Алгоритмы на строках

сборы минской области по информатике, февраль 2021

План

- Алгоритмы на строках
  сборы минской области по информатике, февраль 2021
- Z - функция
- Хэш-функции
- Палиндромы
- Проверка скобочных выражений
- Вычисление арифметических выражений
- Алгоритм Ахо-Корасик
- Использованные источники

  - [1] MAXimal - Электронный ресурс: Z-функция строки и её вычисление. Режим доступа: https://e-maxx.ru/algo/z_function.
  - [2] MAXimal - Электронный ресурс: Алгоритмы хэширования в задачах на строки. Режим доступа: https://e-maxx.ru/algo/string_hashes.
  - [3] MAXimal - Электронный ресурс: Нахождение всех подпалиндромов. Режим доступа: https://e-maxx.ru/algo/palindromes_count.
  - [4] Habr - Электронный ресурс: Алгоритм Ахо-Корасик. Режим доступа: https://habr.com/ru/post/198682/.
  - [5] MAXimal - Электронный ресурс: Алгоритм Ахо-Корасик. Режим доступа: https://e-maxx.ru/algo/aho_corasick.

Z - функция

Z - функция

- Пусть дана строка S длины N.
- Тогда Z-функция от этой строки — это массив Z длины N, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки S. Иными словами, $Z[i]$ — это наибольший общий префикс строки и её -го суффикса.
- Примеры

i	0	1	2	3
S	a	b	a	b
Z	4	0	2	0

i	0	1	2	3
S	a	a	a	a
Z	4	3	2	1

i	0	1	2	3	4	5	6
S	a	b	a	c	a	b	a
Z	7	0	1	0	3	0	1

Тривиальный алгоритм

- Определение можно представить в виде алгоритма за $O(n^2)$
- Для каждой позиции i строки S подбираем ответ $Z[i]$. Идем начиная с нуля, и до тех пор, пока мы не обнаружим несовпадение или не дойдём до конца строки.

```
vector<int> Z(string s) {
   int n = (int)s.length();
   vector<int> z(n);
   for (int i = 1; i < n; i++)
      while (i + z[i] < n && s[z[i]] == s[i + z[i]])
         z[i]++;
   return z;
}
```

Эффективный алгоритм

- Чтобы получить эффективный алгоритм, будем вычислять значения по очереди — от 1 до $n-1$, и при этом при вычислении очередного значения будем использовать уже вычисленные значения.
- **Отрезком совпадения** назовем подстроку совпадающую с префиксом строки S.
- Тогда значение Z-функции $Z[i]$ — это длиннейший отрезок совпадения, начинающийся в позиции i (и заканчивающийся в позиции $i+Z[i]-1$).
- В алгоритме будем поддерживать координаты $[l;r]$ самого правого отрезка совпадения, т.е. отрезок, который оканчивается правее всего. Тогда индекс r — это такая граница, до которой наша строка уже была просканирована алгоритмом, а всё остальное — пока ещё не известно.

Эффективный алгоритм

- На каждой итерации мы имеем два варианта:
- $i > r$ — т.е. еще не успели обработать.
- Тогда будем искать $Z[i]$ тривиальным алгоритмом. И если $Z[i] > 0$ то мы будем обязаны обновить координаты самого правого отрезка — т.к. нашли новый отрезок совпадения.

Эффективный алгоритм

- $i \leq r$ — т.е. мы можем инициализировать $Z[i]$ ненулевым начальным значение .
- Заметим что подстроки $S[l \dots r]$ и $S[0 \dots r-l]$ совпадают:

0		$r-l$		l		r
a	b	a	c	a	b	a
7	0	1	0	3	0	1

- Это означает что для **начального** значения $Z[i]$ мы можем взять соответствующее ему значение из отрезка $S[0 \dots r-l]$, а именно $Z[i-l]$.
- Однако значение Z$[i-l]$ может оказаться слишком большим: таким, что при применении его к позиции i оно выходит за границы r. Этого допустить нельзя, т.к. символы правее r мы еще не обработали.
- Таким образом, в качестве **начального** приближения для Z$[i]$ безопасно брать значение: $Z[i] = \min(Z[i-l],\ r-i+1)$.

Эффективный алгоритм

- Проинициализировав $Z[i]$ начальным значением, дальше действуем тривиальным алгоритмом — т.к. после границы r может быть продолжение отрезка совпадения.
- Так же мы должны обновить координаты самого правого отрезка — т.к. нашли новый отрезок совпадения.

