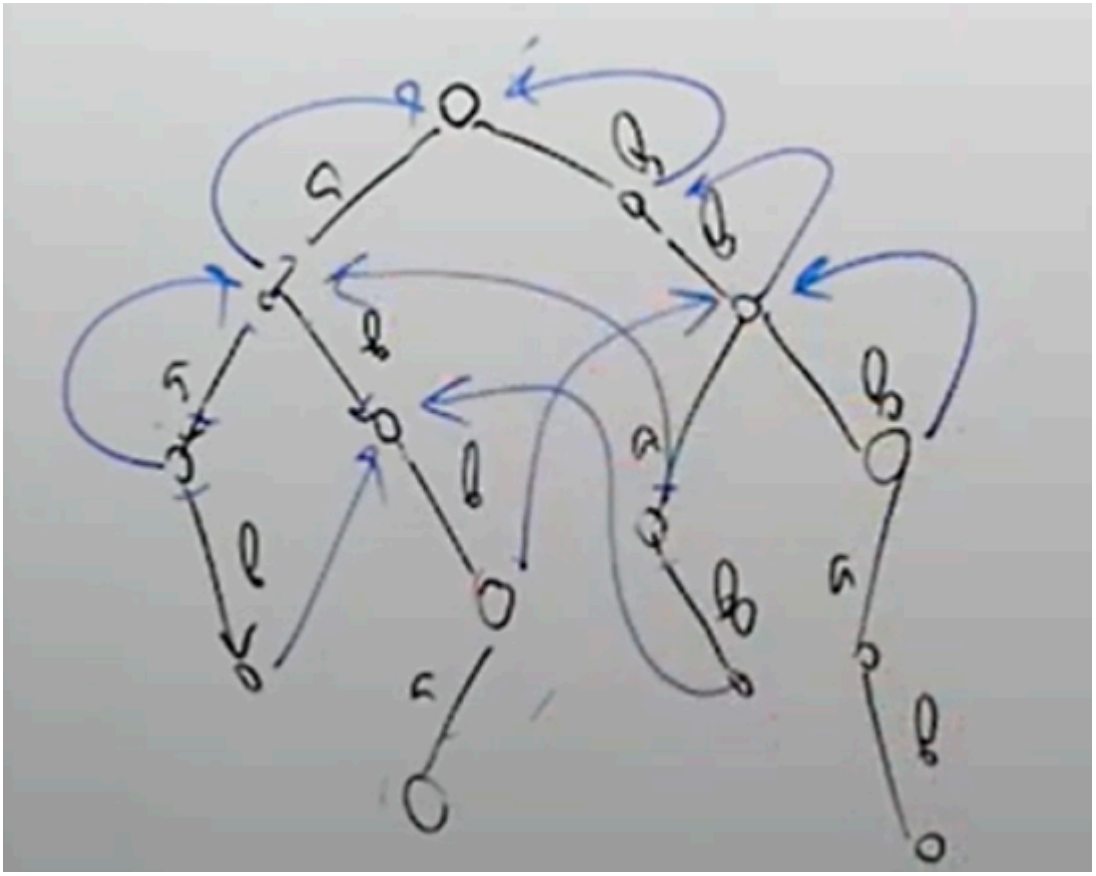
Помним в какой вершине находимся и смотрим на следующую букву. Если переход есть - идём. Иначе нужно найти подходящий префикс



Чтобы такое уметь делать, нужно быстро отрезать символы слева для перехода в другой префикс.
Мы находимся в x и хотим попасть в такой y, что он наибольший.

Сделаем ссылку $x => y, y \neq x, |y| => max$
Это называется суффиксной ссылкой. y - максимальный суффикс x, который является чьим-то префиксом.

