Строковые алгоритмы

Сопоставление с образцом

Простейший пример

Имеется последовательность символов x[1] ... x[n]. Определить, имеются ли в ней идущие друг за другом символы abcd. (Другими словами, требуется выяснить, есть ли в слове x[1] ... x[n] подслово abcd.)

Решение. Имеется примерно n (если быть точным, n-3) позиций, на которых может находиться искомое подслово в исходном слове. Для каждой из позиций можно проверить, действительно ли там оно находится, сравнив четыре символа. Однако есть более эффективный способ. Читая слово x[1] ... x[n] слева направо, мы ожидаем появления буквы a. Как только она появилась, мы ищем за ней букву b, затем c, и, наконец, d. Если наши ожидания оправдываются, то слово abcd обнаружено. Если же какая-то из нужных букв не появляется, мы оказываемся у разбитого корыта и начинаем всё сначала. □

Конечные автоматы

Можно сказать, что при чтении слова X слева направо мы в каждый момент находимся в одном из следующих состояний:

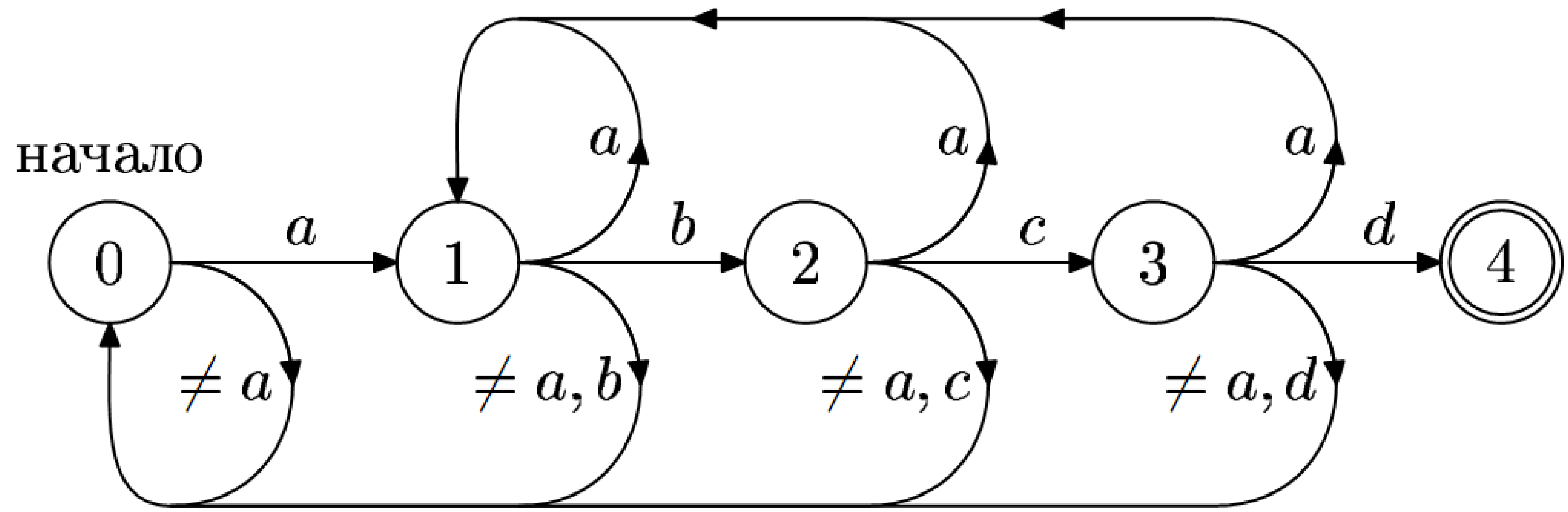
- Начальное (0)
- Сразу после «a» (1)
- Сразу после «ab» (2)
- Сразу после «abc» (3)
- Сразу после «abcd» (4)

Сразу после чтения символа мы переходим в следующее состояние по правилу, указанному в таблице (см. далее). Как только попадем в состояние 4, работа заканчивается.

