Теоретическая информатика II
Лекция 2: Поиск в строке: алгоритм Кнута–Морриса–Пратта. Конечные автоматы, реализация алгоритма Кнута–Морриса–Пратта, алгоритм Ахо–Корасик. Суффиксные деревья*

Александр Охотин

15 апреля 2017 г.

Содержание

1 Алгоритмы поиска в строке (продолжение)

1.1 Алгоритм Кнута–Морриса–Пратта

Улучшение «наивного» поиска. Пусть в строке $w = ababaab$ ищется подстрока $x = abaa$. Сперва $w_1 = abab$ сравнивается с $abaa$ и выясняется, что они различны в третьем символе. Имеет ли после этого смысл сравнивать $w_2 = baba$ с $abaa$? Никакого, потому что раз предыдущая подстрока $w_1 = abab$ отличается от x лишь в четвёртом символе, а первый и второй символы x различны, это значит, что первый символ w_2 не совпадает с первым символом x. Эти сведения можно получить исключительно из самой строки x и количества совпадающих первых символов у w_1 и x (3). Поэтому алгоритм может сразу переходить к сравнению w_3 с x. Более того, поскольку первый и третий символ x совпадают, из результата первого сравнения уже известно, что первый символ w_3 совпадает с первым символом x, и потому их можно сравнивать, начиная со вторых символов.

*Краткое содержание лекций, прочитанных студентам СПбГУ, обучающимся по программе «математика», в весеннем семестре 2016–2017 учебного года. Без посещения самих лекций в этом едва ли что-то возможно понять.

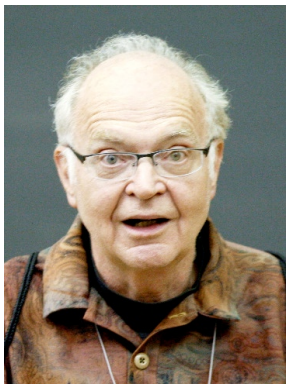


Рис. 1: Дональд Кнут (род. 1938).

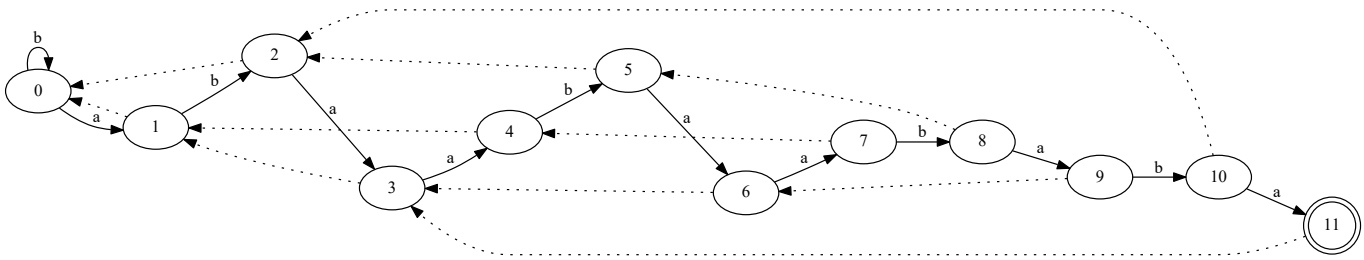
Подобные рассуждения можно автоматизировать. Алгоритм Кнута–Морриса–Пратта [1977] (КМР) строит по данной искомой строке x новую структуру данных — *префиксную функцию* — пользуясь которой, «наивный» алгоритм может ускорить свою работу за счёт пропускания заведомо безуспешных сравнений, равно как и заведомо совпадающих символов.

Определение 1. Префиксная функция *для строки* $x = b_1 \dots b_m$ *— это функция* $\pi\colon \{1, \dots, m\} \to \{0, \dots, m-1\}$, *которая, для всякого префикса* $b_1 \dots b_i$ *строки* x *выдаёт длину наибольшего суффикса подстроки* $b_1 \dots b_i$, *который также является префиксом* x.

$$\pi(i) = \max\{\, j \mid j < i,\ b_1 \dots b_j = b_{i-j+1} \dots b_i \,\}$$

В вышеприведённом примере для строки $x = abab$, значение префиксной функции для $i = 3$ равно $\pi(3) = 1$, и руководствуясь этим, алгоритм может перескочить к сравнению вторых символов w_3 и x.

Пример 1. *Функция* π *для строки* $w = abaabaabababa$ *принимает следующие значения:* $\pi(1) = 0$, $\pi(2) = 0$, $\pi(3) = 1$, $\pi(4) = 1$, $\pi(5) = 2$, $\pi(6) = 3$, $\pi(7) = 4$, $\pi(8) = 5$, $\pi(9) = 6$, $\pi(10) = 2$, $\pi(11) = 3$.



Повторно применяя функцию π, можно получить все префиксы x, являющиеся суффиксами $b_1 \dots b_i$, и алгоритм будет их перебирать, пока не найдёт такой, который можно продолжить следующим символом текста.

Алгоритм поддерживает следующий инвариант: в каждой i-й итерации внешнего цикла, перед выполнением строчки 7, самый длинный префикс x, являющийся суффиксом $a_1 \dots a_i$ — это строка $b_1 \dots b_j$.