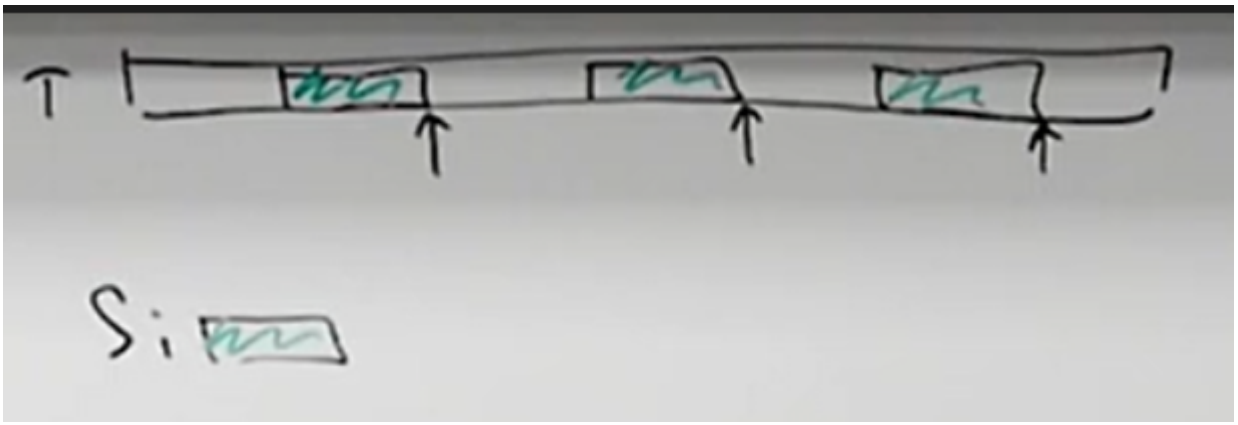
Допустим, для каждого состояния нашли такую суффиксную ссылку. У каждого состояния такая ссылка только одна, за исключением корня



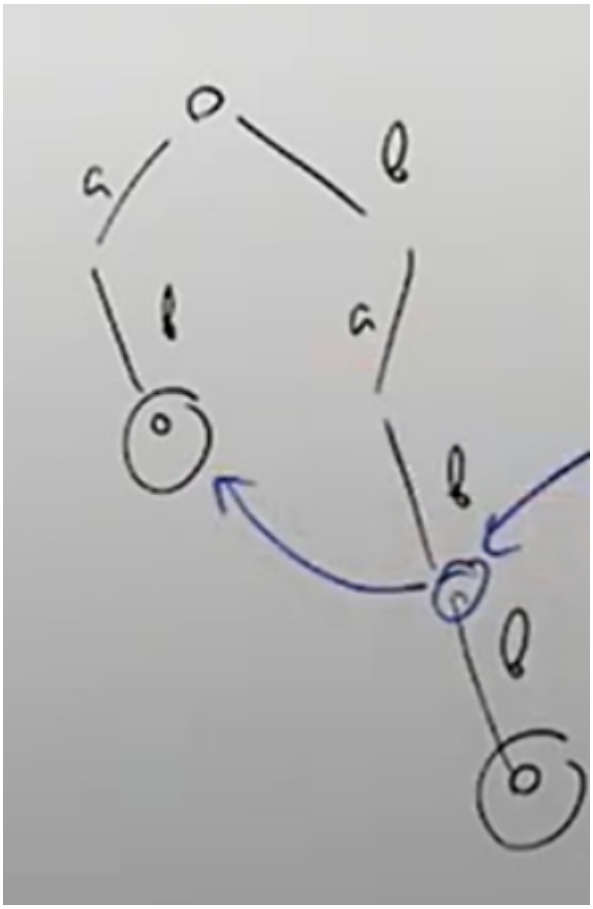
Утверждается, что, имея такие ссылки и проходя по переходам, время обработки строки T будет линейным.
Доказательство:
Поддерживаем l, r - отрезок, $l \leq r$ который в данный момент помним из T. При считывании нового символа, увеличиваем r на 1 (r -> r + 1). Отрезая символы, мы двигаем левую границу l. l и r только увеличиваются. Каждую границу мы можем увеличить не более |T| раз.

Задача

Дано множество строк S (словарь) и строка T. Хотим проверить, какие строки из S входят в T как подстроки.



В момент считывания конца каждого вхождения находились в каком-то состоянии. В состоянии, короче чем s_i быть не могли. Но как понять, что мы, находясь в bab, видели ещё и ab?