Реализация

```
vector<int> Z(string s) {
    int n = (int)s.length();
    vector<int> z(n);
    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
        if (i <= r)
            z[i] = min(z[i - l], r - i + 1);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            z[i]++;
        if (i + z[i] - 1 > r)
            l = i; r = i + z[i] - 1;
    }
    return z;
}
```

Асимптотика алгоритма

- Докажем, что приведённый выше алгоритм работает за линейное относительно длины строки время, т.е. за $\mathrm{O}(N)$
- Нас интересует вложенный цикл $while$, покажем, что каждая итерация этого цикла приведёт к увеличению правой границы r на единицу.
- Для этого рассмотрим обе ветки алгоритма:
- $i > r$ В этом случае либо цикл $while$ не сделает ни одной итерации (если $s[0] \neq s[i]$), либо же сделает несколько итераций, продвигаясь каждый раз на один символ вправо, и увеличивая тем самым границу r.

Асимптотика алгоритма

- $i < r$ В этом случае мы по приведённой формуле инициализируем значение Z$[i]$ некоторым числом Z_0. Сравним это начальное значение Z_0 с величиной $r - i + 1$, получаем три варианта:
- $Z_0 < r - i + 1$ В этом случае ни одной итерации цикл $while$ не сделает.
- $Z_0 = r - i + 1$ Цикл $while$ может совершить несколько итераций, однако каждая из них будет приводить к увеличению нового значения r на единицу.
- $Z_0 > r - i + 1$ Этот случай невозможен по определению Z_0.
- Таким образом, каждая итерация вложенного цикла приводит к продвижению указателя r вправо. Т.к. r не могло оказаться больше $n - 1$, это означает, что всего этот цикл сделает не более $n - 1$ итерации.

Поиск подстроки в строке

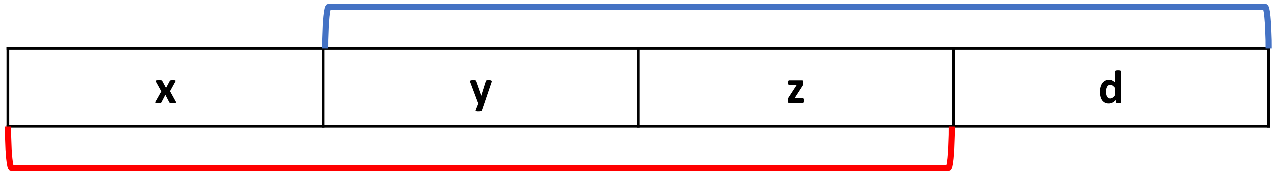
- Назовем одну строку t текстом, а другую словом w.
- Задача заключатся в том, что надо найти все вхождения слова w в текст t.
- Для решения задачи составим строку $S = w + \# + t$, т.е. после слова запишем символ разделитель и текст. Разделителем может быть любой символ который не встречается в строках w и t.
- Вычислив для текста Z-функцию, можно посчитать количество $Z[i] = length(w)$, таким образом получить количество вхождений слова w в текст t.
- Асимптотика решения $O\big(length(w) + length(t)\big)$.

Сжатие строки

- Дана строка s длины n. Требуется найти самое короткое её "сжатое" представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t.
- Для решения посчитаем Z-функцию строки s, и найдём первую позицию i такую, что i + z[i] = n, и при этом n делится на i. Тогда строку s можно сжать до строки длины i.

Сжатие строки

- Доказательство



- Пусть отрезок $[0 - i)$ – x, $[i - 2i)$ – y, $[2i - 3i)$ – z и.т.д
- Тогда $x = y,\ y = z, z = d$ как соответствующие отрезки.
- Значит для получения наименьшей «сжатой» подстроки необходимо найти первую позицию что i + z[i] = n.
- Так же n должно делится на i чтобы в строку помещались полные отрезки.

Количество различных подстрок в строке

- Дана строка S длины N.
- Требуется посчитать количество её различных подстрок.
- Научимся, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец одного символа.
- На каждой итерации добавляем в конец символ c, тогда новая длина len. Очевидно, в результате появилось len новых подстрок, оканчивающиеся на этом символе, но часть новых подстрок могла встречаться ранее, а значит их учитывать не надо. Для этого перевернем строку и вычислим Z-функцию.
- Максимальное значение Z_{max} Z-функции равно количеству подстрок которые встречались до добавления символа c.
- Тогда число новых подстрок, появляющихся при дописывании символа c, равно $len - Z_{max}$.
- Итоговая асимптотика составляет $O(N^2)$.

Хэш-функции

Хэш-функции

- Хэш-функция - функция, осуществляющая преобразование массива входных данных произвольной длины в битовую строку установленной длины, выполняемое определённым алгоритмом
- Один из лучших способов определить хэш-функцию от строки S следующий:
- $h(S) = S[0] + S[1] * P + S[2] * P^2 + S[3] * P^3 + \ldots + S[N] * P^N$
- Разумно выбирать для P простое число, примерно равное количеству символов во входном алфавите. Например, если строки состоят только из маленьких латинских букв, то хорошим выбором будет P = 31. Если буквы могут быть и заглавными, и маленькими, то, например, можно P = 53.