Лемма 1. *Имея готовую таблицу значений функции* π, *алгоритм КМР работает за время* $\Theta(n)$.

Алгоритм 1 Алгоритм Кнута–Морриса–Пратта

```
1: j = 0
2: for i = 1 to n do
3:         while b_{j+1} ≠ a_i и j > 0 do
4:                 j = π(j)
5:         if b_{j+1} = a_i then
6:                 j = j + 1
7:         if j = m then
8:                 подстрока найдена по смещению i − m
9:                 j = π(j)
```

Доказательство. На первый взгляд, строчка 4 при неудачном стечении обстоятельств может выполняться mn раз. Но это не так. Всякий раз, когда выполняется строчка 4, значение j уменьшается, но ниже нуля оно никогда не становится. Поскольку за всё время работы алгоритма значение j увеличивается не более чем на n (в строке 6), уменьшиться больше чем n раз оно не сможет. □

Осталось научиться быстро строить значения префиксной функции.

Алгоритм 2 Построение значений префиксной функции

```
1: π(1) = 0
2: j = 0
3: for i = 2 to m do
4:         while b_{j+1} ≠ b_i и j > 0 do
5:                 j = π(j)
6:         if b_{j+1} = b_i then
7:                 j = j + 1
8:         π(i) = j
```

2 Поиск с помощью конечных автоматов

Сколько памяти использует алгоритм Кнута–Морриса–Пратта? Если пренебречь зависимостью от длины искомой строки, то алгоритм использует $\log n$ битов, необходимых для хранения переменной i. Однако всё, что делается с этой переменной — это чтение символа a_i и увеличение значения i на единицу. За счёт этого входные символы последовательно читаются один за другим. А в остальном, алгоритм использует $O(1)$ памяти — то есть, *конечный её объём*, не зависящий от длины просматриваемой строки.

Работу алгоритма Кнута–Морриса–Пратта можно формализовать в терминах важной теоретической модели — *конечного автомата*.

2.1 Конечные автоматы

Определение 2 (Клини)**.** ***Детерминированный конечный автомат*** *(deterministic finite automaton, DFA) — пятёрка $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, со следующим значением компонентов.*

- Σ — *алфавит (конечное множество).*

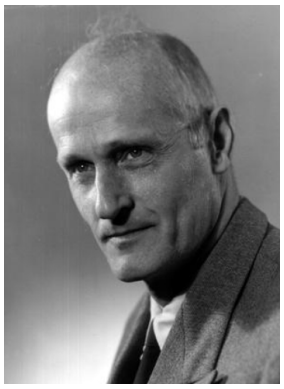


Рис. 2: Стивен Клини (1909–1994).

- Q — *конечное множество состояний.*
- $q_0 \in Q$ — *начальное состояние.*
- *Функция переходов* $\delta\colon Q \times \Sigma \to Q$. *Если автомат находится в состоянии* $q \in Q$ *и читает символ* $a \in \Sigma$, *то его следующее состояние* — $\delta(q, a)$.
- *Множество* ***принимающих состояний*** $F \subseteq Q$.

Для всякой входной строки $w = a_1 \dots a_\ell$, *где* $\ell \geqslant 0$ *и* $a_1, \dots, a_\ell \in \Sigma$, *вычисление — последовательность состояний* $p_0, p_1, \dots, p_{\ell-1}, p_\ell$, *где* $p_0 = q_0$, *и всякое следующее состояние* p_i, *где* $i \in \{1, \dots, \ell\}$, *однозначно определено как* $p_i = \delta(p_{i-1}, a_i)$.

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_{\ell-1}} p_{\ell-1} \xrightarrow{a_\ell} p_\ell$$

Строка ***принимается****, если последнее состояние* p_ℓ *принадлежит множеству* F — *иначе* ***отвергается****.*

Язык, ***распознаваемый*** *автоматом, обозначаемый через* $L(\mathcal{A})$ — *это множество всех строк, которые он принимает.*

Дополнительное обозначение: если DFA начинает вычисление в состоянии $q \in Q$ и читает строку $w = a_1 \dots a_\ell$, то его состояние после её прочтения обозначается через $\delta(q, w)$. Формально, определение функции переходов расширяется до $\delta\colon Q \times \Sigma^* \to Q$, и определяется как $\delta(q, \varepsilon) = q$ и $\delta(q, aw) = \delta(\delta(q, a), w)$. В этих обозначениях, язык, распознаваемый DFA, кратко определяется так.

$$L(\mathcal{A}) = \{\, w \in \Sigma^* \mid \delta(q_0, w) \in F \,\}$$

Реализация на компьютере: таблица значений функции δ хранится в массиве, при чтении каждого символа необходимо один раз обратиться к таблице и изменить одну переменную — состояние DFA.

2.2 Конечный автомат, исполняющий алгоритм Кнута–Морриса–Пратта

Теорема 1. *Для всякой строки* $x = b_1 \dots b_m$ *над алфавитом* Σ *существует DFA с* $m + 1$ *состояниями, распознающий множество* $\Sigma^* x$ *всех строк, заканчивающихся на* x, *и этот DFA можно построить за время* $\Theta(|\Sigma| \cdot m)$.