Конечные автоматы

Текущее состояние	Очередная буква	Новое состояние
0	a	1
0	кроме a	0
1	b	2
1	a	1
1	кроме a,b	0
2	c	3
2	a	1
2	кроме a,c	0
3	d	4
3	a	1
3	кроме a,d	0

Конечный автомат в виде графа



Повторения в образце

Можно ли в предыдущих рассуждениях заменить слово abcd на произвольное слово?

Повторения в образце

Решение.

Нет, проблемы связаны с тем, что в образце могут быть повторяющиеся буквы.

Пусть, например, мы ищем вхождения слова ababc.

```
x  y  z  a  b  a  b | a  b  c  …
         a  b  a  b | c
               a  b | a  b  c
```

Конечный автомат для этого случая

Состояния будут соответствовать наибольшему началу образца, являющемуся концом прочитанной части слова. Их будет шесть:

0

1 (a)

2 (ab)

3 (aba)

4 (abab)

5 (ababc)

Текущее состояние	Очередная буква	Новое состояние
0	a	1 (a)
0	кроме a	0
1 (a)	b	2 (ab)
1 (a)	a	1 (a)
1 (a)	кроме a,b	0
2 (ab)	a	3 (aba)
2 (ab)	кроме a	0
3 (aba)	b	4 (abab)
3 (aba)	a	1 (a)
3 (aba)	кроме a,b	0
4 (abab)	c	5 (ababc)
4 (abab)	a	3 (aba)
4 (abab)	кроме a,c	0

Алгоритм Кнута-Морриса-Пратта

Имеется образец S и строка T, и нужно определить индекс, начиная с которого строка S содержится в строке T. Если S не содержится в T — вернуть индекс, который не может быть интерпретирован как позиция в строке (например, отрицательное число).

При необходимости отслеживать каждое вхождение образца в текст имеет смысл завести дополнительную функцию, вызываемую при каждом обнаружении образца.

Алгоритм КМП

Рассмотрим сравнение строк на позиции *i*, где образец S[0,m-1] сопоставляется с частью текста T[I,i+m-1]. Предположим, что первое несовпадение произошло между T[i+j] и S[j], где $1< j < m$.

Тогда T[i, i + j - 1] = S[0, j - 1] = P и a = T[i + j] != S[j] = b.

- При сдвиге вполне можно ожидать, что префикс (начальные символы) образца S сойдется с каким-нибудь суффиксом (конечные символы) текста P. Длина наиболее длинного префикса, являющегося одновременно суффиксом, есть префикс-функция от строки S для индекса j.
- Это приводит нас к следующему алгоритму: пусть π(j) — префикс-функция от строки S[0,m-1] для индекса j. Тогда после сдвига мы можем возобновить сравнения с места T[i+j] и S[π(j)] без потери возможного местонахождения образца.
- Вопрос – какова вычислительная сложность этого алгоритма?

Префикс-функция

Часто префикс-функцию записывают в виде вектора длиной |S|-1.

Например, для строки 'abcdabscabcdabia' префикс-функция будет такой: π(abcdabscabcdabia)='0000120012345601'. Иногда для полноты считают, что π(S,0)=0.

Алгоритм вычисления префикс-функции

Пусть π(S,i)=k. Попробуем вычислить префикс-функцию для i+1.

Если S[i+1]=S[k+1], то, естественно, π(S,i+1)=k+1. Если нет — пробуем меньшие суффиксы. Очевидно, что S[0 … π(S,k)] также будет суффиксом строки S[0 … i], а для любого j ϵ (k,i) строка S[0 … j] суффиксом не будет. Таким образом, получается алгоритм:

1. При S[i+1]=S[k+1] — положить π (S,i+1)=k+1.
2. Иначе при k=0 — положить π(S,i+1)=0.
3. Иначе — установить k:= π(S,k), GOTO 1.

