Краткий конспект

Лекция 5. Преобразование Барроуза-Уилера

версия 0.1(draft)

Д. Ищенко* Б. Коварский* И. Алтухов* Д. Алексеев*

28 марта, 2016

*МФТИ

1 Оценка памяти необходимой для хранения суффиксного дерева

На предыдущей лекции мы обсуждали структуру суффиксного дерева и показали, что с помощью него можно решить задачу поиска k мотивов длины m в геноме за $O(km)$. Это очень «хорошая» временная сложность и, казалось бы, что можно еще совершенстовать? Зайдем с другой стороны и оценим, какая память необходима для хранения суффиксного дерева? Мы показали, что для последовательности длины n, кол-во суффиксов, а значит и листьев в дереве – n штук. Каждый внутренний узел дерева, в результате ветвления от него, добавляет как минимум один *новый* лист в дерево, т.е. внутренних листов в дереве не больше n. Таким образом всего в дереве не больше $2n$ узлов и, соответственно, ребер. Для каждой метки ребра нам нужно хранить 2 числа (начало и конец подпоследовательности в геноме), т.е. для хранения всех меток, нам потребуется $2n \cdot 2 = 4n$ чисел и сам геном длины n. В итоге, для хранения всего дерева нам необходимо $2n+4n+n = 7n$ чисел, а это в семь раз больше, чем требуется для хранения просто последовательности генома.

Например, в случае генома человека длиной $3 \cdot 10^9$ нуклеотидов, нам было бы необходимо выделить $3Gb \cdot 7 = 21Gb$ оперативной памяти, что, естественно, невозможно сделать на обыкновенном персональном компьютере. Можем ли мы решить эту проблему? Оказывается, что да. Есть несколько вариантов, один из них – это суффиксные массивы, мы же будем говорить сегодня о *преобразованиях Барроуза-Уилера*. Но прежде, чем перейти непосредственно к преобразованию, разберемся, как мы можем компактно хранить информацию о строках.

2 Сжатие данных

Рассмотрим следующую строку длиной 34:

```
AAAAAAAATTTTTTTGGGGGGGAAAAAAAACCCC
```

Если хранить ее просто, как набор чисел, то нам потребуется 34 байта. Можем ли мы уменьшить это кол-во, не потеряв при этом информации? Можем:

```
8A 7T 7G 8A 4C
```

Такая краткая запись позволит нам хранить строку, используя всего 10 байт. Важно отметить, что взяв сокращенную запись строки, мы легко можем восстановить исходную. Собственно, этот пример и демонстрирует один из простейших вариантов сжатия данных. Но, что если мы возьмем следующую строку длиной в 8 символов:

```
TATATAGA
```

Применив вышеописанный подход, мы получим:

```
1T 1A 1T 1A 1T 1A 1G 1A
```

В рассмотренном случае мы даже увеличили необходимую для хранения строки память. Конечно, мы можем заметить, что в строке идут повторяющиеся блоки TA и записать следующее:

```
3TA 1GA
```

Но каким образом находить все подобные повторяющиеся блоки? Сколько времени это займет? А как быть, если они идут не подряд? Как работать потом со сжатой таким образом строкой и искать в ней мотивы? Возникает много вопросов и ответом на них и решением задачи сжатия как раз и является преобразование Барроуза-Уилера.

3 Преобразование Барроуза-Уилера

Рассмотрим строку S = TATATAGA. Как и при рассмотрении суффиксных деревьев, добавим в конец исследуемой последовательности символ \$. И рассмотрим набор из всех циклических перестановок строки. Он будет иметь следующий вид:

```
T A T A T A G A $
A T A T A G A $ T
T A T A G A $ T A
A T A G A $ T A T
T A G A $ T A T A
A G A $ T A T A T
G A $ T A T A T A
A $ T A T A T A G
$ T A T A T A G A
```

Введем следующее свойтсво алфавита: $\$ < A < C < G < T$. Эта свойство позволяет нам сравнивать между собой символы и производить сортировку строк одинаковой длины. Например две строки ATG и ATA в осортированном порядке:

```
ATA
ATG
```

Первые два символа у них совпадают, а третий $A < G$, поэтому строка ATA идет перед ATG.

Отсортируем все циклические перестановки строки S. Получим следующий набор строк (его можно представить матрицей M размера $n \times n$).

```
F               L

$ T A T A T A G A
A $ T A T A T A G
A G A $ T A T A T
A T A G A $ T A T
A T A T A G A $ T
G A $ T A T A T A
T A G A $ T A T A
T A T A G A $ T A
T A T A T A G A $
```

Назовем первый столбец матрицы – **F** («first») и последний – **L** («last»). Очевидно, что первый столбец состоит из подряд идущих блоков состоящих из $, A, C, G и T, т.к. по нему в первую очередь шла сортировка. Но куда интересней последний столбец **L** (на самом деле он и представляет из себя преобразование Барроуза-Уилера $BWT(S) = L$). В столбце **L** тоже встречаются подряд идущие символы, почему так происходит, если сортировка по нему шла в последнюю очередь (после сортировки по всем предыдущим символам)?