Хэш-функции

- Предположим, нам дана строка S, и даны индексы I и J. Требуется найти хэш от подстроки S[I..J].
- По определению имеем:
- $H[I..J] = S[I] + S[I+1] * P + S[I+2] * P\hat{}2 + \ldots + S[J] * P\hat{}(J-I)$
- откуда:
- $H[I..J] * P[I] = S[I] * P[I] + \ldots + S[J] * P[J]$,
- $H[I..J] * P[I] = H[0..J] - H[0..I-1]$
- Полученное свойство является очень важным.
- Действительно, получается, что, зная только хэши от всех префиксов строки S, мы можем за O (1) получить хэш любой подстроки.

Хэш-функции

- Однако возникает проблема, так как мы получаем $H[I..J] * P[I]$, нам нужно разделить результат на $P[I]$, а это сделать не так просто учитывая что мы получаем хэш по модулю 2^{64}.
- Более простое решение - привести сравниваемые хэши к одной степени.
- Пусть есть два хэша один умножен на $P[I]$, дугой на $P[J]$. Если $I < J$, то умножим перый хэш на $P[J - I]$, иначе же умножим второй хэш на $P[I - J]$.

Применение хэш-функции

- Поиск подстроки в строке за $O(N)$.
- Определение количества палиндромов внутри строки. $O(N \log N)$
- Определение количества различных подстрок за $O(N^2 \log N)$.

Определение количества различных подстрок

- Пусть дана строка S длиной N, состоящая только из маленьких латинских букв. Требуется найти количество различных подстрок в этой строке.
- Для решения переберём по очереди длину подстроки: $L = 1..N$.
- Для каждого L мы построим массив хэшей подстрок длины L, причём приведём хэши к одной степени, и отсортируем этот массив. Количество различных элементов в этом массиве прибавляем к ответу.
- Обычным сравнением подстрок мы бы получили алгоритм со сложностью $O(N^3)$, в то время как используя хэши, мы получим $O(N^2 + N^2 \log N)$.
- Алгоритм. Посчитаем хэш от каждой строки, и отсортируем строки по этому хэшу.

Палиндромы

Задача нахождения палиндромов

- Постановка задачи
- Дана строка S длины N. Требуется найти все такие пары (i, j), где $i < j$, что подстрока $S[i \dots j]$ является палиндромом (т.е. читается одинаково слева направо и справа налево).
- Методы решений
- Тривиальный алгоритм $\mathrm{O}(N^2)$.
- Ускорение с помощью хеширования $\mathrm{O}(N \log N)$.
- Алгоритм Манакера $\mathrm{O}(N)$.

Тривиальный алгоритм

```
vector<int> d1(n), d2(n);
for (int i = 0; i < n; ++i) {
   d1[i] = 1;
   while (i - d1[i] >= 0 && i + d1[i] < n
         && s[i - d1[i]] == s[i + d1[i]])
      ++d1[i];

   d2[i] = 0;
   while (i - d2[i] - 1 >= 0 && i + d2[i] < n
         && s[i - d2[i] - 1] == s[i + d2[i]])
      ++d2[i];
}
```

Ускорение с помощью хеширования

- Перебираем центральный элемент нашего палиндрома.
- Бинпоиском подбираем наибольший радиус палиндрома (под радиусом здесь понимается расстояние от центрального элемента до крайнего.
- Во время подбора мы должны как-то быстро сравнивать подстроки на идентичность. делаем это с помощью хешей.
- Асимптотика, как легко догадаться: N тратим на перебор центрального элемента, $log\ N$ в худшем случае тратим на подбор радиуса палиндрома, за единицу сравниваем подстроки с помощью хешей.
- Итоговая асимптотика O($NlogN$).

Алгоритм Манакера

- Научимся сначала находить все подпалиндромы нечётной длины, т.е. вычислять массив $d_1[]$ решение для палиндромов чётной длины получится модификацией этого.
- Для быстрого вычисления будем поддерживать границы (l, r) самого правого из обнаруженных подпалиндрома (т.е. подпалиндрома с наибольшим значением r). Изначально можно положить $l = 0, r = -1$.

Алгоритм Манакера

- Итак, пусть мы хотим вычислить значение $d_1[i]$ для очередного i, при этом все предыдущие значения $d_1[i]$ уже подсчитаны.
- Если i не находится в пределах текущего подпалиндрома, т.е. $i > r$, то просто выполним тривиальный алгоритм. Т.е. будем последовательно увеличивать значение $d_1[i]$, и проверять каждый раз $i - d_1[i] = i + d_1[i]$ т.е. является палиндромом. Когда мы найдём первое расхождение, либо когда мы дойдём до границ строки s — останавливаемся: мы окончательно посчитали значение $d_1[i]$. После этого мы должны не забыть обновить значения (l, r).

