Read Through Transcription
the official Avadis NGS blog

Elegant exact string match using BWT

April 23, 2012 by Santhosh Kumar

> *This post is the first of a series where we describe some interesting algorithms used in our product Avadis NGS. The first post in this series is about string matching using BWT, a technique that forms the core of our aligner COBWeb. We have intentionally kept the usage of bioinformatics jargon minimum to benefit a wider audience.*

This post describes an elegant and fast algorithm to perform exact string match. Why another string matching algorithm? To answer the question, let's first understand the problem we are trying to solve.

In short, the problem is to match billions of short strings (about 50-100 characters long) to a text which is 3 billion characters long. The 3 billion character string (also called reference) is known ahead and is fixed (at least for a species). The shorter strings (also called reads) are generated as a result of an experiment. The problem arises due to the way the sequencing technology works, which in its current form, breaks the DNA into small fragments and 'reads' them. The information about where the fragments came from is lost and hence the need to 'map' them back to the reference sequence.

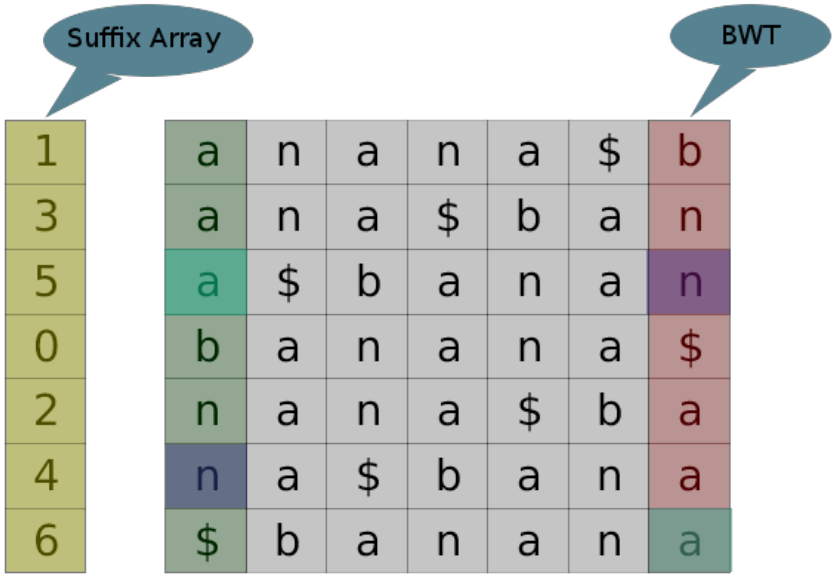
We need an algorithm that allows repeatedly searching on a text as *fast* as possible. We are allowed to perform some preprocessing on the text once if that will help us achieve this goal. BWT search is one such algorithm. It requires a one-time preprocessing of the reference to build an index, after which the query time is of the order of the length of the query (instead of the reference).

Burrows Wheeler transform is a reversible string transformation that has been widely used in data compression. However the application of BWT to perform string matching was discovered fairly recently in this paper. This technique is the topic of this post. Before we get to the searching application, a little background on how BWT is constructed and some properties of BWT.

BWT for a given text T is constructed as follows:

- Append an end of string character (say '$') to the text. This character should not be present in the text and is treated as the lexicographically largest character.
- Collect all the cyclic permutations of the text and sort them.
- Arrange all the sorted strings one below the other. The last column forms the BWT.

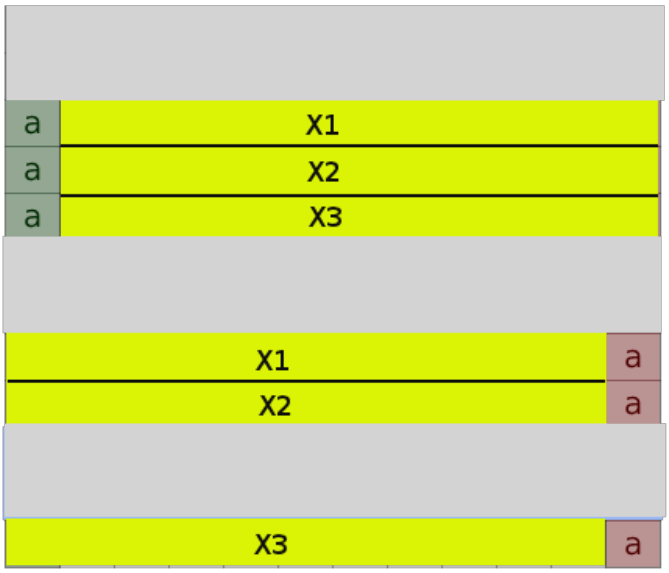
The image below shows the BWT transform for banana$.



A few things to note from the image

- The red column, as indicated, is the BWT of BANANA$'
- The yellow column, as indicated, is the Suffix Array. For every row, the element in the suffix array is the index into the original text of the suffix starting in that row. Note that if you have the suffix array, constructing BWT is straightforward. The above algorithm of performing a full sort on all suffixes to find the suffix array and BWT is very inefficient. Efficiently finding the suffix array will be the topic for our next post in this series.
- The green column is just the sorted characters in the text.