Доказательство. Сперва строится префиксная функция π для искомой строки x.

Автомат будет читать строку w посимвольно, слева направо — в точности как это делает алгоритм КМР. За один переход автомат должен проделать целую i-ю итерацию алгоритма. Состояния автомата соответствуют значениям переменной j в алгоритме КМР: после чтения строки w автомат должен перейти в такое состояние j, что $a_1 \dots a_j$ — это самый длинный суффикс строки w.

$$Q = \{0, 1, \dots, m\}$$

В начальном состоянии автомат не успел ещё прочитать ни одного символа входной строки.

$$q_0 = 0$$

Переход в каждом состоянии описывает поведение алгоритма КМР. Определение даётся индуктивно по j, сперва для случая $j = 0$.

$$\delta(0, a) = \begin{cases} \delta(\pi(j), a), 0, & b_{j+1} \neq a \\ 1, & b_{j+1} = a \end{cases}$$

Далее, функция определяется для всех i от 1 до $m - 1$.

$$\delta(j, a) = \begin{cases} \delta(\pi(j), a), & \text{если } b_{j+1} \neq a \\ j + 1, & \text{если } b_{j+1} = a \\ \delta(\pi(j), a), & \text{если } j = m \end{cases}$$

В случае $j = m$ автомат повторяет переходы из состояния $\pi(j)$.

$$\delta(m, a) = \delta(\pi(j), a)$$

Наконец, принимающие состояния — это те, в которых только что было закончено чтение подстроки x, и алгоритм КМР готов об этом сообщить.

$$F = \{m\}$$

□

Пример 2 (продолжение примера 1)**.** *Конечный автомат, построенный для строки* $w = abaabaababa$.

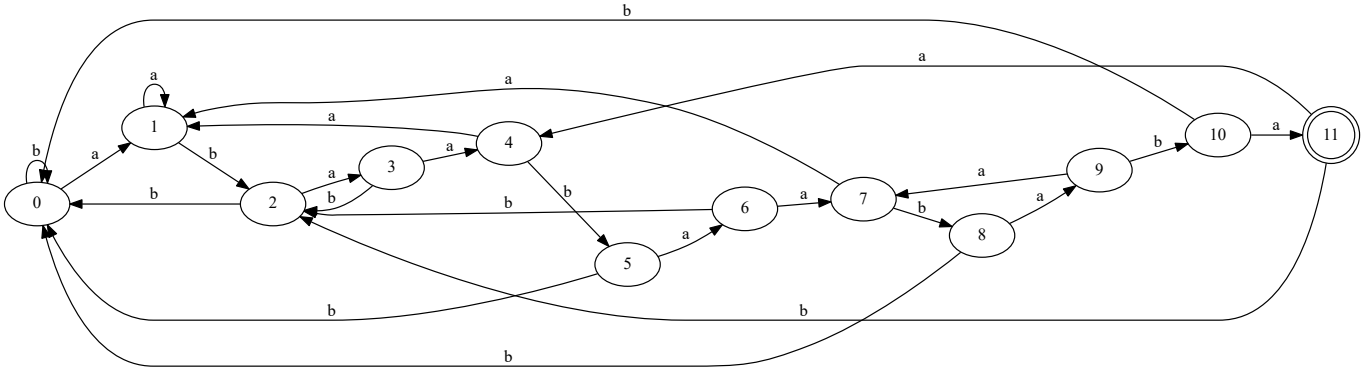


Рис. 3: Альфред Ахо (род. 1941).

2.3 Поиск сразу нескольких строк: алгоритм Ахо–Корасик

Альфред Ахо и Маргарет Корасик [1975].

Пусть для данного текста $w = a_1 \dots a_n$ требуется найти все вхождения не одной строки, а всех строк из конечного множества $K = \{x_1, \dots, x_k\}$.

Алгоритм Ахо–Корасик строит конечный автомат, читающий текст слева направо и определяющий все эти вхождения.

Сперва строится префиксное дерево для данного множества. Надо сказать, что префиксное дерево само по себе — это по сути конечный автомат, распознающий хранимое в нём множество: его вершины — состояния автомата, а помеченные вершины — принимающие состояния. В префиксном дереве только не хватает переходов по символам, не встречающимся в строках данного множества.

В данном случае, однако, надо распознать не само множество K, а множество всех строк, у которых есть суффикс из K. Для этого надо добавить в префиксное дерево недостающие переходы. Пусть v — вершина, которой соответствует строка $v \in \Sigma^*$, и пусть в префиксном дереве нет перехода из этой вершины по символу $a \in \Sigma$. Тогда у строки va нужно определить самый длинный её суффикс, являющийся префиксом некоторой строки из K, и определить переход из вершины v по a в состояние, соответствующее этому суффиксу.