Для строки <code>'abcdabscabcdabia'</code> вычисление будет таким:

'a'!='b' => π=0;
'a'!='c' => π=0;
'a'!='d' => π=0;
'a'=='a' => π=π+1=1;
'b'=='b' => π=π+1=2;
'c'!='s' => π=0;
'a'!='c' => π=0;
'a'=='a' => π=π+1=1;
'b'=='b' => π=π+1=2;
'c'=='c' => π=π+1=3;
'd'=='d' => π=π+1=4;

Z-функция

Z-функция от строки S - массив Z, каждый элемент которого Z[i] равен длиннейшему префиксу подстроки, начинающейся с позиции i в строке S, который одновременно является и префиксом всей строки S. Значение Z-функции в нулевой позиции считается равным длине всей строки.

Часто Z-функцию записывают в виде вектора длиной |S|. Например, для строки 'abcdabscabcdabia' Z-функция будет такой: Z(abcdabscabcdabia)=[16,0,0,0,2,0,0,0,6,0,0,0,2,0,0,1].

Алгоритм вычисления Z-функции

Будем хранить индексы L и R, обозначающие начало и конец префикса с наибольшим найденным на данный момент значением R. Изначально $L = R = 0$.

Пусть нам известны значения Z-функции для позиций 1..i - 1. Попробуем вычислить значение Z-функции для позиции i.

- Если i ϵ [L,R] рассмотрим значение Z-функции для позиции $j = i - L$.
  Если i + Z[j] <= R, то Z[i] = Z[j], так как мы находимся в подстроке, совпадающей с префиксом всей строки.
- Если же $i + Z[j] > R$, то необходимо досчитать значение Z[i] простым циклом, перебирающим символы после R, пока не найдется символ, не совпадающий с соответствующим символом из префикса. После этого изменяем, значение L на i и значение R на номер последнего символа, совпавшего с соответствующим символом из префикса.
- Если I не входит в [L…R], то считаем значение Z[i] простым циклом, сравнивающим символы подстроки начинающейся с i-того символа и соответствующие символы из префикса. Когда будет найдено несоответствие или будет достигнут конец строки, изменяем значение L на i и значение R на номер последнего символа, совпавшего с соответствующим символом из префикса.

Примеры использования Z-функции

- 1) Z-функцию можно использовать для поиска образца T в строке S, с помощью алгоритма Кнута — Морриса — Пратта, использующего Z-функцию, вместо префикс-функции.
- 2) Зная Z-функцию строки, можно однозначно восстановить префикс-функцию этой строки, и наоборот.

Алгоритм Бойера — Мура

Считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке. Был разработан Робертом Бойером и Джеем Муром в 1977 году.

Преимущество этого алгоритма в том, что ценой некоторого количества предварительных вычислений над шаблоном (но не над строкой, в которой ведётся поиск) шаблон сравнивается с исходным текстом не во всех позициях — часть проверок пропускаются как заведомо не дающие результата.

Описание алгоритма БМ

Алгоритм основан на трёх идеях:

1. **Сканирование слева направо, сравнение справа налево.**
2. **Эвристика стоп-символа.**
3. **Эвристика совпавшего суффикса.**

Сканирование слева направо, сравнение справа налево.

- Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен.
- Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на **несколько** символов вправо, и проверка снова начинается с последнего символа.
- Эти «несколько», упомянутые в предыдущем абзаце, вычисляются по двум следующим эвристикам.

Эвристика стоп-символа

Предположим, что мы производим поиск слова «колокол». Первая же буква не совпала — «к» (назовём эту букву *стоп-символом*). Тогда можно сдвинуть шаблон вправо до *последней* буквы «к».

```
Строка:          * * * * * * к * * * * * *
Шаблон:          к о л о к о л
Следующий шаг:       к о л о к о л
```

Эвристика стоп-символа

Если стоп-символа в шаблоне вообще нет, шаблон смещается за этот стоп-символ.

```
Строка:        * * * * * а л * * * * * * * *
Шаблон:        к о л о к о л
Следующий шаг:               к о л о к о л
```

Эвристика стоп-символа

Если стоп-символ «к» оказался за другой буквой «к», эвристика стоп-символа не работает.

```
Строка:          * * * * к к о л * * * * *
Шаблон:            к о л о к о л
Следующий шаг: к о л о к о л      ?????
```

В таких ситуациях выручает третья идея алгоритма БМ — эвристика совпавшего суффикса.