Отметим несколько наблюдений:

(i) символ L[i] *всегда идет перед* символом F[i] в исходной строке S, т.к. каждая строка это циклическая перестановка.

(ii) кажая строка матрицы M начинается с некоторого суффикса строки S, который заканчивается символом $, т.е. в некотором роде матрица – это набор всех суффиксов строки S (*аналогия с суффиксным деревом*).

(iii) если в строке S, есть повторяющиеся блоки (в нашем случае – это блоки TA), то строки матрицы, соответствующие суффиксам S, начинающимся с этих блоков идут подряд.

А что, если мы рассмотрим суффиксы (и соответствующие им строки в матрице), которые начинаются со вторых символов повторяющихся блоков? Они тоже должны оказаться рядом в матрице, но при этом первые символы блоков окажутся в последнем столбце матрицы (столбце **L**) по свойству (i), т.к. они идут перед вторыми символами, а эти символы в повторяющихся блоках одинаковы. Т.е. в **L** будут стоять подряд идущие одинаковые символы. Чем больше повторяющихся блоков, тем длиннее группы подряд идущих символов в **L**.

Из этих рассуждений мы и приходим к тому, что при наличии повторов в строке в последнем столбец **L** находятся группы одинаковых подряд идущих символов. А значит, мы можем «сжимать» столбец **L**, например, описанным ранее способом.

Итак, мы умеем сжимаем столбец **L**, но пока не знаем самого главного: можем ли мы по столбцу **L** воостановить исходную строку S? Другими словами, можем ли мы, зная только один столбец **L**, восстановить всю матрицу M? Оказывается, да. Сделаем еще одно наблюдение:

(iv) в каждом столбце матрицы M представлены все символы из строки S

Чтобы получить первый столбец, достаточно отсортировать все символы столбца **L**. Используя наблюдение (i), поставим перед столбцом **F** столбец **L** и отсортируем такие пары символов, тем самым получим первые два столбца матрицы M. Опять добавим перед осортированными парами столбец **L** и осортируем тройки символов, получим первые три столбца и т.д. В итоге мы получим всю матрицу M. Для получения исходной строки S, достаточно взять любую строку из матрицы и циклически сдвинуть ее так, чтобы $ стоял в конце. Готово.

```
L

A        $      A$        $T      A$T        $TA
G        A      GA        A$      GA$        A$T
T        A      TA        AG      TAG        AGA
T  sort  A  +L  TA  sort  AT  +L  TAT  sort  ATA  etc...
T   =>   A  =>  TA   =>   AT  =>  TAT   =>   ATA   =>
A        G      AG        GA      AGA        GA$
A        T      AT        TA      ATA        TAG
A        T      AT        TA      ATA        TAT
$        T      $T        TA      $TA        TAT
```

Отлично, мы показали возможность восстановления исходной строки из сжатой, хотя и не самым оптимальным способом. Стоит отметить, что этот подход используется, как одна из стадий во многих архиваторах (например, bzip2). Осталось продемонстрировать, как осуществлять поиск мотива в такой строке **L** (за линейное время) и задача оптимизации памяти будет решена.

Опять же отметим, что результатом преобразования Барроуза-Уилера является столбец **L**. Т.е. $BWT(S) = L$.

4 Индексы Ферраджина-Манзини

В конце XX века Ферраджина и Манзини предложили две функции, названные FM-индексом и позволяющие решить задачу восстановления исходной строки по **L** и поиска мотива в S за линейное время. Опишем эти две функции.

I. $C(x)$ – возвращает кол-во символов в строке S, лексиграфически меньших, чем x.

II. $Occ(x, t)$ – возвращает кол-во символов x в префиксе строки **L** длиной t (т.е. в подстроке L[1..t]).

Возвращаемые значения этих функций, удобно представить в виде таблиц. Запишем их для строки S = TATATAGA$, для которой L = BWT(S) = AGTTTAAA$.