Как это сделать эффективно? Сперва для каждой строки v, лежащей в дереве, нужно найти самый длинный её суффикс, являющийся префиксом некоторой строки из K (то есть, также лежащий в дереве). Это обобщение функции префиксов π из алгоритма КМР. Обобщённую функцию можно обозначить через $\pi\colon \text{prefixes}(K) \to \text{prefixes}(K)$. В дереве её можно обозначить пунктирной линией, ведущей из одной вершины в другую, расположенную выше по уровню.

Для всех односимвольных строк $a \in \Sigma$, находящихся в дереве, самый длинный суффикс — это пустая строка: $\pi(a) = \varepsilon$.

Затем на каждом шаге берётся некоторая строка u, для которой самый длинный суффикс $\pi(u)$ уже определён, и рассматривается всякое её продолжение по символу $a \in \Sigma$ находящееся в дереве. Делается попытка продлить самый длинный суффикс $v = \pi(u)$ символом a: если в дереве префиксов есть такое продолжение va, то значение $\pi(ua)$ и будет на него указывать. Кроме того, если в va распознаётся одна из строк $x_i \in K$, то она же должна распознаваться и в ua.[1] Если же такого продолжения нет, то будут последовательно перебираться суффиксы $\pi(u)$, $\pi(\pi(u))$ и т.д., пока среди них не найдётся такой, у которого в дереве префиксов есть переход по a. Всё это делается за линейное время, если хранить

[1] На рис. 4(среднем) это происходит при добавлении ссылки $\pi(3) = 7$: поскольку строка bb распознаётся в вершине 7, она будет распознаваться и в вершине 3.

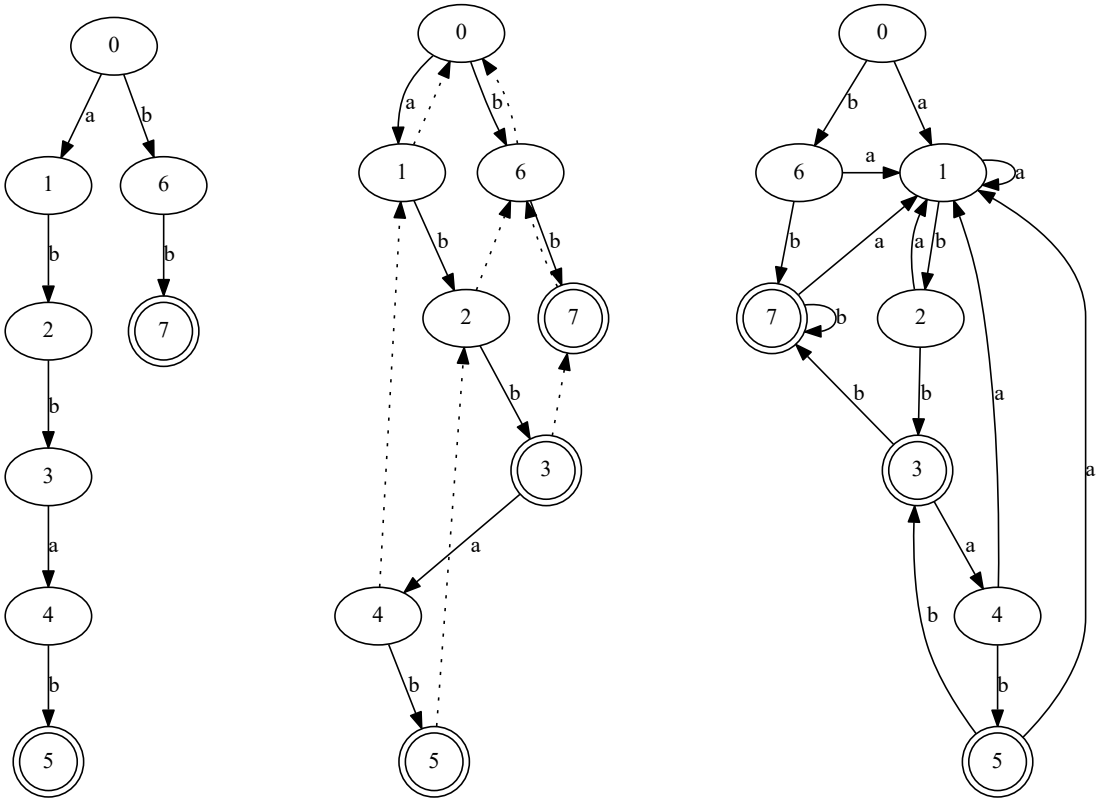


Рис. 4: Структуры данных в алгоритме Ахо–Корасик для множества $K = \{abbab, bb\}$: (слева) префиксное дерево; (посередине) префиксное дерево с суффиксными ссылками; (справа) конечный автомат.

очередь вершин, которые надо рассмотреть.

На последнем этапе строится конечный автомат. Все дуги, входящие в префиксное дерево, становятся переходами конечного автомата. Далее, если в вершине u значение функции переходов $\delta(u, a)$ ещё не определено, то оно определяется процедурой $d(u, a)$, работающей так: если из u есть переход по a, то она возвращает вершину, куда этот переход делается, а если перехода нет, то она запускает $d(\pi(u), a)$ и возвращает то, что та вернёт. Таким образом, опять перебираются суффиксы $\pi(u)$, $\pi(\pi(u))$ и т.д., пока не найдётся переход по a — и $\delta(u, a)$ повторяет этот переход. Как это сделать за линейное время: процедура, вычисляющая $\delta(u, a)$, запоминает результат своей работы в таблице, и когда он потребуется опять, повторные вычисления не производятся.