из bab можно будет добраться до ab по суффиксным ссылкам, т.к. ab является каким-то суффиксом bab. Мы встретили строку s_i, если мы в какой-то момент времени посетили состояние v такое, что существует путь из v в s_i по суффиксным ссылкам.
Для решения поставленной задачи можно поддерживать в каких состояниях мы были во время обработки строки T. После обработки T нужно запустить обход графа, построенного на суффиксных ссылках (и только на них)

3 основных этапа:

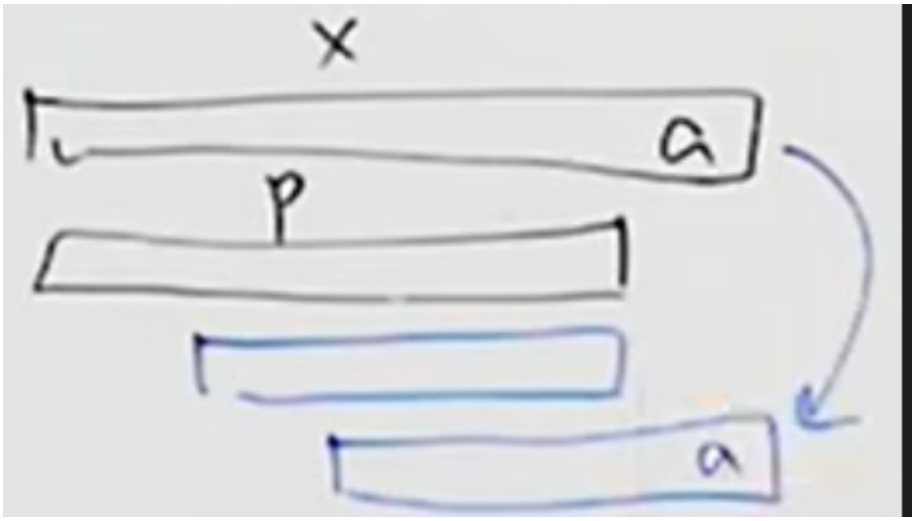
1. Построить автомат с суффиксными ссылками
2. Обработать строки и насчитать статистику (количество посещений состояния или были/не были)
3. Обработать полученную статистику.