Таблица 1: $C(x)$

x	$	A	C	G	T
Occ(x)	0	1	5	5	6

Таблица 2: $Occ(x, t)$

t	1	2	3	4	5	6	7	8	9
L[t]	A	G	T	T	T	A	A	A	$
$	0	0	0	0	0	0	0	0	1
A	1	1	1	1	1	2	3	4	4
C	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1
T	0	0	1	2	3	3	3	3	3

Что позволяют делать эти две функции? Пусть i-ый символ в столбце **L** это j-ый символ в строке S (L[i] = S[j]), с помощью функций $C(X)$ и $Occ(x)$ мы можем определить позицию этого же символа (S[j]) в столбце **F**.

Прежде, чем показать, как это сделать, докажем один интересный факт. Проставим каждому символу в **F** ранг, другими словами его порядковый номер среди всех идентичных ему символов, двигаясь сверху-вниз (красные цифры на Рис. 1). Тем самым мы каждому символу присвоим уникальный номер среди всех ему подобных. Отметим эти же номера в последнем столбце **L**, здесь подразумевается не новое проставление рангов, а сохранение рангов из первого столбца, например, A_1 из **F** это восьмой символ в строке S, этот же символ находится первым в **L**, значит его ранг тоже A_1. Так вот: *последовательность рангов в последнем столбце не будет нарушена.* Т.е. второй сверху нуклеотид «A» в первом столбце также будет вторым сверху в последнем.

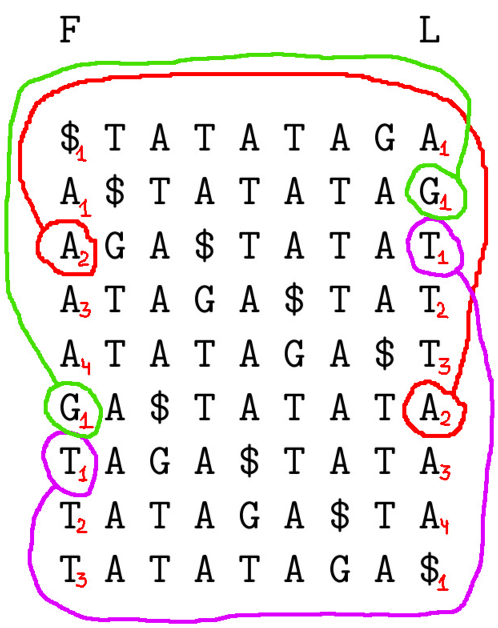

Рис. 1: Пример сохранения рангов в **F** и **L**.

Откуда это следует? Рассмотрим, например, все строки в M, начинающиеся на T (в **F** стоит T), пусть они составляют множество Ψ, очевидно, что они идут подряд и отсортированы. Строки, которые заканчиваются на T (у которых в **L** стоит T), образованы из всех строк Ψ циклическим сдвигом на одну позицию «влево». Причем, находятся они в матрице в отсортированном порядке, начиная с их первого символа (второго относительно Ψ). Но в множестве Ψ они были точно в таком же порядке, т.к. первый символ у всех строк из Ψ одинаковый (равен T), значит они сортировались по второму, третьему и т.д. символам.

Что это дает? Зная ранг i-го символа в последнем столбце $rank(x)$, мы легко можем определить его позицию в первом столбце. Этот переход задается функцией «last-to-first»:

$$LF(i) = C(L[i]) + rank(L[i])$$

Заметим, что $rank(L[i]) = Occ(L[i], i)$, т.к. как ранг – это и есть количество вхождений символа в префикс, тогда:

$$LF(i) = C(L[i]) + Occ(L[i], i)$$

Для i-го символа в столбце **F** мы можем определить символ, который идет перед ним в строке S, просто посмотрев в этой же строке символ L[i]. Для L[i] с помощью функции $LF(i)$ мы можем определить строку, в которой L[i] стоит в первым в строке, пусть это строка $j = LF(i)$. Теперь мы можем узнать, что идет в S перед этим символом опять же посмотрев последний символ в j, т.е. L[j] и т.д. Т.е. такими циклическими переходами мы можем восстановить всю строку.

5 Поиск мотива

Эта же логика применяется при поиске мотива в строке. Допустим, мы хотим определить кол-во вхождений мотива TAT в строку S. Будем двигаться *от последнего символа мотива к первому*, начинаем с символа T, находим строки, которые начинаются с T, это строки с индексами $\{8, 9, 10\}$ (мы легко это можем сделать с помощью $C(x)$, Рис. 2). Следующий символ с конца в мотиве – A, в отобранных нами строках перед символами T, идут символы, которые находятся в последнем столбце в $\{8, 9, 10\}$ строках, выберем из них те, которые оканчиваются на A, это строки $\{8, 9\}$ с A_3 и A_4 в конце. Найдем строки, которые начинаются с A_3 и A_4 – это $\{4, 5\}$ строки (этот переход мы легко сможем совершить с помощью функции $LF(i)$). Берем следующий символ мотива с конца – T. Находим в отобранных нами строках, заканчивающиеся на T, они обе подходят и заканчиваются на T_2 и T_3, перейдем к строкам (с помощью $LF(i)$), начинающимися на эти T – это строки $\{8, 9\}$. Собственно все, мы прошли по всем символам мотива и определили строки, а значит суффиксы и кол-во вхождений мотива в геном S.