3 Суффиксные деревья

Вариант задачи поиска: длинная строка («текст») фиксирована, и в ней нужно искать много длинных подстрок — так, чтобы сложность каждого поиска была минимальной. Все ранее описанные алгоритмы плохо для этого приспособлены. Цель: придумать новый алгоритм, который, получив текст на входе, сперва подготовился бы к поиску в этом тексте, а потом мог бы быстро отвечать на запросы с новыми подстроками.

Можно сделать с помощью особой структуры данных: суффиксного дерева. Изобретено

Алгоритм 3 Алгоритм Ахо–Корасик: построение функции π

Дано: множество искомых строк $K = \{x_1, \ldots, x_k\}$.

построить префиксное дерево для K
for all $a \in \Sigma$ **do**
    if в дереве есть вершина a **then**
        добавить её в очередь
    else
        добавить петлю по a в корне
while в очереди есть вершины **do**
    извлечь вершину u из очереди
    for all $a \in \Sigma$ **do**
        if в дереве есть вершина ua **then**
            пусть $v = \pi(u)$
            while в дереве нет вершины va **do**
                $v = \pi(v)$
            определить $\pi(ua) = va$
            пометить, что все строки $x \in K$, распознаваемые в va, распознаются и в ua
            добавить вершину ua в конец очереди

Питером Винером [1973] (род. 1942).

3.1 Упрощённое суффиксное дерево

В упрощённом виде, суффиксное дерево — это префиксное дерево для множества суффиксов данной строки. Пусть $w = a_1 \ldots a_n$ — очень длинная строка. У неё есть не более чем $n+1$ суффиксов — множество $\text{suffixes}(w) = \{\, a_1 \ldots a_i \mid i \in \{0, \ldots, n\} \,\}$ Если для этого множества построить префиксное дерево, то тогда в строке w будет очень удобно искать подстроки: ведь всякая подстрока w — это начало одного из суффиксов w, и его можно непосредственно найти в префиксном дереве, причём время поиска не будет зависеть от длины текста.

Кроме того, суффиксное дерево содержит *суффиксные ссылки* — такие же, как и в алгоритме Ахо–Корасик. Пусть $w = uav$, где $u, v \in \Sigma^*$ и $a \in \Sigma$. Тогда суффиксам av и v соответствуют две вершины в дереве, и в вершине av даётся *суффиксная ссылка* на вершину v, находящуюся на один уровень выше.

Если $w = a_1 \ldots a_n$, то, следуя по суффиксным ссылкам из вершины w, можно последовательно обойти все n вершин, соответствующих суффиксам w.

3.2 Построение упрощённого суффиксного дерева

Для данной строки $w = a_1 \ldots a_n$ алгоритм последовательно строит суффиксные деревья для строк $w_0 = \varepsilon$, $w_1 = a_1$, $w_2 = a_1 a_2$, и так далее для всех префиксов w. Суффиксное дерево для пустой строки состоит из одной вершины и не содержит суффиксных ссылок. Каждое последующее суффиксное дерево получается из предыдущего его достраиванием.

Пусть дано суффиксное дерево для строки w, а также указатель на вершину, соответствующую всей строке w. Требуется получить суффиксное дерево для строки wa, где $a \in \Sigma$. Для этого необходимо продолжить вершины, соответствующие всем суффиксам строки w, символом a. Вершины, соответствующие всем суффиксам строки w, можно просмотреть, следуя по суффиксным ссылкам вплоть до возвращения в корень. Каждую из них следует продолжить символом a, и полученные вершины связать в цепочку новыми суффиксными ссылками.

Реализуется это с помощью рекурсивной процедуры *продолжение*(x, a), получающей в качестве аргумента вершину x, продлевающей её до xa, делающей то же самое со всеми её суффиксами и наконец возвращающей вершину xa. Тогда вызов w = *продолжение*(w,a) вычисляет все необходимые продолжения.

Алгоритм 4 Построение упрощённого суффиксного дерева

Дано: $w = a_1 \dots a_n$.

создать вершину ε
$w = \varepsilon$
for $i = 1, \dots, n$ **do**
    $w =$ продолжение(w, a_i)

процедура *продолжение*(w, a)

if из w ещё нет перехода по a **then**
    создать вершину wa, соединить w с wa
if $w \neq \varepsilon$ **then**
    пусть $v =$ суффиксная_ссылка$[w]$
    пусть $va =$ продолжение(v, a)
    суффиксная_ссылка$[wa] = va$
else
    суффиксная_ссылка$[wa] = \varepsilon$
return wa

Пример построения — на рис. 5.

Общий недостаток: размер префиксного дерева для n строк составит $O(n^2)$, и эта верхняя оценка в худшем случае достигается.

Пример 3. *Префиксное дерево для множества суффиксов строки* $w = a^n b^n$ *содержит* $(n+1)^2$ *вершин. Дерево для* $n = 3$ *изображено на рис. 8(сверху).*

Это слишком много! И как его ни строй, это будет работать слишком медленно.

