**Описание идеи и реализации алгоритма Бойера-Мура v1.0**

Прежде чем переходить к описанию алгоритма, определим формат строковых данных, с которыми будем работать. Для реализации возможности работы с произвольным алфавитом реализован модуль abc.cpp. В данном модуле присутствует функция int abc\_set(char \*s), которая принимает на входе строку с перечислением всех символов используемого алфавита и задаёт соответствие между кодировкой ASCII и внутренней кодировкой, используемой алгоритмом. Неапример, если подать на вход строку “ABXYZ”, то внутренний код «A» будет равен 0, «B» - 1, «X» - 2 и тд…

Строки в алгоритме представлены массивами целых чисел. Преобразование из строки в массив и обратно осуществляются с помощью функций int abc\_str\_to\_code( char\* src, int\*\* dst ) и void abc\_code\_to\_str( int\* src, char\*\* dst, int len ) соответственно. Внедрение своей кодировки не является необходимым, но позволяет сделать алгоритм более универсальным и «чистым», а также не формировать таблицу плохих символов для всех символов ASCII.

На рисунке 1 изображено представление входных данных алгоритма Бойера-Мура. Имеется строка str, длинны str\_len и шаблон tem длинны tem\_len. Алгоритм ищет все вхождения шаблона в строку. Буквой p обозначено текущее смещение шаблона относительно строки.

C

str:

B

A

A

B

C

A

C

A

A

B

C

A

tem:

str\_len

tem\_len

p

*Рисунок 1. Описание исходных данных*

Начнём с того, как работает самый простой алгоритм поиска подстроки в строке. Сначала мы совмещаем начала строки с началом шаблона (p=0). Затем в цикле, начиная с первого символа шаблона и до последнего сравниваем соответствующие символы строки и шаблона. Если все символы совпали, значит подстрока найдена. После того, как мы нашали или не нашли совпадение шаблона со строкой, смещаем шаблон на одну позицию вправо (p++) и повторяем операцию до тех пор, пока правая граница шаблона не достигнет конца строки.

Теперь немного изменим этот алгоритм. Шаблон будем всё так-же двигать слева на право, а сравнивать символы шаблона со строкой справа налево (от последнего к первому). Это никак не повлияет на работу алгоритма. Тем самым мы получили заготовку для Алгоритм Бойера-Мура.

Алгоритм Бойера-Мура использует две эвристические функции, которые позволяют в некоторых случаях смещать шаблон более, чем на один шаг вправо, что экономит вычислительные ресурсы. Эти функции независимы и могут использоваться по отдельности или совместно. Каждая из этих функций выдаёт минимальное безопасное смещение шаблона вправо, такое, что не будет пропущено не одно вхождение шаблона в строку. Если функции используются совместно, то выбирается наибольшее значение смещения.

Сначала рассмотрим функцию плохого символа. Представим ситуацию, как на рисунке 2. При p=1 мы начали сравнивать строку с шаблонам с последнего символа и сразу нашли несовпадение. Это означает, что шаблон не совпадает со строкой при p=1. По идее мы должны сдвинуть шаблон на один шаг вправо и начать следующее сравнение, НО мы видим, что в исходной строке на месте не совпавшего символа стоит «С», а в шаблоне последний и предпоследний символ не равны «С». Это значит, что и при p=2 шаблон не будет совпадать со строкой. Первое совпадение возможно только в том случае, если на месте последнего символа строки окажется «С», а это произойдет только тогда, когда шаблон сдвинется на две позиции вправо, поскольку это пододвинет последнее вхождение символа «С» шаблона к последнему символу строки также равному «С».

A

str:

B

D

D

A

C

A

B

С

B

C

A

B

tem:

str\_len

tem\_len

p

*Рисунок 2. Плохой символ*

Мы можем заранее определить для каждого символа алфавита на каком смещении от конца шаблона он встречается первый раз. Для символа «C», это значение равно 2. Теперь каждый раз, когда мы встретим в качестве последнего символа строки, символ «С», мы сможем сместить шаблон на две позиции вправо, поскольку для совпадения последний символ строки «С» должен оказаться над символом «С» шаблона. Если мы сместим не до последнего вхождения «С» в шаблон, а до первого, то есть риск пропустить одно из вхождений. Определим значения сдвига для всех символов алфавита, рисунок 3. Обратим внимание на две детали. Мы не учитываем последний символ «B» - это не имеет смысла, поскольку если последний символ строки равен «B», то полученное смещение равно 0. Мы уже проверили совпадение шаблона со строкой на позиции p и вне зависимости от результата нет смысла оставаться на месте. Именно поэтому мы игнорируем последний символ и используем смещение предпоследнего. Во вторых в шаблоне отсутствует символ «D». Ну тут всё просто. Раз этого символа нету в шаблоне, то можно сдвинуть шаблон на всю его длину. Логика в том, что если мы встретили символ, которого не было в шаблоне, то пока последний символ строки будет стоять над шаблоном, вхождения не будет и нужно сдвинуть шаблон на всю его длину.