Эвристика совпавшего суффикса

Если при сравнении строки и шаблона совпало один или больше символов, шаблон сдвигается в зависимости от того, какой суффикс совпал.

```
Строка:             * * * * т о к о л * * * * *
Шаблон:                     к о л о к о л
Следующий шаг:                      к о л о к о л
```

В данном случае совпал суффикс «окол», и шаблон сдвигается вправо до ближайшего «окол». Если подстроки «окол» в шаблоне больше нет, но он начинается на «кол», сдвигается до «кол», и т. д.

Предварительные вычисления

- Обе эвристики требуют предварительных вычислений — в зависимости от шаблона поиска заполняются две таблицы.
- Таблица стоп-символов по размеру соответствует алфавиту (например, если алфавит состоит из 256 символов, то её длина 256); таблица суффиксов — искомому шаблону.
- Именно из-за этого алгоритм Бойера-Мура не учитывает совпавший суффикс и несовпавший символ *одновременно* — это потребовало бы слишком много предварительных вычислений.

Таблица стоп-символов

Считается, что символы строк нумеруются с 1 (как в Паскале).

В таблице стоп-символов указывается последняя позиция в *шаблоне поиска* (**исключая последнюю букву**) каждого из символов алфавита.

Для всех символов, не вошедших в шаблон, пишем 0 (для нумерации с 0 — соответственно, –1).

Таблица стоп-символов

Например, если шаблон=«abcdadcd», таблица стоп-символов будет выглядеть так:

Символ	**a**	**b**	**c**	**d**	**[все остальные]**
Последняя позиция	**5**	**2**	**7**	**6**	**0**

Таблица стоп-символов

Например, если шаблон=«abcdadcd», таблица стоп-символов будет выглядеть так:

Символ	a	b	c	d	[все остальные]
Последняя позиция	5	2	7	6	0

Для стоп-символа «d» последняя позиция будет 6, а не 8 — последняя буква не учитывается. Это известная ошибка, приводящая к неоптимальности. Для АБМ она не фатальна («вытягивает» эвристика суффикса), но фатальна для упрощённой версии АБМ —алгоритма Хорспула.

Если несовпадение произошло на позиции i, а стоп-символ c, то сдвиг будет i-StopTable[c].

Таблица суффиксов

Для каждого возможного суффикса *S* шаблона указываем наименьшую величину, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал с *S*.

Если такой сдвиг невозможен, ставится |шаблон| (в обеих системах нумерации). Например, для того же шаблона=«abcdadcd» будет:

Суффикс	[пустой]	d	cd	dcd	...	abcdadcd
Сдвиг	1	2	4	8	...	8
Иллюстрация						
было	?	?d	?cd	?dcd	...	abcdadcd
стало	abcdadcd	abcdadcd	abcdadcd	abcdadcd	...	abcdadcd

Таблица суффиксов

Если шаблон начинается и заканчивается одной и той же комбинацией букв, |шаблон| вообще не появится в таблице. Например, для шаблона=«колокол» для всех суффиксов (кроме, естественно, пустого) сдвиг будет равен 4.

Суффикс	[пустой]	л	ол	...	олокол	колокол
Сдвиг	1	4	4	...	4	4
Иллюстрация						
было	?	?л	?ол	...	?олокол	колокол
стало	колокол	колокол	колокол	...	колокол	колокол

Алгоритм вычисления таблицы суффиксов

Существует быстрый алгоритм вычисления таблицы суффиксов.

Этот алгоритм использует префикс-функцию строки.

```
m = length(шаблон)
pi[] = префикс-функция(шаблон)
pi1[] = префикс-функция(обращение(шаблон))
for j=0..m
  suffshift[j] = m - pi[m]
for i=1..m
  j = m - pi1[i]
  suffshift[j] = min(suffshift[j], i - pi1[i])
```

suffshift[0] соответствует всей совпавшей строке; suffshift[m] — пустому суффиксу

Алгоритм Рабина — Карпа

Алгоритм поиска строки, который ищет шаблон, то есть подстроку, в тексте используя хеширование.

Разработан в 1987 году.