3.3 Суффиксное дерево и его построение

Суффиксное дерево для w — это компактное префиксное дерево для множества суффиксов w. Дуги, как обычно, помечены парами позиций в строке, и из каждой внутренней вершины исходит не менее двух продолжений. При этом суффиксные ссылки сохраняются только между оставшимися вершинами. Пример — на рис. 8(снизу).

Число вершин в суффиксном дереве — $O(n)$.

Построение: алгоритм Укконена [1995].

Чтобы продлить всех суффиксы на один новый символ, в каждом необходимо увеличить на единицу последнюю позицию на входящей дуге. Чтобы проще выполнять продления, дуги, входящие в листья, помечаются парами вида (i, ∞) — и такие пары считаются продлёнными автоматически. Но остаются ещё некоторые внутренние вершины, которые тоже необходимо продлить. Это в алгоритме Укконена делается совсем иначе, чем в алгоритме построения упрощённого суффиксного дерева: ни одна внутренняя вершина не помечается как содержащая суффикс, а вместо того новые суффиксы, уже лежащие в дереве, продлеваются до тех пор, пока не дорастают до суффиксов, в дереве не лежащих.

Алгоритм дополнительно хранит следующие данные:

- *текущее положение* — тройку вида (вершина, дуга, с какого символа этой дуги начинать), указывающую на самый длинный суффикс, который ещё не был внесён в дерево;

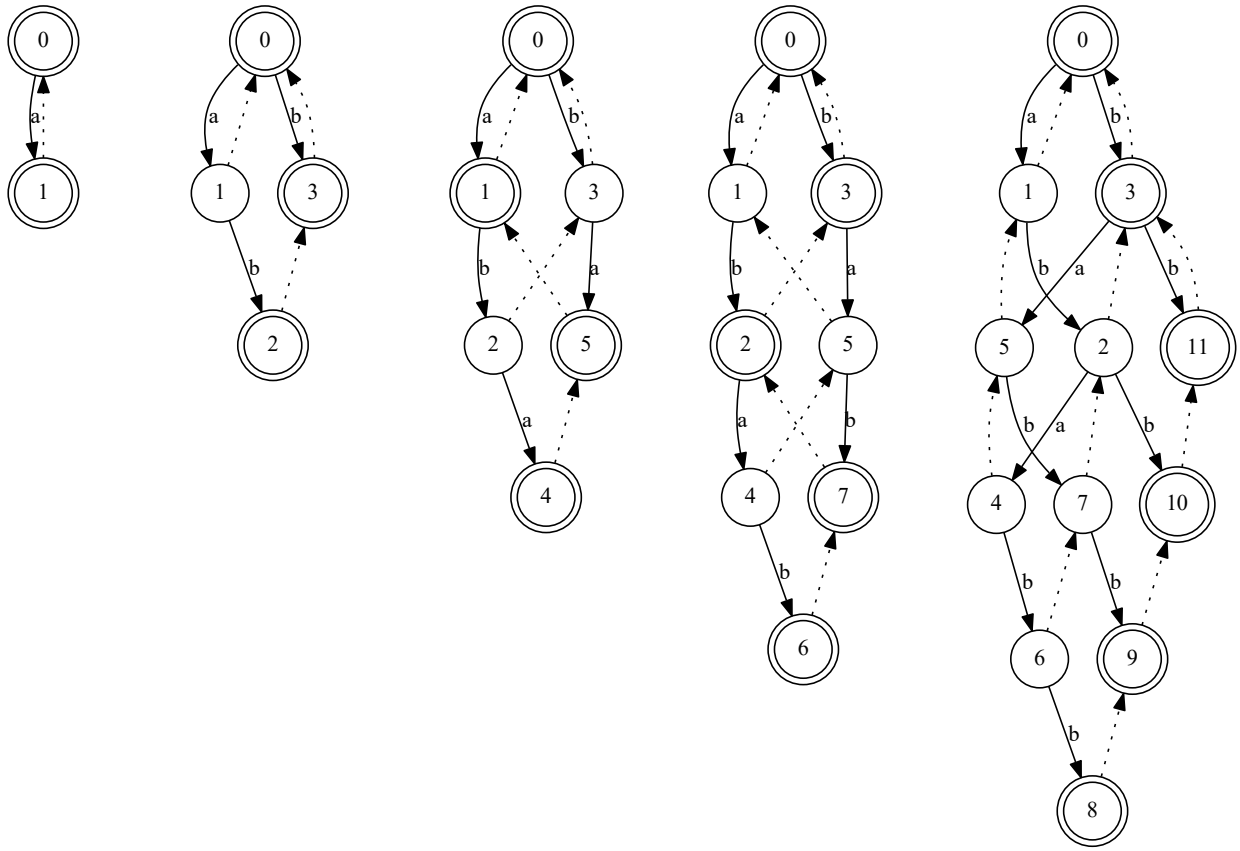


Рис. 5: Упрощённое суффиксное дерево для строки $w = ababb$, его последовательное построение.