Построение автомата

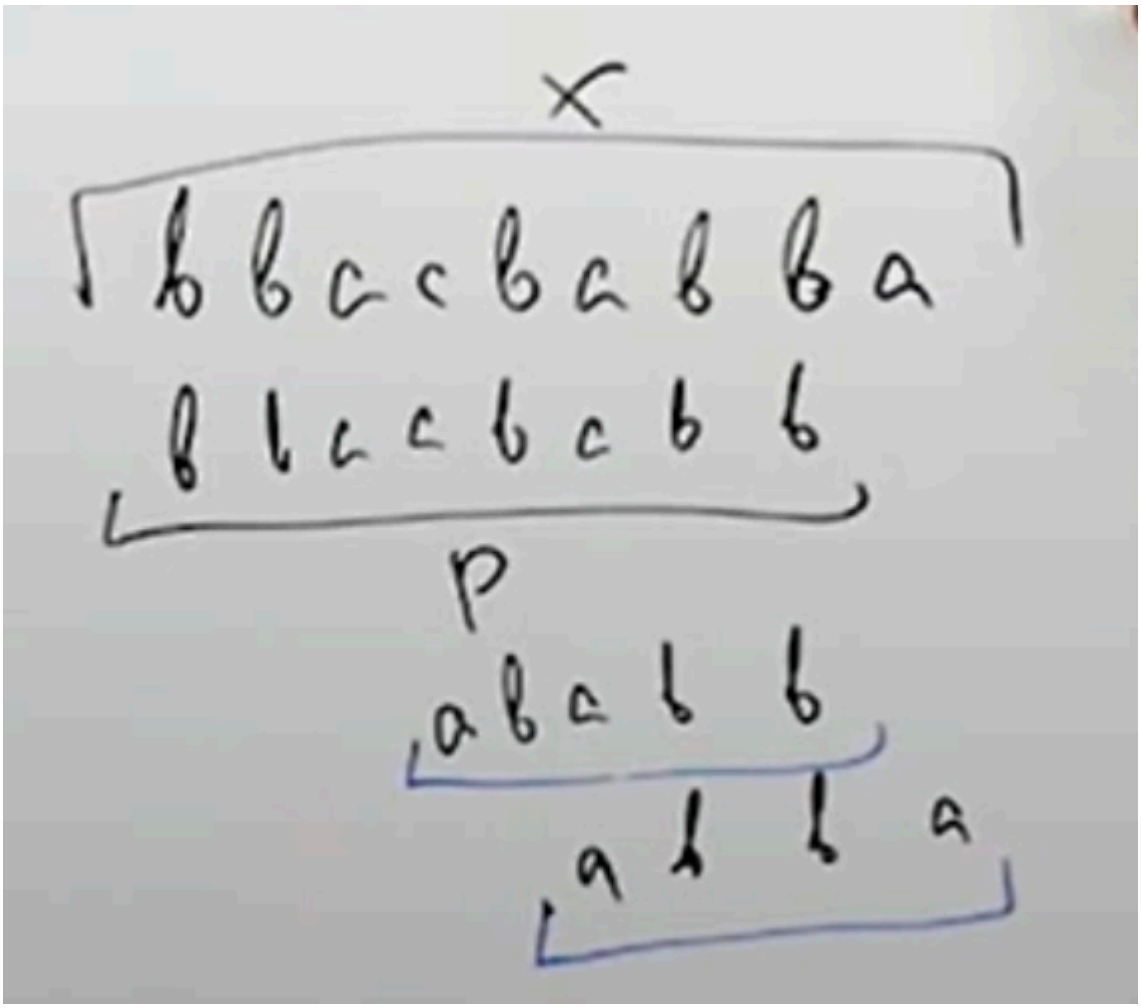
Построим префиксное дерево слов из множества S и дополним всеми переходами, которых не хватает (из каждого состояния будет переход по каждому символу).
Обработаем префиксное дерево обходом в ширину и одновременно будем считать и суффиксные ссылки, и автоматные переходы.
Допустим, мы сейчас находимся в состоянии x и хотим посчитать автоматный переход по букве a. Если такой переход есть в префиксном дереве, то он и будет автоматным переходом иначе, нужно отрезать какой-то префикс x до ближайшего (наибольшего) допустимого суффикса.

```
x.move['a'] = (x -> suff).move['a']
```

В случае, если суффиксные ссылки будут посчитаны, то автоматные переходы мы сможем посчитать.
Для поиска суффиксной ссылки посмотри на строку x. Хотим найти наибольший суффикс, который есть в префиксном дереве. Для этого посмотрим на последний символ x и отрежем его, т.е. посмотрим на родителя вершины x в дереве. От этого родителя возьмём суффиксную ссылку и попробуем сделать переход по последнему символу.




```
x.suff = x.parent.suff.move['a']
```



Все суффиксы короче x. В случае, если мы будем считать ссылки и переходы в порядке возрастания длин, то всё будет корректно.

```
p = x.parent
a = x.last
k = p.suf
while k.children['a'] = Null
    k = k.suf
x.suf = k.children['a']
```

Переходим по суфф ссылкам, пока не найдём подходящий суффикс, у которого есть переход по символу a
Если такого суффикса нет, то суффиксная ссылка = корень дерева.
Алгоритм Кнута - Морриса - Пратта - это Ахо-Корасик на множестве из одной строки.

```
// (именно так в GNU grep)
```