Алгоритм редко используется для поиска одиночного шаблона, но имеет значительную теоретическую важность и очень эффективен в поиске совпадений множественных шаблонов.

Для текста длины n и шаблона длины m, его среднее время исполнения и лучшее время исполнения это $O(n)$, но в (весьма нежелательном) худшем случае он имеет производительность $O(nm)$, что является одной из причин того, почему он не слишком широко используется. Однако, алгоритм имеет уникальную особенность находить **любую** из k строк менее, чем за время $O(n)$ в среднем, независимо от размера k.

Алгоритм Рабина-Карпа

- Одно из простейших практических применений алгоритма Рабина — Карпа состоит в определении плагиата. Скажем, например, что студент пишет работу по ***Моби Дику***.
- Коварный профессор находит различные исходные материалы по *Моби Дику* и автоматически извлекает список предложений в этих материалах.
- Затем, алгоритм Рабина — Карпа может быстро найти в проверяемой статье примеры вхождения некоторых предложений из исходных материалов.
- Для устранения чувствительности алгоритма к небольшим различиям, можно игнорировать детали, такие как регистр или пунктуация при помощи их удаления.
- Поскольку количество строк, которые мы ищем, *k*, очень большое, обычные алгоритмы поиска одиночных строк становятся неэффективными.

Алгоритм Рабина-Карпа

Использование хеширования для поиска подстрок сдвигом

```
function RabinKarp(string s[1..n], string sub[1..m])
    hsub := hash(sub[1..m])
    hs := hash(s[1..m])
    for i from 1 to (n-m+1)
        if hs = hsub
            if s[i..i+m-1] = sub
                return i
        hs := hash(s[i+1..i+m])
    return not found
```

Алгоритм Р-К и производительность

- Ключом к производительности алгоритма Рабина-Карпа является эффективное вычисление хэш-значения последовательных подстрок текста.
- Одна популярная и эффективная кольцевая хэш-функция интерпретирует каждую подстроку как число в некоторой системе счисления, основание которой является большим простым числом.
- Например, если подстрока "hi" и основание системы счисления 101, хэш-значение будет $104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCII код 'h' — 104 и 'i' — 105)

Алгоритм Р-К и производительность

- Технически, этот алгоритм только подобен настоящему числу в недесятичной системе представления, так как для примера мы взяли "основание" меньше, чем одну из его "цифр".
- Существенная польза достигается таким представлением, которое возможно для расчёта хэш-значения следующей подстроки из значения предыдущей путём исполнения только постоянного набора операций, независимо от длин подстрок.

Алгоритм Р-К и производительность

- Например, если мы имеем текст "abracadabra" и ищем образец длины 3, мы можем рассчитать хэш подстроки "bra" из хэша подстроки "abr" (предыдущая подстрока), вычитая число добавленное для первой буквы 'a' из "abr", т.е. 97×101^2 (97 — ASCII для 'a' и 101 — основание, которое мы используем), умножение на основание и наконец добавляя последнее число для "bra", т.е. $97 \times 101^0 = 97$.
- Если подстроки в запросе длинны, этот алгоритм достигает большой экономии сравнимо с многими другими схемами хэширования.

Алгоритм Р-К и производительность

- Алгоритм **Рабина-Карпа** в поиске одиночного образца хуже алгоритма Кнута — Морриса — Пратта, алгоритма Бойера — Мура и других быстрых алгоритмов поиска строк по причине его медленного поведения в худшем случае.
- Алгоритм Рабина-Карпа можно также использовать для поиска множественных образцов с трудоемкостью, линейной в лучшем случае и квадратичной в труднодостижимом худшем случае.

Алгоритм Р-К и производительность

- Алгоритм **Рабина-Карпа** в поиске одиночного образца хуже алгоритма Кнута — Морриса — Пратта, алгоритма Бойера — Мура и других быстрых алгоритмов поиска строк по причине его медленного поведения в худшем случае.
- Алгоритм Рабина-Карпа можно также использовать для поиска множественных образцов с трудоемкостью, линейной в лучшем случае и квадратичной в труднодостижимом худшем случае.
- Но и здесь он проигрывает алгоритму **Ахо - Корасик**, имеющему линейное время работы в худшем случае.