- *конечное положение* — лист, соответствующий всей прочитанной части строки;
- *количество недостроенных суффиксов*.

На каждом шаге работы алгоритма очередной входной символ вставляется, начиная из текущего положения. Проще всего тот случай, когда следующий символ на текущей дуге — это как раз и есть входной символ: тогда текущее положение продвигается на один символ вперёд, дерево не изменяется никак, а количество недостроенных суффиксов увеличивается на единицу. Если же на текущей дуге стоит какой-то другой символ, то её необходимо разрезать, создав в текущем положении новую промежуточную вершину, из которой выходят две дуги: остаток разрезанной дуги и новая дуга в ещё одну новую вершину, помеченная прочитанным символом.

Вставить символ в одном этом месте может быть недостаточно, нужно найти следующий суффикс. Если вершина в текущем положении — это корень, то следующий суффикс находится в одной из соседних дуг. Далее, если есть суффиксная ссылка (а суффиксные ссылки в суффиксном дереве такие же, как и в упрощённом суффиксном дереве), то следующий суффикс находится по ней. Наконец, если суффиксной ссылки нет, такое может случиться, только если текущая вершина — непосредственный потомок корня, и тогда следующий суффикс опять-таки находится из корня.

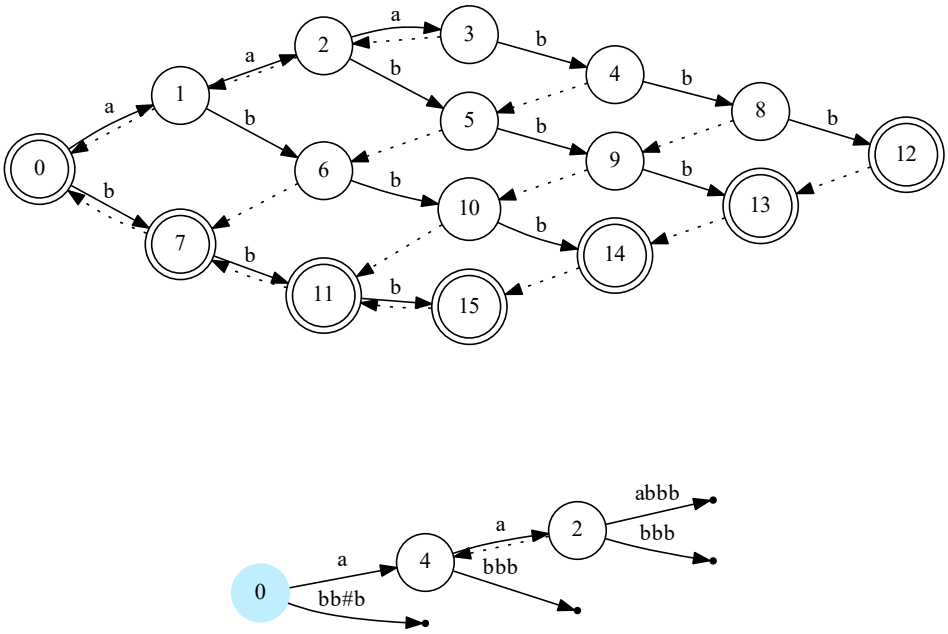


Рис. 6: (сверху) Упрощённое суффиксное дерево для строки $w = aaabbb$; (снизу) суффиксное дерево для этой же строки.

Алгоритм работает за линейное время. Общее число итераций цикла while за всё время работы равно количеству недостроенных суффиксов — а их всего будет n.

«Переход к следующему суффиксу» может работать за линейное время, если из прошлой вершины выходила дуга, помеченная длинной подстрокой, а в новой вершине эта же подстрока представлена длинной последовательностью дуг — тогда через эту последовательность надо будет продвинуться. Однако общее количество таких продвижений ограничено количеством недостроенных суффиксов, а так как за всё время работы алгоритма их ровно n, то общее время работы остаётся $O(n)$.

Из линейного времени работы следует, что и само суффиксное дерево имеет линейный размер.

3.4 Что ещё можно делать с помощью суффиксного дерева?

Кроме поиска, суффиксное дерево позволяет выполнить другие интересные операции. «Вычислить количество вхождений подстроки x в w»: сперва спуститься из корня по строке x, а затем посчитать число вершин в поддереве.

«Найти наибольшую общую подстроку w и x». Делается это следующим обходом суффиксного дерева для w: сперва надо попробовать спуститься по x, и всякий раз, когда дальше спускаться будет некуда (то есть, целой подстроки x в w не содержится), надо подняться по суффиксной ссылке на уровень выше и продолжать дочитывать x. После прочтения всей подстроки x следует вернуть самую длинную из встреченных по дороге подстрок.

Рис. 7: Эско Укконен (род. 1950).

