Burrows-wheeler transform

SATHVIK TUMOJU

(180001060)

Y.SAI ARAVIND

(180001063)

CONTENTS:

INTRODUCTION:

The Burrows-Wheeler Transform (BWT) is a way of permuting the characters of a string S into another string BWT(S). This permutation is reversible; one procedure exists for turning S into BWT(S) and another exists for turning BWT(S) back into S. The transformation was originally discovered by David Wheeler in 1983, and was published by Michael Burrows and David Wheeler in 1994.

Burrows-Wheeler transform (BWT) is an algorithm maximizing repeated letters in a text, which is useful in data compression. The transform is done by sorting all the circular shifts of a text in lexicographic order (dictionary order) and by extracting the last column and the index of the original string in the set of sorted permutations of string.

For example:

if input text = “banana$”.

Output of BWT = “annb$aa”.

You can notice some repetition in burrows wheeler transform of input text.

BURROWS-WHEELER TRANSFORM IN COMPRESSION:

How is the Burrows-Wheeler Transform useful for compression? First, it's reversible. Transformations used in compression must be reversible to allow both compression and decompression. Second, characters with similar right-contexts in S tend to come together in BWT(S). This can, for instance, bring several occurrences of the same character together in a tight bunch. This is hard to see in small examples; in the following example, this bunching is more obvious:

Input = "tomorrow and tomorrow and tomorrow".

Output = " wwdd nnoooaatttmmmrrrrrrooo wooo".

This makes BWT(S) more compressible. For example, we could take BWT(S) and shrink it (reversibly) with run-length encoding (RLE). Software tools for compression and decompression employ various methods to shrink BWT(S), including move-to-front transforms, run-length encoding, Huffman coding, and arithmetic coding. The popular bzip2 compression tool uses these and other methods.

In biological sciences there are many long strings which cannot be compressed easily. BWT of such long strings can be compressed easily so that they can be transferred with less errors.

For example:

Input = “SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES”.

Output = “TEXYDST.E.IXIXIXXSSMPPS.B..E.S.EUSFXDIIOIIIT”.

The output is easier to compress because it has many repeated characters. In this example the transformed string contains six runs of identical characters: XX, SS, PP, .., II, and III, which together make 13 out of the 44 characters.

It is also used for text data compression. Lets see with some commonly used texts

For example:

Input = “iloveyou”.

Output = “vuiylooe”.

You can notice some repetition.

IMPLEMENTATION:

Following are the steps involved in bwt algorithm

Let , input string text =“ decode”.

Step1:

All cyclic rotations

decode

edecod

dedeco

odedec

codede

ecoded

Step 2:

sort in lexical order

codede

decode

dedeco

ecoded

edecod

odedec

Step 3:

Take the last column

codede

decode

dedeco

ecoded

edecod

odedec

output string = “eeoddc” .

BWT(decode) = “eeoddc”.

If input string is “abraca$”.

Step1:

abraca$

$abraca

a$abrac

ca$abra

aca$abr

raca$ab

braca$a

Step2 :

$abraca

a$abrac

abraca$

aca$abr

braca$a

ca$abra

raca$ab

Step3:

$abraca

a$abrac

abraca$

aca$abr

braca$a

ca$abra

raca$ab

output =“ac$raab”.

- The ‘$’ sign is viewed as first letter lexicographically, even before ‘a’.

BWT(abraca$) = “ac$raab”.

NOTE : BWT string text can be converted back to original string text using inverse burrows wheeler transform.

- The major reasons for considering last column as BWT is for its clustering nature and if we only have BWT of our string, we can recover the rest of the cyclic rotations entirely. The rest of the columns don’t possess this characteristic which is highly important while computing the inverse of BWT

WHAT MAKES IT USEFUL:

1) After BWT is applied to a string, the transformed string tends to have repetitive characters appearing close together, making the string more easier to compress.
2) It doesn't add any extra data, just gives a permutation of original string
3) Above two things can be perfectly achieved even by just sorting the string (Ex: 'chaitanya' can be transformed to 'aaachinty'), but the real strength of BWT lies in the fact that it's reversible. By sorting you might get all repetitive characters appear together, but you can't get your data back unless you had some extra information about the original string.

DATA COMPRESSION BASED ON BWT:

BWT itself doesn't compress the data, so compression based on BWT combined the BWT with currently compression techniques.