Алгоритм Ахо-Корасик

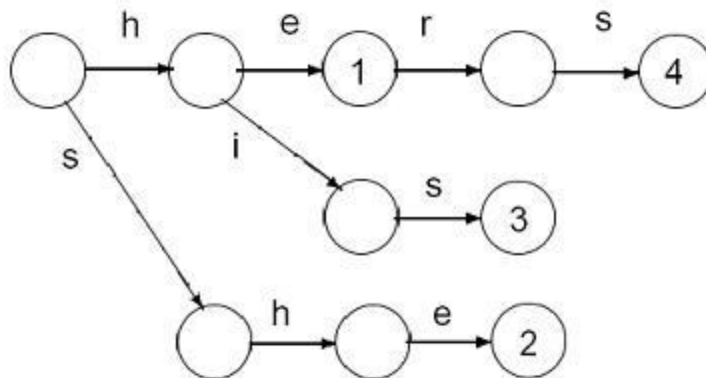
- **Алгоритм Ахо-Корасик (АК)** ***(Aho-Corasick algorithm) (AC)*** - классическое решение задачи точного сопоставления множеств. Он работает за время O (n + m + z) , где z - количество появлений шаблонов в T.
  АК основан на структуре данных **"дерево ключевых слов"** ***(keyword tree)***.

- **Дерево ключевых слов (или "бор")** ***(keyword tree, trie)*** для множества шаблонов P - это дерево с корнем K, такое что:
  1. Каждое ребро e в K отмечено одним символом.
  2. Всякие два ребра, исходящие из одной вершины, имеют разные метки.
  Определим **метку вершины** v как конкатенацию меток ребер, составляющих путь из корня в v, и обозначим ее L (v).
  3. Для каждого шаблона P_i из множества P есть вершина v, такая что L (v) = P_i.
  4. Метка каждой вершины-листа является шаблоном из множества P.

Алгоритм Ахо-Корасик

Пример дерева ключевых слов (бора)

Дерево ключевых слов для P = {he, she, his, hers}:



Алгоритм Ахо-Корасик

Построение бора

Начинаем с дерева из одной вершины (корня); добавляем шаблоны P_i один за другим:

- Следуем из корня по ребрам, отмеченным буквами из P_i, пока возможно.
- Если P_i заканчивается в v, сохраняем идентификатор P_i (например, i) в v.
- Если ребра, отмеченного очередной буквой P_i нет, то создаем новые ребра и вершины для всех оставшихся символов P_i.

Это занимает, очевидно, $O(|P_1| + ... + |P_k|) = O(n)$ времени

Алгоритм Ахо-Корасик

Поиск строки в бору

- Начинаем в корне, идем по ребрам, отмеченным символами S, пока возможно.
- Если с последним символом S мы приходим в вершину с сохраненным идентификатором, то S - слово из словаря.
- Если в какой-то момент ребра, отмеченного нужным символом, не находится, то строки S в словаре нет.

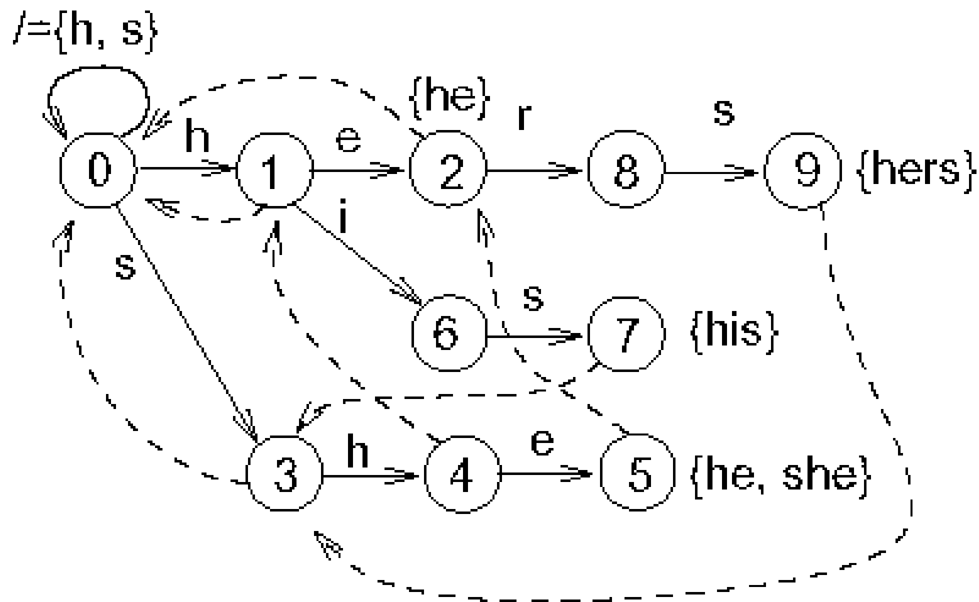
Ясно, что это занимает O (|S|) времени. Таким образом, бор - это эффективный способ хранить словарь и искать в нем слова

