**Алгоритм Кнута - Морриса - Пратта**

Реферат

Алгоритм Кнута-Морриса-Пратта (КМП) получает на вход слово

X=x[1]x[2]... x[n]

и просматривает его слева направо буква за буквой, заполняя при этом массив натуральных чисел l[1]... l[n], где

l[i]=длина слова l(x[1]...х[i])

(функция l определена в предыдущем пункте). Словами: l[i] есть длина наибольшего начала слова x[1]...x[i], одновременно являющегося его концом.

Какое отношение все это имеет к поиску подслова?

Другими словами, как использовать алгоритм КМП для определения того, является ли слово A подсловом слова B?

Решение. Применим алгоритм КМП к слову A#B, где # - специальная буква, не встречающаяся ни в A, ни в B. Слово A является подсловом слова B тогда и только тогда, когда среди чисел в массиве l будет число, равное длине слова A.

Описать алгоритм заполнения таблицы l[1]...l[n].

Решение. Предположим, что первые i значений l[1]...l[i] уже найдены. Мы читаем очередную букву слова (т.е. x[i+1]) и должны вычислить l[i+1].

Другими словами, нас интересуют начала Z слова

x[1]...x[i+1,

одновременно являющиеся его концами -из них нам надо брать самое длинное. Откуда берутся эти начала? Каждое из них (не считая пустого) получается из некоторого слова Z' приписыванием буквы x[i+1] . Слово Z' является началом и

концом слова x[1]...x[i]. Однако не любое слово, являющееся началом и концом слова x[1]...x[i], годится - надо, чтобы за ним следовала буква x[i+1].

Получаем такой рецепт отыскания слова Z. Рассмотрим все начала слова x[1]...x[i], являющиеся одновременно его концами. Из них выберем подходящие - те, за которыми идет буква x[i+1]. Из подходящих выберем самое длинное. Приписав в его конец х[i+1], получим искомое слово Z. Теперь пора воспользоваться сделанными нами приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l из предыдущего раздела.

Вот что получается:

i:=1; 1[1]:=0;

{таблица l[1]..l[i] заполнена правильно}

while i <> n do begin

len:= l[i]

{len - длина начала слова x[1]..x[i], которое является

его концом; все более длинные начала оказались

неподходящими}

while (x[len+1]<>х[i+1]) and (len>0) do begin

{начало не подходит, применяем к нему функцию l}

len:=l[len];

end;

{нашли подходящее или убедились в отсутствии}

if x[len+1]=x[i+1] do begin

{х[1]..x[len] - самое длинное подходящее начало}

l[i+1]:=len+1;

end else begin

{подходящих нет}

l[i+1]:= 0;

end;

i:=i+1;

end;

Доказать, что число действий в приведенном только что алгоритме не превосходит Cn для некоторой константы C.

Решение. Это не вполне очевидно: обработка каждой очередной буквы может потребовать многих итераций во внутреннем цикле. Однако каждая такая итерация уменьшает len по крайней мере на 1, и в этом случае l[i+1] окажется заметно меньше l[i]. С другой стороны, при увеличении i на единицу величина l[i] может возрасти не более чем на 1, так что часто и сильно убывать она не может - иначе убывание не будет скомпенсировано возрастанием.

Более точно, можно записать неравенство

l[i+1]<l [i] - (число итераций на i-м шаге)+1

или

(число итераций на i-м шаге)<= l[i]-l[i+1]+1

Остается сложить эти неравенства по всем i и получить оценку

сверху для общего числа итераций.

Будем использовать этот алгоритм, чтобы выяснить, является ли слово X длины n подсловом слова Y длины m. (Как это делать с помощью специального разделителя #, описано выше.) При этом число действий будет не более C(n+m}, и используемая память тоже. Придумать, как обойтись памятью не более Cn (что может быть существенно меньше, если искомый образец короткий, а слово, в котором его ищут - длинное).

Решение. Применяем алгоритм КМП к слову А#В. При этом: вычисление значений l[1],...,l [n] проводим для слова X длины n и запоминаем эти значения. Дальше мы помним только значение l[i] для текущего i - кроме него и кроме таблицы

l[1]...l[n], нам для вычислений ничего не нужно.

На практике слова X и Y могут не находиться подряд, поэтому просмотр слова X и затем слова Y удобно оформить в виде разных циклов. Это избавляет также от хлопот с разделителем.

Написать соответствующий алгоритм (проверяющий, является ли слово X=x[1]...x[n] подсловом слова Y=y[1]...y[m]

Решение. Сначала вычисляем таблицу l[1]...l[n]как раньше. Затем пишем такую программу:

j:=0; len:=0;

{len - длина максимального качала слова X, одновременно

являющегося концом слова y[1]..j[j]}

while (len<>n) and (j<>m) do begin

while (x[len+1]<>у[j+1]) and (len>0) do begin

{начало не подходит, применяем к нему функцию l}

len: = l[len];

end;

{нашли подходящее или убедились в отсутствии}

if x[len+1]=y[j+1] do begin

{x[1]..x[len] - самое длинное подходящее начало}

len:=len+1;

end else begin

{подходящих нет}

len:=0;

end;

j:=j+1;

end;

{если len=n, слово X встретилось; иначе мы дошли до конца

слова Y, так и не встретив X}

Алгоритм Бойера - Мура

Этот алгоритм делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный образец. Как так может быть? Идея проста. Пусть, например, мы ищем образец abcd. Посмотрим на четвертую букву слова: если, к примеру, это буква e, то нет никакой необходимости читать первые три буквы. (В самом деле, в образце буквы e нет, поэтому он может начаться не раньше пятой буквы.)