С

B

C

A

B

tem:

A

bad char table (tabBC):

1

B

3

C

2

D

5

*Рисунок 3. Bad table*

С этим разобрались. Но что будет, если не совпавший символ будет не первым, а как показано на рисунке 4? Сразу отметим, что совпавшую подстроку «AC» будем в дальнейшем называть суффиксом, но сейчас сам суффикс нас не интересует. Символ шаблона «D» находится на смещении 3 от конца шаблона. Но если мы сдвинем шаблон на 3 клетки, то промажем и «D» шаблона окажется правее «D» из строки. Выходит, что функция плохого символа неприменима при непустом суффиксе? Не совсем. Нас просто нужно учитывать смещение плохого символа от конца последнего (относительно шаблона) символа строки. Расчитываем итоговое смещение по формуле: tabBC[«D»] – suff\_len = 3 – 2 = 1.

A

str:

B

D

D

A

C

A

B

С

D

C

A

С

tem:

str\_len

tem\_len

p

suff\_len

*Рисунок 4. Суффикс и плохой символ*

Отлично! Но что, если произойдёт ситуация, как на рисунке 5, когда последнее вхождение символа шаблона будет правее или равно позиции плохого символа? 1 – 2 = -1 … хмм отрицательное смещение. Как упоминалось ранее смещение <=0 не имеет смысла, поэтому просто смещаемся на один шаг. В теории можно сделать двумерный массив tabBS[suff\_len][char] и найти последнее вхождение каждого символа для каждой длинны суффикса, чтобы сделать алгоритм эффективнее. Кстати значение нулевой строки такой матрицы (суффикс пуст) равно нашей обычно таблице. Но такой подход потребует уже tem\_len\*abc\_len памяти, что не круто. Можно исхитриться и обойтись tem\_len+abc\_len памяти, и использовать бинарный поиск, вместо обращения к ячейке по индексу, но это уже выходит за рамки рассказа. В любом случае алгоритм Бойера-Мура не использует двумерный массив.

A

str:

B

D

D

D

C

A

B

С

D

C

D

С

tem:

str\_len

tem\_len

p

suff\_len

*Рисунок 5. Суффикс и плохой символ*

С плохим символом вроде разобрались, что теперь насчёт хорошего суффикса. Взглянем на рисунок 6. У нас совпал суффикс «AB» и нам нужно сдвинуть шаблон таким образом, чтобы на месте нашего суффикса «AB» оказалось последнее вхождение в шаблон этого же суффикса. На рисунке 6 синим цветом отмечен суффикс, а зелёным последнее вхождение этого суффикса в шаблон. Последнее вхождение находиться на смещении 3 от конца шаблона (смещение считаем относительно последнего символа вхождения), поэтому и сам шаблон мы сдвигаем на 3 шага.

A

str:

B

D

D

C

A

B

B

A

B

D

A

tem:

str\_len

tem\_len

p

suff\_len

C

A

B

B

B

*Рисунок 6. Хороший суффикс*

Вроде элементарно, а теперь рассмотрим некоторые моменты. Последнее вхождение может наезжать на суффикс, но должно быть левее конца шаблона хотя бы на один шаг. Если полное вхождение суффикса не найдено, то нужно найти префикс шаблона максимальный длинны, который совпадает с суффиксом суффикса. Звучит немного запутанно, но на деле всё просто. Взглянем на рисунок 7. У нас есть суффикс «ABC» больше этот суффикс целиком не встречается. Но если мы сдвинем шаблон на всю его длину, то можем пропустить вхождение шаблона в строку. Поэтому если мы не нашли полное вхождение суффикса в шаблона, то нужно найти префикс максимальной длинны, который будет совпадать с суффиксом. В данном примере это префикс «BC». Если оказалось так, что такого префикса нету, то только в том случае можно сдвинуть шаблон на всю его длину. Как в случае нахождения полного вхождения суффикса, так и совпадения части суффикса с префиксом мы сдвигаем шаблон на смещение найденного фрагмента относительно конца шаблона. Смещение считаем от последнего символа фрагмента.