Алгоритм Манакера

- Рассмотрим теперь случай, когда $i < r$. Попробуем извлечь часть информации из уже подсчитанных значений $d_1[]$. А именно, отразим позицию i внутри подпалиндрома (l, r), т.е. получим позицию $j = l + (r - i)$, и рассмотрим значение $d_1[i]$. Поскольку j — позиция, симметричная позиции i, то почти всегда мы можем просто присвоить **начальное** значение $d_1[i] = d_1[j]$.

	l		j				i		r		
S:	1	2	3	2	5	2	3	2	1		
$d_1[]$	1	1	2	1	5	1	2	1	1		

Алгоритм Манакера

- Когда "внутренний палиндром" достигает границы внешнего или вылазит за неё $i + d_1[j] - 1 \leq r$. Поскольку за границами внешнего палиндрома симметрии не гарантируется, то просто присвоить $d_1[i] = d_1 [j]$ будет уже некорректно:
- В качестве **начального** значения необходимо взять:

$$\min(\ d1[l + r - i], \qquad r - i + 1\ )$$

- После этого следует пустить тривиальный алгоритм, который будет пытаться увеличить значение $d_1[i]$, пока это возможно.

Асимптотика алгоритма Манакера

- Асимптотика алгоритма Манакера $O(N)$.
- Доказательство линейности алгоритма аналогично доказательству для Z-функции.

Реализация алгоритм Манакера

```
vector<int> d1(n);
int l = 0, r = -1;
for (int i = 0; i < n; i++) {
        d1[i] = i > r ? 1 : min(d1[l + r - i], r - i + 1);
        while (i + d1[i] < n && i - d1[i] >= 0
                && s[i + d1[i]] == s[i - d1[i]])
                ++d1[i];
        if (i + d1[i]  - 1 > r)
        l = i - d1[i]  + 1, r = i + d1[i]  - 1;
}
```

Проверка скобочных выражений

Постановка задачи

- Дана последовательность из N круглых скобок. Выяснить, можно ли добавить в неё цифры и знаки арифметических действий так, чтобы получилось правильное арифметическое выражение.
- Правильная скобочная последовательность (анлг. Correct Bracket Sequences) — частный случай скобочной последовательности, определяющийся следующими образами:
- Пустая строка — есть правильная скобочная последовательность;
- Пусть S — правильная скобочная последовательность, тогда (S) — правильная скобочная последовательность;
- Пусть S_1, S_2 — правильные скобочные последовательности, тогда S_1S_2 есть правильная скобочная последовательность;

Постановка задачи

Примеры правильных скобочных последовательностей:

(( )) ( )

( )

(((( ))))

Примеры НЕправильных скобочных последовательностей:

) ( ) ( )

( ) (

((((

Решение

- Пусть $depth$ — это текущее количество открытых скобок. Изначально $depth = 0$.
- Будем двигаться по строке слева направо, если текущая скобка открывающая, то увеличим $depth$ на единицу, иначе уменьшим.
- Если при этом когда-то получалось отрицательное число, или в конце работы алгоритма $depth$ отлично от нуля, то данная строка не является правильной скобочной последовательностью, иначе является.

Решение

- Если допустимы скобки нескольких типов, то алгоритм нужно изменить.
- Вместо счётчика следует создать стек, в который будем класть открывающие скобки по мере поступления. Если текущий символ строки — открывающая скобка, то кладём его в стек, а если закрывающая — то проверяем, что стек не пуст, и что на его вершине лежит скобка того же типа, что и текущая, и затем достаём эту скобку из стека.
- Если какое-либо из условий не выполнилось, или в конце работы алгоритма стек остался не пуст, то последовательность не является правильной скобочной, иначе является.

Вычисление арифметических выражений

Постановка задачи

- Дана строка, представляющая собой математическое выражение, содержащее числа, скобки, различные операции. Требуется вычислить его значение за O (n), где n — длина строки.

Приоритет операций

- Операции в порядке убывания приоритета:
- () скобки
- * / деление и умножение
- + - сложение и вычитание
- В случае равных приоритетов операции выполняются по порядку, слева направо.

Решение задачи

- Для решения нам необходимо два стека – стек операций O, стек чисел N.
- Идем по строке S содержащей выражение, если встретили число – помещаем его в стек чисел.
- Если встретили оператор:
- В случае если последний оператор в стеке имеет больший приоритет чем рассматриваемый оператор то помещаем операцию в стек.
- Прежде чем добавить операцию с меньшим или равным приоритетом мы должны выполнить все операции с большими приоритетом (по определению приоритета).