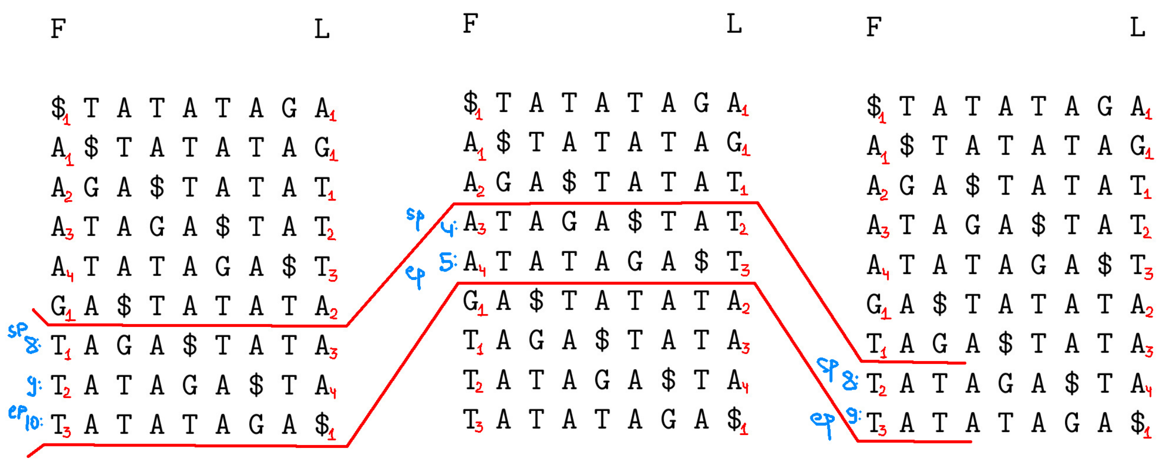

Рис. 2: Поиск мотива TAT с помощью преобразования Бароуза-Уилера и индексов Ферраджина-Манзини.

Формализуем алгоритм:

procedure BWMATCH(Pattern[1, p])
  $c = P[p], i = p$
  ▷ $(c+1)$ – следующий символ алфавита после c
  $sp = C(c) + 1, ep = C(c+1)$

  while $sp \le ep$ *and* $i \ge 2$ **do**
    $c = P[i-1];$
    $sp = C(c) + Occ(c, sp-1) + 1;$
    $ep = C(c) + Occ(c, ep);$
    $i = i - 1;$
  end while

  if $ep < sp$ **then**
    return 0
  else
    return $ep - sp + 1$
  end if
end procedure

6 Как хранить $C(x)$ и $Occ(x, t)$?

Очевидно, что для хрнения функции-таблицы $C(x)$ нам необходимо $O(\sigma)$, где σ – размер алфавита, т.е. пренебрежительно малая величина. Но для $Occ(x, t)$, нам потребуется $O(\sigma n)$ памяти, что даже при нуклеотидной последовательности достигает $5n$ и сравнимо с суффиксным деревом. Т.е. весь выигрыш от сжатия самого генома теряется. На самом деле в настоящих реализациях индексов все устроено немного иначе, но подробное объяснение займет слишком много времени. Ограничимся тем фактом, что мы можем хранить значения $Occ(x, t)$ не для всех префиксов $t \in 1..n$, а, скажем, для каждого *пятого* значения t, тем самым уменьшив в пять раз размер необходимой памяти. Что будет происходить при поиске мотива? Если мы попадаем на строку, для которой не указано $Occ(x, t)$, нам необходимо «спуститься» или «подняться» до блийжайшей строки в матрице M, для которой это значение посчитано, при этом скорректировав новое $Occ(x, t)$. Тем самым мы увеличим время поиска мотива, но существенно уменьшим объем необходимой оперативной памяти для хранения $Occ(x, t)$, а, как мы убедились, в случаях с большими геномами, важна даже константа в $O(n)$. С настоящим решением этой проблемы можно ознакомиться в оригинальной статье про FM-индексы[2].

7 Ссылки

[1] Burrows M., Wheeler D. A block-sorting lossless data compression algorithm //DIGITAL SRC RESEARCH REPORT. – 1994.

[2] Ferragina P., Manzini G. Opportunistic data structures with applications //Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. – IEEE, 2000. – С. 390-398.