Мы приведем самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть x[1]...х[n] - образец, который надо искать. Для каждого символа s найдем самое правое его вхождение в слово X, то есть наибольшее k, при котором х[k]=s. Эти сведения будем хранить в массиве pos[s]; если символ s вовсе не встречается, то нам будет удобно положить pos[s]=0 (мы увидим дальше, почему).

Как заполнить массив pos?

Решение.

положить все pos[s] равными 0

for i:=1 to n do begin

pos[x[i]]:=i;

end;

В процессе поиска мы будем хранить в переменной last номер буквы в слове, против которой стоит последняя буква образца. Вначале last=n (длина образца), затем last постепенно увеличивается.

last:=n;

{все предыдущие положения образца уже проверены}

while last<= m do begin {слово не кончилось}

 if x[m]<>y[last] then begin {последние буквы разные}

last:=last+(n-pos[y[last]]);

{n - pos[y[last]] - это минимальный сдвиг образца,

при котором напротив y[last] встанет такая же

буква в образце. Если такой буквы нет вообще,

то сдвигаем на всю длину образца}

end else begin

если нынешнее положение подходит, т.е. если

x[i]..х[n]=y[last-n+1]..y[last],

то сообщить о совпадении;

last:=last+1;

end;

end;

Знатоки рекомендуют проверку совпадения проводить справа налево, т.е. начиная с последней буквы образца (в которой совпадение заведомо есть). Можно также немного сэкономить, произведя вычитание заранее и храня не pos[s], а n-pos[s],

т.е. число букв в образце справа от последнего вхождения буквы Возможны разные модификации этого алгоритма. Например, можно строку

last:=last+i

заменить на

last:=last+(n-u),

где u - координата второго справа вхождения буквы x[n] в образец.

Как проще всего учесть это в программе

Решение. При построении таблицы pos написать

for i:=1 to n-1 do...

(далее как раньше), а в основной программе вместо

last:=last+1

написать

last:=last+n-pos[y[last]];

Приведенный упрощенный вариант алгоритма Бойера-Мура в некоторых случаях требует существенно больше n действий (число действий порядка mn), проигрывая алгоритму Кнута-Морриса-Пратта.

Пример ситуации, в которой образец не входит в слово, но алгоритму требуется порядка mn действий, чтобы это установить.

Решение. Пусть образец имеет вид baaa... aa, а само слово состоит только из букв а. Тогда на каждом шаге несоответствие выясняется лишь в последний момент.

Настоящий (не упрощенный) алгоритм Бойера-Мура гарантирует, что число действий не превосходит C(m+n) в худшем случае. Он использует идеи, близкие к идеям алгоритма Кнута-Морриса-Пратта. Представим себе, что мы сравнивали образец со входным словом, идя справа налево. При этом некоторый кусок Z (являющийся концом образца) совпал, а затем обнаружилось различие: перед Z в образце стоит не то, что во входном слове. Что можно сказать в этот момент о входном слове? В нем обнаружен фрагмент, равный Z, а перед ним стоит не та буква, что в образце. Эта информация может позволить сдвинуть образец на несколько позиций вправо без риска пропустить его вхождение. Эти сдвиги следует вычислить заранее для каждого конца Z нашего образца. Как говорят знатоки, все это (вычисление таблицы сдвигов и ее использование) можно уложить в C(m+ n) действий.

**Алгоритм Рабина**

Этот алгоритм основан на простой идее. Представим себе, что в слове длины m мы ищем образец длины n. Вырежем окошечко размера n и будем двигать его по входному слову. Нас интересует, не совпадает ли слово в окошечке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определенную на словах длины n. Если значения этой функции на слове в окошечке и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

В чем выигрыш при таком подходе. Казалось бы, ничего - ведь чтобы вычислить значение функции на слове в окошечке, все равно нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, и вот за счет чего. При сдвиге окошечка слово не меняется полностью, а лишь добавляется буква в конце и убирается в начале. Хорошо бы, чтобы по этим данным можно было рассчитать, как меняется функция.

Привести пример удобной для вычисления функции.

Решение. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошечка нужно добавить новое число и вычесть пропавшее.)

Для каждой функции существуют слова, к которым она применима плохо. Зато другая функция в этом случае может работать хорошо. Возникает идея: надо запасти много функций и в начале работы алгоритма выбирать из них случайную. (Тогда враг, желающий подгадить нашему алгоритму, не будет знать, с какой именно функцией ему бороться.)

Привести пример семейства удобных функций.

Решение. Выберем некоторое число p (желательно простое, смотри далее) и некоторый вычет x по модулю p. Каждое слово длины n будем рассматривать как последовательность целых чисел (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени n-1 и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя функция). Сдвиг окошка на 1 соответствует вычитанию старшего члена (хn-1 следует вычислить заранее), умножению на x и добавлению свободного члена.

Следующее соображение говорит в пользу того, что совпадения не слишком вероятны. Пусть число p фиксировано и к тому же простое, а X и Y - два различных слова длины n. Тогда им соответствуют различные многочлены (мы предполагаем, что коды всех букв различны - это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, то есть их разность обращается в 0. Разность есть многочлен степени n-1 и имеет не более n-1 корней. Таким образом, если и много меньше p, то случайному x мало шансов попасть в неудачную точку.