Алгоритм 5 Алгоритм Укконена для построения суффиксного дерева

В начале работы алгоритма текущее положение и конечное положение — это корень дерева, количество недостроенных суффиксов — 0.

for $i = 1$ to n **do**
    увеличить количество недостроенных суффиксов на 1
    while есть недостроенные суффиксы **do**
        if в текущем положении можно прочитать a_i **then**
            продвинуть текущее положение на единицу вперёд
            выйти из цикла
        вставить a_i в текущем положении, при необходимости разрезая дугу
        если на этой итерации уже что-то вставляли, поставить оттуда суффиксную ссылку сюда
        уменьшить количество недостроенных суффиксов на 1
        if текущее положение — в корне **then**
            перейти к следующему суффиксу из корня (он рядом)
        else if есть суффиксная ссылка **then**
            перейти к следующему суффиксу по суффиксной ссылке
        else
            перейти к следующему суффиксу из корня

Список литературы

[1975] A. V. Aho, M. J. Corasick, “Efficient string matching: An aid to bibliographic search”, *Communications of the ACM*, 18:6 (1975), 333–340.

[1977] D. E. Knuth, J. H. Morris Jr., V. R. Pratt, “Fast pattern matching in strings”, *SIAM Journal on Computing*, 6:2 (1977), 323–350.

[1995] E. Ukkonen, “On-line construction of suffix trees”, *Algorithmica*, 14:3 (1995), 249–260.

[1973] P. Weiner, “Linear pattern matching algorithms”, *SWAT 1973*, 1–11.

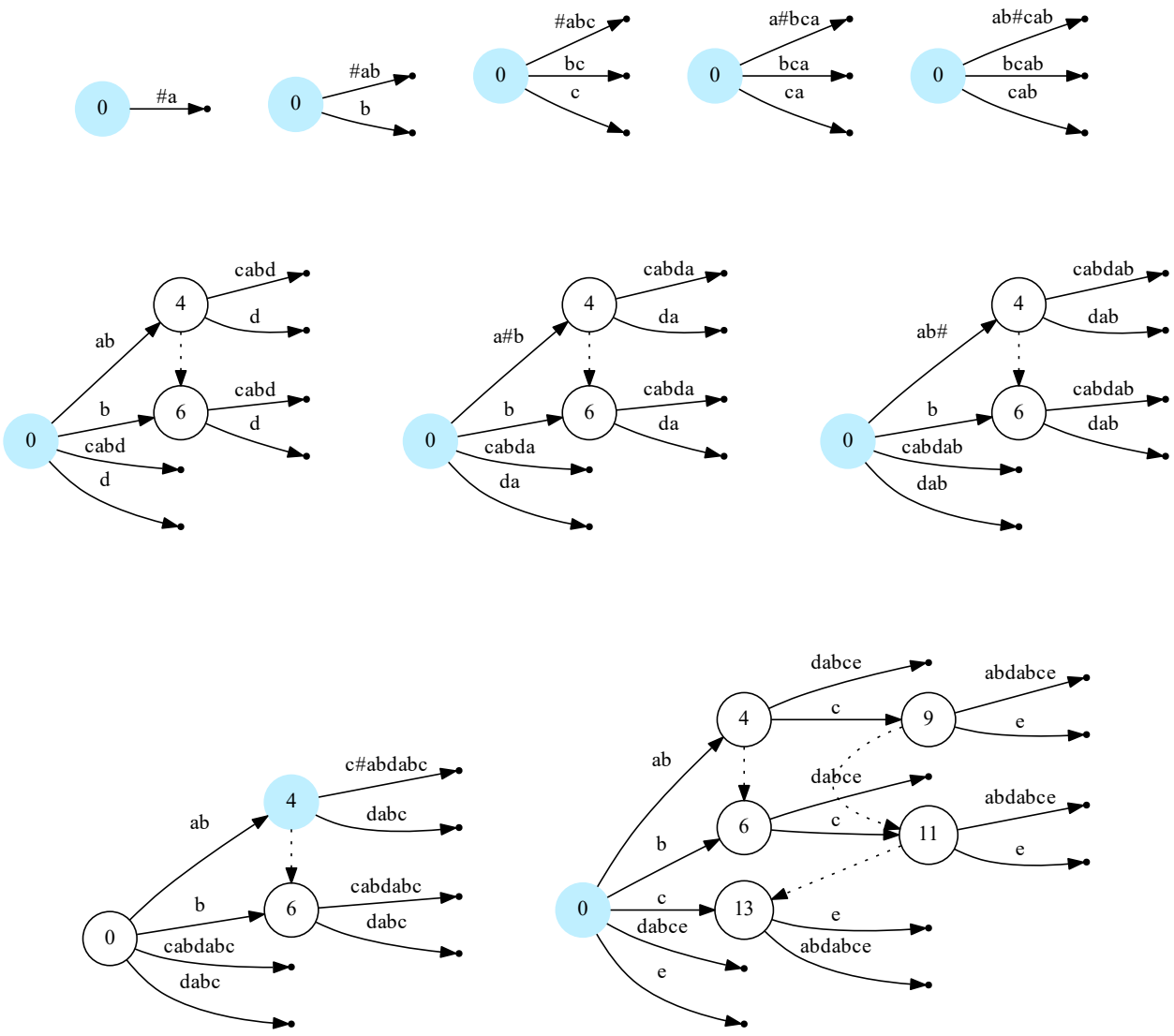


Рис. 8: Построение суффиксного дерева для строки $w = abcabdabce$.