Решение задачи

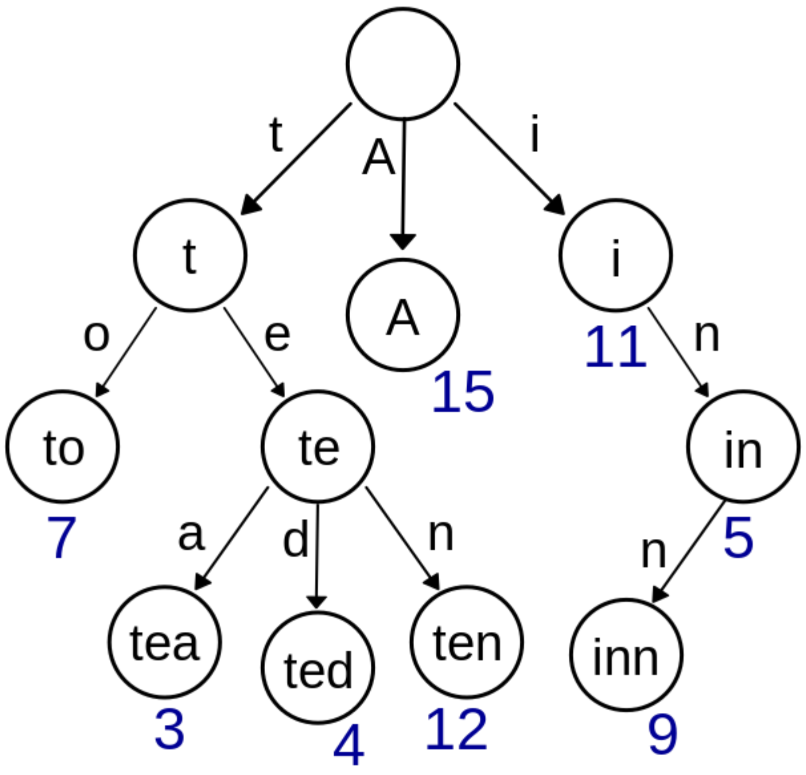
- Для выполнения операции необходимо достать оператор из стека операций, достать два числа из стека чисел, произвести соответствующую операцию между двумя числами, результат поместить в стек чисел.
- Открывающая скобка просто помещается в стек.
- При встрече закрывающей скобки мы должны выполнить все операции из стека, пока не достанем из стека открывающую скобку.
- После окончания парсинга строки необходимо выполнить операции пока стек операции не пуст.
- Результат вычисления будет храниться в стеке чисел.

Алгоритм Ахо-Корасик

Предисловие

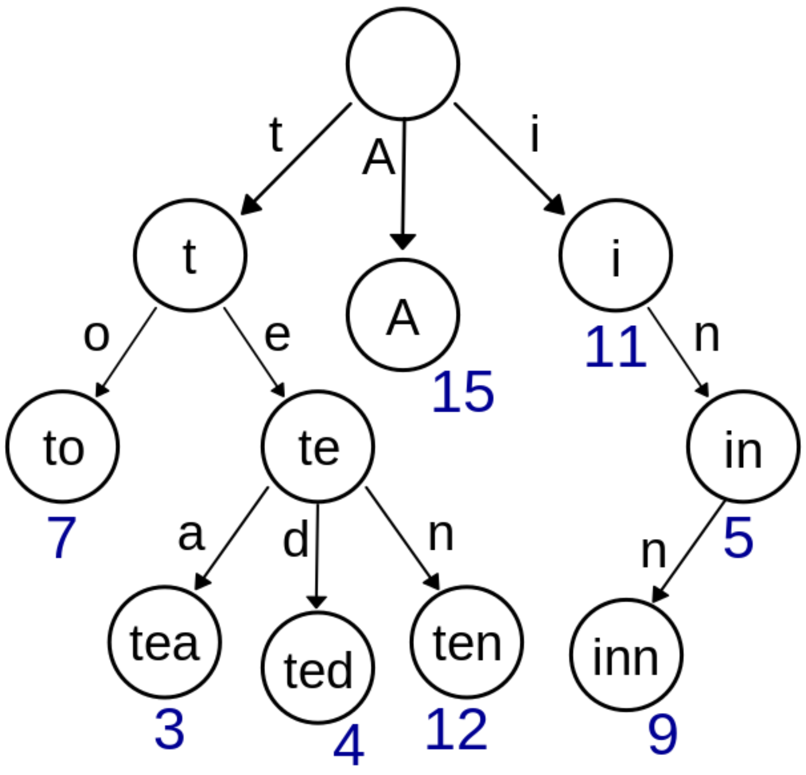
Бор

- Префиксное дерево, бор, луч, нагруженное дерево — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах.
- Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной.
- Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.



Бор

- Рассмотрим в боре любой путь из корня:
- Выпишем подряд метки рёбер этого пути. В результате мы получим некоторую строку, которая соответствует этому пути. Если же мы рассмотрим любую вершину бора, то ей поставим в соответствие строку, соответствующую пути из корня до этой вершины.