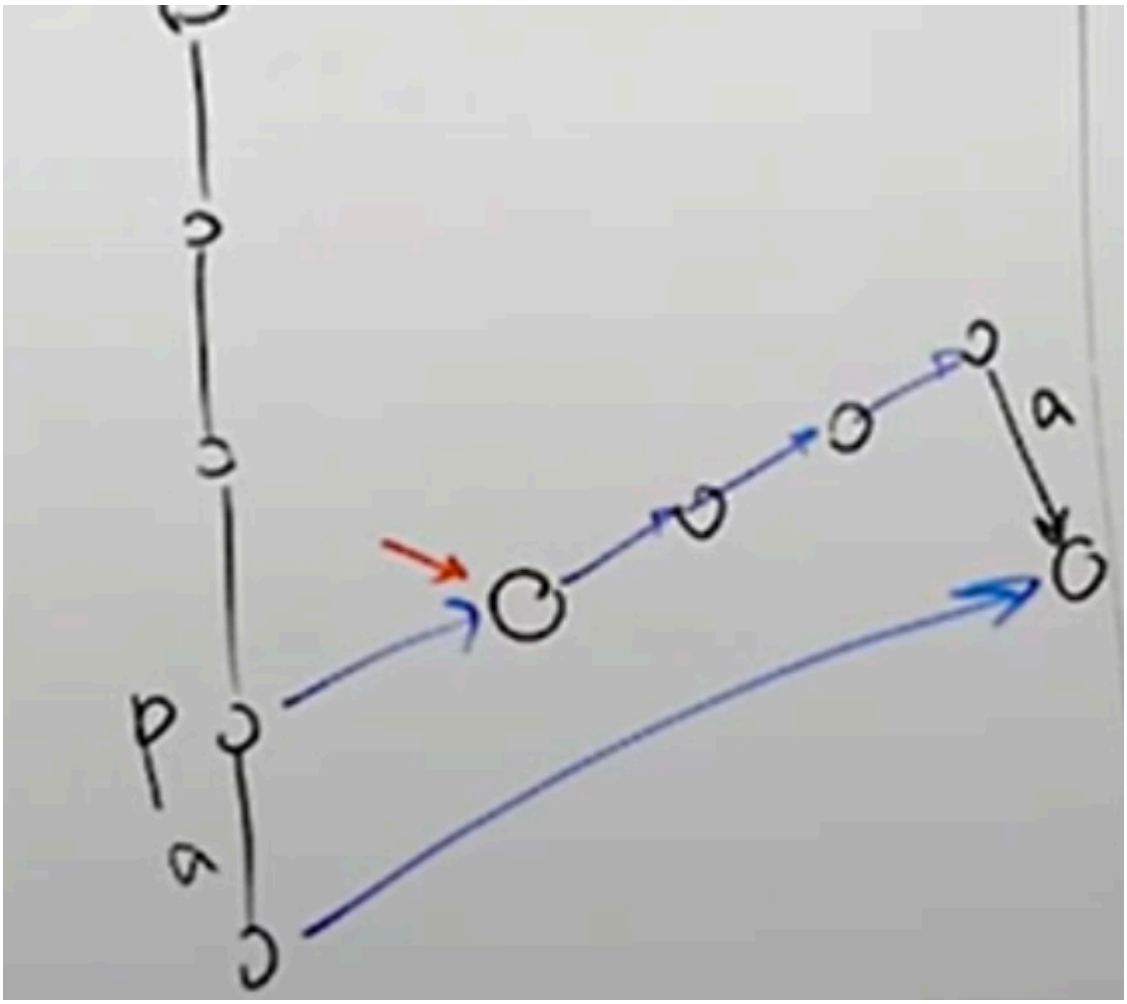
```
cur->fail = (cur->parent->fail)->go[char];
```

«Это ровно как префикс-функция в KMP, только вместо массива — дерево. Мы идём по родителям, пока не найдём того, у кого уже есть переход по этой букве.»

```
ver: "he" → parent "h" → fail от "h" это root → root->go['e'] = "e"
→ fail("he") = "e"
```

Время работы

Сложность по времени будет O(n), где n - количество вершин дерева. Работает за линейное время по тому же принципу, по которому построение префикс функции (и алгоритм Кнута - Морриса - Пратта) работает за линию.
Чем дольше делаем while, тем меньше строка (двигается правая граница)



Левый конец двигаем вправо не более, чем n раз,, правый конец не более n раз, итого суммарно линия.