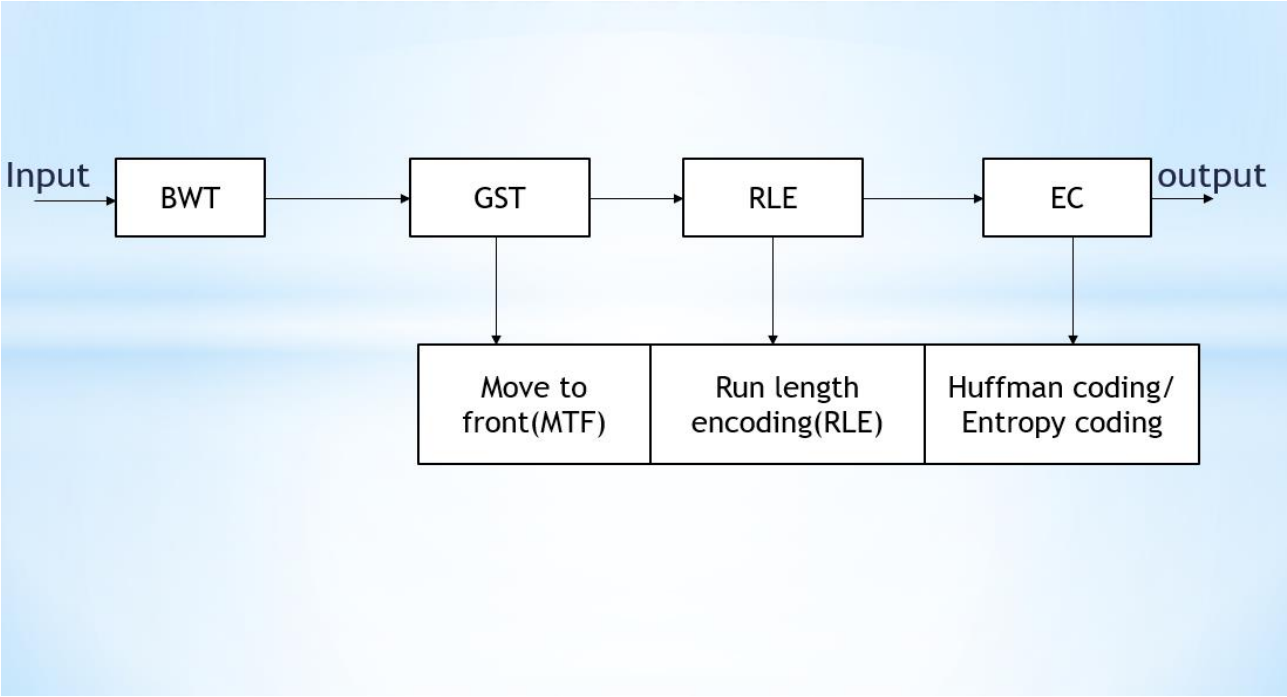

Fig. Data Compression

PSEUDO CODE:

```
function BWT (string s)
    make a table, with rotated rows;
    sort rows alphabetically;
    return (last column of the table);
```

COMPLEXITY OF ALGORITHM:

The major part in this algorithm is sorting. We use quicksort algorithm whose time complexity is O(nLogn) in sorting and we recurre this quicksort algorithm for O(n) times. So, the time complexity of our algorithm is **O(n^2 Logn).**

But, our algorithm is not optimized .We can optimize our algorithm by using different sorting methods.

With dictionaries that do not repeat many value the complexity would be O(nLogn) . Because, it doesn't need recurrence of quicksort.

OPTIMIZATION:

We can optimize our algorithm and reduce the time complexity to O(n*Logn*Logn).

The idea is to use the fact that strings that are to be sorted are suffixes of a single string.
We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than 2n. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in O(nLogn) time using a nLogn sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in O(1) time (we need to compare only two values, see the below example and code).
The sort function is called O(Logn) times (Note that we increase number of characters to be considered in powers of 2).

Therefore overall time complexity becomes **O(nLognLogn).**

PSEUDO CODE:

function BWT (string s)

make a table, with rotated rows;

sort rows according to 2^0 , 2^1 , 2^2 ,…. 2^i characters alphabetically;

return (last column of the table);

TEST CASES:

Let’s take S= “mississippi” as input and test the run time.

```
babyyy@sathvik:~$ cd Desktop
babyyy@sathvik:~/Desktop$ g++ bwtAlgo.cpp && ./a.out
 input text : mississippi
Burrows - Wheeler Transform : pssmipissii
Time taken by program is : 0.000062 sec
babyyy@sathvik:~/Desktop$ g++ optimBwtAlgo.cpp && ./a.out
Following is suffix array for mississippi
10 7 4 1 0 9 8 6 3 5 2
Time taken by program is : 0.000049 sec
babyyy@sathvik:~/Desktop$
```

Time taken by intial algorithm = 0.000062 sec.

Time taken by optimized algorithm = 0.000049 sec.

Clearly, optimized algorithm takes less run time compared to intial algorithm.

ALGORITHMIC ISSUES:

Most BWT-based compressors process the input file in blocks. A single block is read, compressed and written to the output file before the next one is considered. This technique provides a simple means for controlling the memory requirements of the algorithm and a limited capability of error recovering. As a general rule, the larger is the block size the slower is the algorithm and the better is the compression. In bzip2 the block size can be chosen by the user in the range from 100Kb to 900Kb.

The most time consuming step of BWT-based algorithms is the computation of the transformed string bwt(s). In Sect. 2 we defined bwt(s) to be the string obtained by lexicographic sorting the prefixes of s. This view has been adopted in some implementations, whereas in other cases bwt(s) is defined considering the lexicographic sorting of the suffixes of s. This difference does not affect significantly neither the running time nor the final compression. From an algorithmic point of view the problems of sorting suffixes or prefixes are equivalent.

CONCLUSIONS:

We have presented with the burrows wheeler transform that is used for lossless data compression. We looked at the structure and various stages in compression and in decompression stages run in reverse order compared to compression. Burrows wheeler transform is good for compression of text based files. However, is worse for non-text based files compared to other compression methods. The biggest drawback of BWT-based algorithms is that they are not on-line, that is, they must process a large portion of the input before a single output bit can be produced.

REFERENCES:

A. Andersson and S. Nilsson. Efficient implementation of suffix trees. Software —Practice and Experience, 25(2):129–141, 1995.

("Burrows-Wheeler transform", n.d.,main page).

Anureet Kaur, Burrows-Wheeler transform algorithm [https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm].

B.S. Shajeemohan , V.K. Govindan, 15 August 2005, International Conference on Mobile Business (ICMB'05)[https://ieeexplore.ieee.org/].