Построение бора

- Каждая вершина бора также имеет флаг $leaf$, который равен $true$, если в этой вершине оканчивается какая-либо строка из данного набора.
- Соответственно, построить бор по данному набору строк — значит построить такой бор, что каждой $leaf$-вершине будет соответствовать какая-либо строка из набора, и, наоборот, каждой строке из набора будет соответствовать какая-то $leaf$-вершина.
- Бор по набору строк строится за линейное время относительно их суммарной длины.

Построение бора

- Структура узла:

```
struct node {
  node* next[p];
  bool leaf;
};
```

- Где `next` – массив указателей на вершины соответствующих символов или `nullptr` если таких строк нет.
- `p` – количество различных символов в строке (26 для латинских символов).
- `leaf` – показывает есть ли строка заканчивающаяся в этой вершине.
- Изначально бор состоит только из одной вершины – корня.

Построение бора

Рассмотрим функцию которая будет добавлять строку S в бор.

Начинаем в корне бора, проверяем, есть ли из корня переход по букве $S[0]$:

если переход есть, то просто переходим по нему в другую вершину, иначе создаём новую вершину и добавляем переход в эту вершину по букве $S[0]$. Затем мы, стоя в какой-то вершине, повторяем процесс для буквы $S[1]$, и т.д. После окончания процесса помечаем последнюю посещённую вершину флагом $leaf = true$.

```
void add(const string& s){
   node* ptr = root;
   for (int i = 0; i < s.length(); i++) {
      char c = s[i] - 'a';
      if (ptr->next[c] == nullptr)
         ptr->next[c] = new node;
      ptr = ptr->next[c];
   }
   ptr->leaf = true;
}
```

Постановка задачи

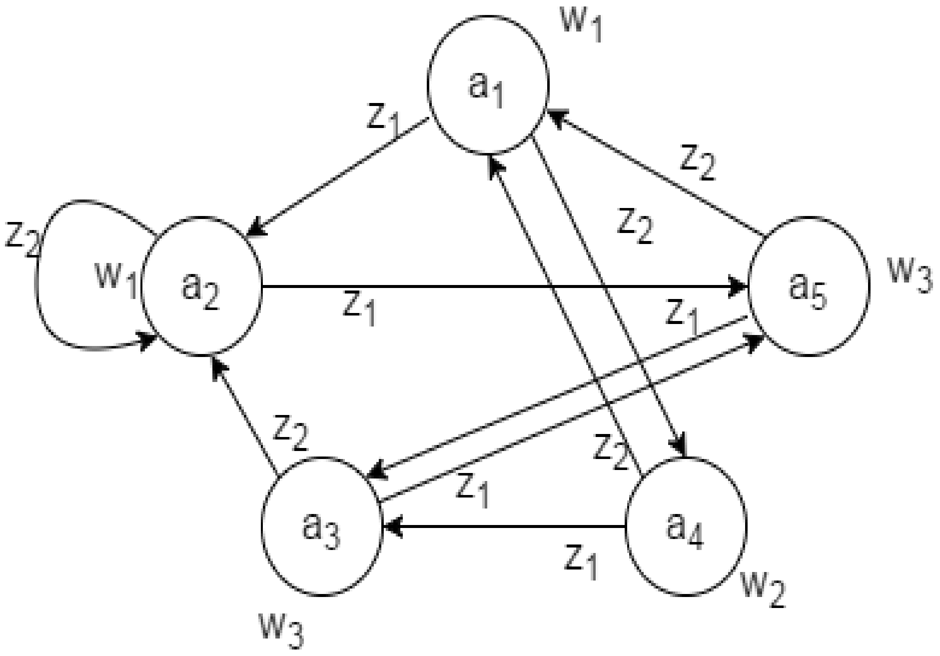
- Пусть дан набор из m строк размера k, необходимо найти все вхождения строк из набора в тексте t размера n.
- Асимптотика тривиального поиска составит O(MNK).
- Ускорение с помощью префиксной функции (Z-функция или алгоритм КМП) позволит достичь асимптотики O $(M(N+K))$. Хэширование даст аналогичную асимптотику.
- Построение паттерна – конечного автомата Ахо-Корасик - позволяет решить задачу за линейное время O$(MK+N)$.

Пример задачи

- Алгоритм Ахо-Корасик применятся для быстрого поиска набора строк в тексте.
- Например поиск вирусов в файлах и программах по базе данных вирусных сигнатур представленных в виде автомата Ахо-Корасик.