A

str:

B

D

D

A

B

C

D

C

D

A

B

tem:

str\_len

tem\_len

p

suff\_len

A

B

C

C

B

*Рисунок 7. Префикс*

Есть ещё одна вещь без которой алгоритм будет работать, но в некоторых случаях – это значительно снизит производительность. Вернёмся к рисунку 6. Есть суффикс длинны 2 и перед этим суффиксом в шаблоне стоит «D». Очевидно, что раз длинна шаблона равна 2, а не 3 и более, то «D» не равно соответствующему символу в строке, в конкретном случае «C». Давайте представим, что шаблон имеет вид «DABDAB», если мы просто найдём вхождение «AB» со смещением 3, то мы не учтём, что раз у нас соответствующий символ строки (НЕ <D>), то и предыдущее вхождение «AB», перед котором стоит «D» никак не может привести к тому, что шаблон целиком совпадёт со строкой. Отсюда следует правило, что при поиске предыдущих вхождений, если перед суффиксом строит символ X, то в его искомом предыдущем вхождении, перед первым символом этого вхождения должен быть символ отличный от X или его может не быть вовсе.

При операции поиска шаблона в строке единственная переменная часть - это длинна суффикса, поэтому аналогично плохому символу мы создаём таблицу хороших суффиксов. Длинна таблицы будет равна tem\_len+1. Индекс таблицы – это длинна совпавшего суффикса от 0 до tem\_len включительно.

Попробуем реализовать самый простой алгоритм, В коде это функция bm\_tabGS\_slow. Для каждой длинны суффикса будем просматривать подстроки длинны искомого суффикса справа налево. Начнём со смещения 1 от конца. Напомню, что смещение считается от последнего символа подстроки. Если все символы подстроки совпадают с суффиксом, то мы запоминаем это смещение в таблицу и переходим к следующей длине суффикса. Если мы левой границей подстроки уже вышли за край шаблона, то продолжаем работу и все символы «за границей» считаем совпавшими. Это позволяет учитывать частичное совпадение с префиксом. В конечном счёте совпадение всегда будет найдено, ведь когда подстрока целиком выйдет за левую границу шаблона, все символы условно говоря совпадут, на деле это означает, что для суффикса заданной длинны не найдено его вхождений или вхождений суффикса суффикса в префикс.

У этого алгоритма есть огромный недостаток, а именно сложность O(n3) попробуем снизить её до квадрата. Сначала проедемся по префиксам. Для каждой длинны префикса ищем совпадение этого префикса с суффиксом. Если совпало записываем смещение в таблицу. Вот только если мы нашли, что префикс длинны n равен суффиксу, то и все суффиксы длины >n также будут включать в себя этот префикс. Можно конечно записать найденное смещение для всех суффиксов от n и до tem\_len, но мы схитрим и введём переменную shift, куда будем записывать смещение последнего найденного совпадения префикса и суффикса. Если в дальнейшем для большего суффикса мы найдём другой префикс, то запишем и запомним его смещение, если нет, то используем смещение ранее найдённого префикса.

Пример на рисунке 8. На шаге 2 мы нашли совпадение префикса и суффикса. После нахождения совпадения мы запомнили смещение для префикса длинны 2. Само смещение равно 6. Этот префикс и его смещение актуально для всех суффиксов длинной больше или равных 2. На шаге 3 мы не нашли совпадения и использовали найденный на предыдущем шаге префикс длинны 2. На шаге 4 мы уже нашли совпадение длинной 4 и использовали его, а не предыдущий результат. Шаг 4 не последний!

B

A

B

A

step 1:

B

A

A

B

shift = 8

tabGS[1] = 8

B

A

B

A

step 2:

B

A

A

B

shift = 6

tabGS[2] = 6

B

A

B

A

step 3:

B

A

A

B

shift = 6

tabGS[3] = 6

B

A

B

A

step 4:

B

A

A

B

shift = 4

tabGS[4] = 4

B

A

B

A

step 0:

B

A

A

B

shift = 8

tabGS[0] = 8

*Рисунок 8. поиск префиксов*

После поисков префиксов, перейдём к поиску полных вхождений. Проходим по шаблону слева направо и на каждом шаге смотрим, сколько символов подстроки совпало с суффиксом. Сравниваем подстроки с суффиксом посимвольно (с последнего до нулевого) и на каждом совпавшем символе записываем смещение в таблицу. Звучит запутанно, но на деле всё проще. Допустим мы сравнили суффикс длинны n с подстрокой и они совпали, отсюда следует, что для сравнения суффикса длинны n+1 с подстрокой длинный n+1 достаточно сравнить всего один символ! Именно это обеспечивает понижение сложности алгоритма с О(n3) до О(n2).

Также не забываем перед записью в таблицу убедится, что символ перед подстрокой не равен символу перед суффиксом или не существует.

Кстати можно объединить поиск префиксов и полных вхождений в один цикл, поскольку обе этих «половины» на каждой итерации работают с подстроками на одном смещении от конца шаблона. В результате они не перебивают результаты друг друга.

Существует алгоритм построения таблицы за O(n), но это как-нибудь потом)))