Before proceeding we need to understand one interesting property of the BWT. The rank of a character in the BWT column is the same as that of the corresponding character in the first column. This is illustrated in the image above using colors. The 2nd N in the BWT column correponds to the 2nd N in the first column, the 3rd A in the BWT column corresponds to the 3rd A in the first column and so on. To see why this is the case, remember that all the rows in the grid are cyclic permutations of a single string. Let AX1, AX2, AX3 be three consecutive rows in the grid where A is the first character and X1, X2, X3 denote the rest of the row. As the rows are sorted lexicographically, X1, X2, X3 are sorted too. On cyclic shifts, we get X1A, X2A, X3A which are again sorted, hence in the same relative order in the grid. Note that now the A's are entries in the BWT *in the same relative order*. This is illustrated in the figure below.



Search using BWT

Our index data structure has the following

1. BWT – Indexed access. We will denote this by BWT(i).
2. Suffix Array – Indexed access. We will denote this by SA(i).
3. First column – Queries : Start index, end index of an alphabet and index given alphabet and a rank. For example, find the index of letter 'G' with rank 3. This can be achieved by just storing the frequency of each alphabet in order. For example, for BANANA$, its enough to store 3A1B2N1$. This query can be answered by a simple walk over this compressed structure. We will denote the queries by Start(alphabet), End(alphabet) and IndexOf(alphabet, rank).
4. Ranks in BWT – We need fast access to the following query : Get number of occurences of a particular character above a particular row in the BWT. For example, get the number of 'A's above row 5. One way to achieve this is to store at each row the frequency of each alphabet till that row. This can be achieved by a single pass over the BWT. In practice though, we don't store this at every row, but store at regular intervals. We will assume here that it is stored at every row for simplicity. We will denote this query by RankOf(index, alphabet)

Let T be the text searched on and Q be the query string. The algorithm is as follows

```
index = len(Q) # Walk the query backwords
nextChar = Q[index--]
(start, end) = (Start(nextChar), End(nextChar))
while index >= 0:
  nextChar = Q[index--]
  (startrank, endrank) = (RankOf(start, nextChar), RankOf(end, next
  (start, end) = (IndexOf(nextChar, startrank), IndexOf(nextChar, e
for i in range(start, end):
  print SA(i)
```

The output of this algorithm are the indices into the original text where the query is present. Note that error condition handling has been skipped in the pseudocode above for brevity. It involves checking if the range found at every iteration of the loop is a valid range.

The visualization below explains the algorithm. Few points about the visualization

- Yellow text at the top indicates the text to search on and the yellow text at the bottom,

the query

- Red column indicates BWT, orange the suffix array
- The grey part of the grid is not stored as part of the index. It is shown here just for clarity
- Rows highlighted in blue indicates the range where search till that point is successful. This appears once the search animation starts.

Let's step through the visualization for text ABRACADABRA and query DAB. Follow the points below by hitting on 'Step' button after each point.

- Remember that the query is walked from the last to first. The first character is 'B', which results in the range shown
- Next character is 'A'. Get the rank of 'A' on the BWT for the first and last rows on the current range. The ranks as shown in the tooltips are 0 and 2. The new range is determined by the rows where 'A' has rank 0 and rank 2 on the first column. This is shown by the new range
- Now the final character 'D'. Performing the the same steps as above, we get the final range as the row '9'. The corresponding element on the suffix array is '6' which is the index in the original text where 'DAB' starts

Play around with the visualization with different texts and queries to understand the algorithm better.

a	b	r	a	c	a	d	a	b	r	a	$

0	a	b	r	a	c	a	d	a	b	r	a	$
7	a	b	r	a	$	a	b	r	a	c	a	d
3	a	c	a	d	a	b	r	a	$	a	b	r
5	a	d	a	b	r	a	$	a	b	r	a	c
10	a	$	a	b	r	a	c	a	d	a	b	r
1	b	r	a	c	a	d	a	b	r	a	$	a
8	b	r	a	$	a	b	r	a	c	a	d	a
4	c	a	d	a	b	r	a	$	a	b	r	a
6	d	a	b	r	a	$	a	b	r	a	c	a
2	r	a	c	a	d	a	b	r	a	$	a	b
9	r	a	$	a	b	r	a	c	a	d	a	b
11	$	a	b	r	a	c	a	d	a	b	r	a

d	a	b

Play Step

Text: abracadabra
Query: dab
Load

How does this search work? Let's first understand what the loop invariant here is. Let 'i' be the index into the query string at some point of time when the range in the grid is (s,e). Remember that we are walking the search query backwards. The loop invariant is that Q[i:] is the prefix of all strings in the range (s,e). To prove that that is indeed the case, let's move one more character on the query. Consider the characters on the BWT for the indices between s and e. All these characters precede various occurences of the prefix Q[i:] in the text. The rank data structure enables us to quickly check if any of those characters are the character we want to match. If yes we get the start and end rank of the character. Note that if the character is not present in that range, the start and end rank of the character would be same, ending our search. Then we use the property of BWT described above to jump to the corresponding character on the first column and continue our search. Thus when we exhaust all the characters in the query, we are left with the range on the grid where the prefix of every row is the query.

On the next post in this series, we will describe how to find the BWT index of a very long string efficiently.

About **SANTHOSH KUMAR**

Santhosh is a software engineer at Strand. He is one of the engineers behind Avadis NGS. He did his BTech and MTech in Computer Science from IIT Madras and interned at Google before joining Strand.

Comments (6)

Pingback: Cool Algorithm – Fast text search using BWT « Tyson Zinn « Tyson Zinn

Pingback: Robert McGhee » April 24th

Pingback: High Speed Searching Possible with Jazzy Algorithm : Beyond Search

Pingback: Elegant exact string match using BWT « Another Word For It

Pingback: Hemprasad Badgujar

Pingback: URL

← Previous Post Next Post →