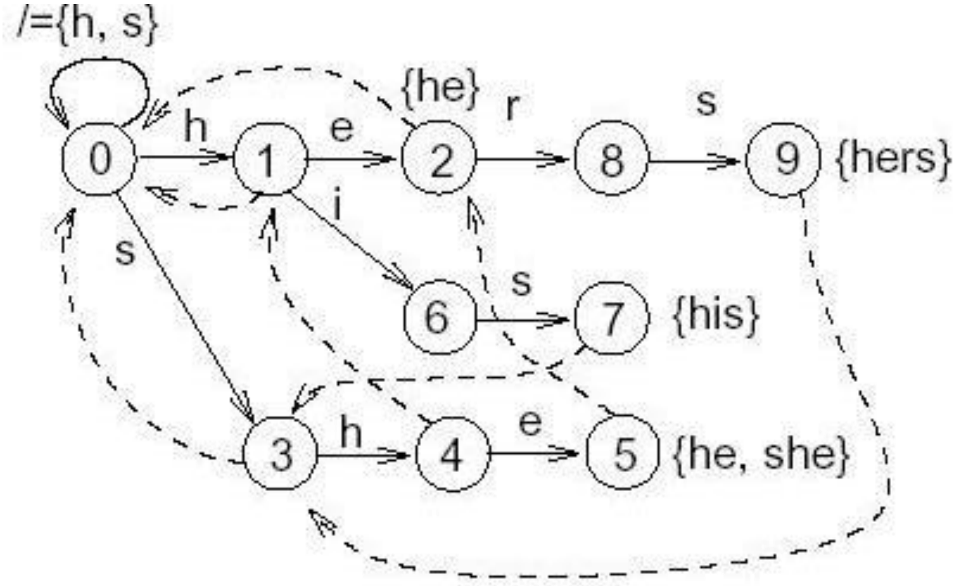
Конечный автомат

- Коне́чный автома́т (КА) — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных.
- Детерминированные КА — автоматы, в которых следующее состояние однозначно определяется текущим состоянием и выход зависит только от текущего состояния и текущего входа



Построение автомата Ахо-корасик

- Вершины бора можно понимать как состояния конечного детерминированного автомата.
- Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние — т.е. в другую позицию в наборе строк. Например, если в боре (Рисунок) мы стоим в состоянии 4 (которому соответствует строка sh), то под воздействием буквы e мы перейдём в состояние 5 (she).

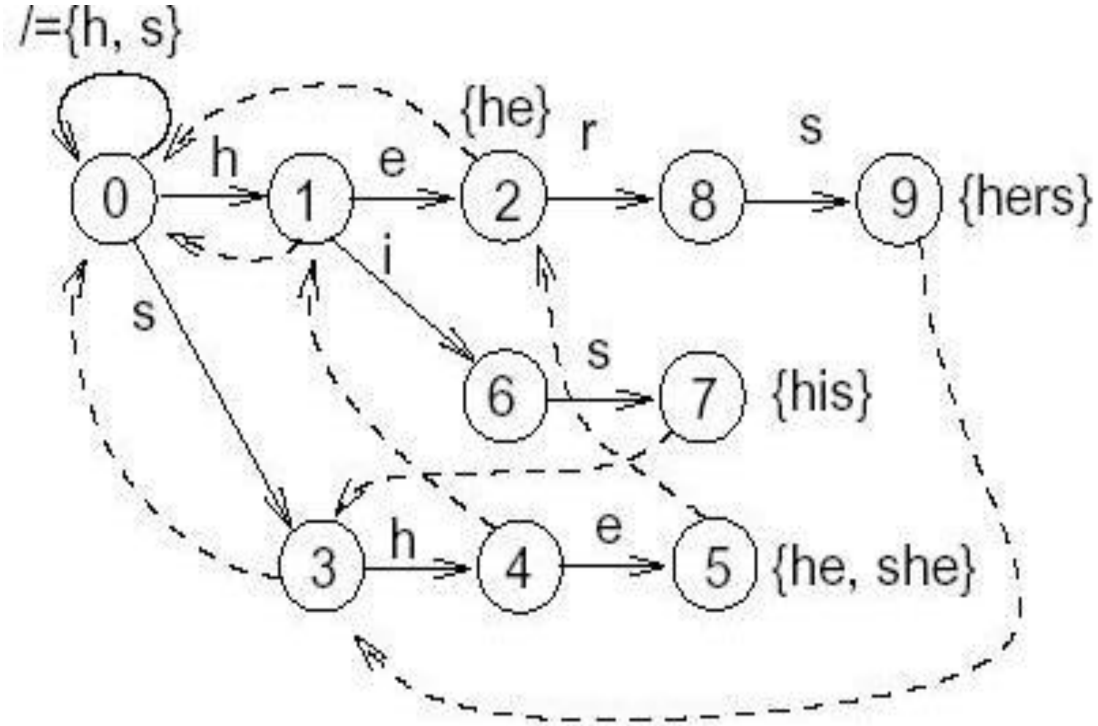


Построение автомата Ахо-корасик

- Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние.
- Пусть мы находимся в состоянии p, которому соответствует некоторая строка t, и хотим выполнить переход по символу c. Если в боре из вершины p есть переход по букве c, то мы просто переходим по этому ребру. Если же такого ребра нет, то мы должны найти состояние, соответствующее наидлиннейшему собственному суффиксу строки t и выполнить переход по букве c из него.