Автомат Ахо-Корасик

Состояния: узлы бора.
Начальное состояние: корень, обозначим его 0.
Действия автомата определяются тремя функциями, определенными для всех состояний:
1. **Функция goto** *g (s, a)* указывает, в какое состояние переходить из данного состояния s при просмотре символа a.
Если ребро *(u, v)* отмечено символом *a*, то *g (u, a) = v;*
g (0, a) = 0 для всех символов *a*, которыми не отмечено ни одно ребро, выходящее из корня.
~>Автомат остается в корне, пока просматриваются побочные символы.
При всех остальных аргументах *g* пусть выдает -1.
2. **Функция неудачи** *f (s)* указывает, в какое состояние переходить при просмотре неподходящего символа.
Рассмотрим метку вершины *s* и найдем самый длинный суффикс этой метки, такой, что с него начинается некоторый шаблон из множества P. Тогда *f (s)* пусть указывает на вершину, метка которой - этот суффикс.
3. **Выходная функция** *out (s)* выдает множество шаблонов, которые обнаруживаются при переходе в состояние *s*

Автомат Ахо-Корасик

Пример автомата АК



Пунктиром обозначены переходы при неудаче (значения функции *f*); те, которые не показаны, ведут в корень

Автомат Ахо-Корасик

Поиск шаблонов с помощью автомата АК

```
q := 0; // начальное состояние - корень.
  for i := 1 to m do
    while g (q, T [i]) = -1 do
      q := f (q);
    q := g (q, T [i]);
    if out (q) ≠ 0 then print i, out (q);
  endfor;
```

Построение автомата АК

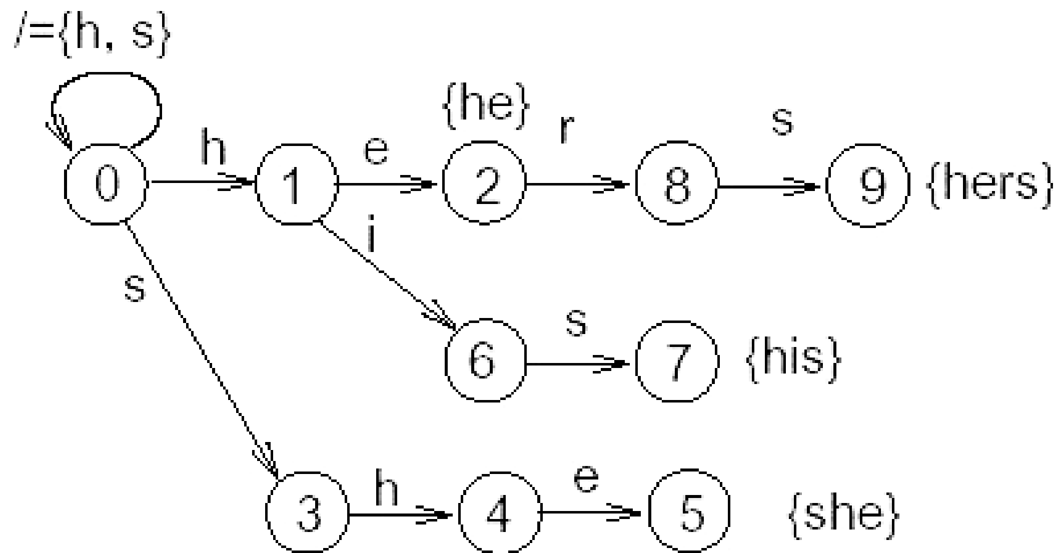
Этап I:

1. Строим бор для словаря P.
   При добавлении каждого слова P_i из P, вершине v с меткой P_i, сопоставим out (v) := $\{P_i\}$
2. Закончим построение функции *goto*, добавив несуществующие переходы из корня: g (0, a) = 0 для всех символов a, не отмечающих ни одного ребра, выходящего из корня. Это также можно сделать неявно.

Если алфавит фиксирован, этап I занимает O (n) времени

Построение автомата АК

Результат этапа I:



Построение автомата АК

Этап II:

```
Q := пустая очередь
для каждого символа a, отмечающего хоть одно ребро из корня [g(0,a)≠0]
    f (g (0, a)) := 0
    добавить в очередь Q вершину g (0, a)
пока очередь Q не пуста
    взять вершину r из очереди Q
    для каждого символа a, отмечающего хоть одно ребро из
            вершины r [g (r, a) ≠ -1]
      u := g (r, a)
      добавить в очередь Q вершину u
      v := f (r)
      пока g (v, a) = -1
        v := f (v)
       f (u) := g (v, a)
      out (u) := out (u)∪ out (f (u))
```

Построение автомата АК

- Функция неудачи и выходная функция вычисляются для всех вершин в порядке обхода в ширину.
  ~>Когда мы работаем с вершиной, все вершины, находящиеся ближе, чем она, к корню (в т. ч. все те, метки которых короче, чем метка данной), уже обработаны.
- Рассмотрим узлы r и $u = g\ (r, a)$, т. е. r - родитель u, и L (u) = L (r) a.
  Теперь нужно, чтобы $f\ (u)$ указывало на ту вершину, метка которой является самым длинным суффиксом L (u), являющимся также началом некоторого шаблона из множества P.
- Эта вершина ищется путем просматривания вершин, метки которых являются все более и более короткими суффиксами L (r), пока не находится вершина v, для которой $g\ (v, a)$ определено; тогда $g\ (v, a)$ и присваивается $f\ (u)$.
- Кстати, и v, и $g\ (v, a)$ могут быть корнем.
  Теперь разберемся с out (u) := out (u) ∪ out ($f\ (u)$).
  Это делается потому, что все шаблоны, распознаваемые при переходе в состояние $f\ (u)$ (и только они) являются надлежащими суффиксами L (u) и должны быть отслежены при переходе в состояние u

Построение автомата АК

Этап II тоже может быть выполнен за время O (n):

- Обход в ширину сам по себе занимает время пропорциональное размеру дерева, т. е. O (n).

Сколько же времени требуется на переходы по функции *f* в самом внутреннем цикле?

- Рассмотрим вершины $u_1, ..., u_l$, проходимые при введении в бор шаблона $a_1, ..., a_l$, и глубины вершин, на которые указывают их функции неудачи, обозначенные $df(u_1), ..., df(u_l)$.
- При этом $df(u_{i+1}) \leq df(u_i) + 1$. Это значит, что значения *df* могут увеличиваться не более *l* раз за весь путь. С другой стороны каждое выполнение $v := f(v)$ уменьшает значение $df(u)$ как минимум на 1.
  ~>Итого при просчете функций *f* для вершин шаблона длины *l* совершается не более *l* переходов.
  ~>Для всех вершин будет совершено не более *n* переходов.
- А много ли времени нужно на выполнение out (u) := out (u) ∪ out ($f(u)$)?
  Нет: множества обнаруживаемых шаблонов можно хранить в виде связных списков, так что операция объединения выполняется за константное время.
  (Все шаблоны в out ($f(u)$) короче, чем L (u), которая (возможно) является единственным членом out (u) перед объединением)

Спасибо!

Вопросы?