Построение автомата Ахо-Корасик

- Например, под воздействием строки she мы перешли в состояние 5, являющееся листом. Тогда под воздействием буквы r мы вынуждены перейти в состояние 2, соответствующее строке he , и только оттуда выполнить переход по букве r.



Построение автомата Ахо-Корасик

- **Суффиксная ссылка** для каждой вершины v — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине v. Единственный особый случай — корень бора; для удобства суффиксную ссылку из него проведём в себя же.
- Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке.

Построение автомата Ахо-Корасик

- Заметим, что если мы хотим узнать суффиксную ссылку для некоторой вершины v, то мы можем перейти в предка p текущей вершины (пусть c — буква, по которой из p есть переход в v), затем перейти по его суффиксной ссылке, а затем из неё выполнить переход в автомате по букве c.
- Таким образом, задача нахождения перехода свелась к задаче нахождения суффиксной ссылки, а задача нахождения суффиксной ссылки — к задаче нахождения суффиксной ссылки и перехода, но уже для более близких к корню вершин. Мы получили рекурсивную зависимость, но не бесконечную, и, более того, разрешить которую можно за линейное время.

Построение автомата Ахо-Корасик

- Для каждой вершины необходимо хранить её предка p, а также символ ch, по которому из предка есть переход в нашу вершину. Также в каждой вершине будем хранить $node^*\ link$ — суффиксная ссылка (или $nullptr$, если она ещё не вычислена), и массив $node^* go[k]$ — переходы в автомате по каждому из символов (опять же, если элемент массива равен $nullptr$, то он ещё не вычислен).

Поиск набора строк в тексте

- Построим по данному набору строк бор. Будем теперь обрабатывать текст по одной букве, перемещаясь— по состояниям автомата. Изначально мы находимся в корне дерева.
- Пусть мы на очередном шаге мы находимся в состоянии v, и очередная буква текста c. Тогда следует переходить в состояние $go(v, c)$, тем самым либо увеличивая на 1 длину текущей совпадающей подстроки, либо уменьшая её, проходя по суффиксной ссылке.
- Если мы стоим в помеченной вершине ($leaf = true$), то имеется совпадение с тем образцом, который в боре оканчивается в вершине v.

Поиск набора строк в тексте

- Однако это далеко не единственный возможный случай достижения совпадения: если мы, двигаясь по суффиксным ссылкам, можем достигнуть одной или нескольких помеченных вершин, то совпадение также будет, но уже для образцов, оканчивающихся в этих состояниях.
- Простой пример такой ситуации — когда набор строк — это $\{dbac, abc, bc\}$, а текст — это $dabc$.
- Таким образом, за $\mathrm{O}(n)$ можем найти номера всех образцов, для которых достигнуто совпадение, просто пройдя по суффиксным ссылкам от текущей вершины до корня. Однако это недостаточно эффективное решение, поскольку в сумме асимптотика получится $\mathrm{O}(n\ Len)$.

Поиск набора строк в тексте

- Можно заметить, что движение по суффиксным ссылкам можно оптимизировать, предварительно посчитав для каждой вершины ближайшую к ней помеченную вершину, достижимую по суффиксным ссылкам (это называется "функцией выхода").
- Эту величину можно считать ленивой динамикой за линейное время. Тогда для текущей вершины мы сможем за $O(1)$ находить следующую в суффиксном пути помеченную вершину, т.е. следующее совпадение. Тем самым, на каждое совпадение будет тратиться $O(1)$ действий, и в сумме получится асимптотика $O(Len + Ans)$.
- Тогда $leaf\_link$ — это ближайший суффикс, имеющийся в боре, для которого $leaf = true$.

Использованные источники

- [1] MAXimal - Электронный ресурс: Z-функция строки и её вычисление. Режим доступа: https://e-maxx.ru/algo/z_function.
- [2] MAXimal - Электронный ресурс: Алгоритмы хэширования в задачах на строки. Режим доступа: https://e-maxx.ru/algo/string_hashes.
- [3] MAXimal - Электронный ресурс: Нахождение всех подпалиндромов. Режим доступа: https://e-maxx.ru/algo/palindromes_count.
- [4] Habr - Электронный ресурс: Алгоритм Ахо-Корасик. Режим доступа: https://habr.com/ru/post/198682/.
- [5] MAXimal - Электронный ресурс: Алгоритм Ахо-Корасик. Режим доступа: https://e-maxx.ru/algo/aho_corasick.