League Table Weights, LTW for BWT

by

Abderrahim Hechachena

27/05/2018

abderrahimhechachena@yahoo.co.uk

BWT, Theory and Practice

In this paper we will expose two diffent applications to solving BWT, without using any sorting algorithms. The first one is purely linear algebra problem that may help shed the light on this type of matrices and their composition. The second application is a fast algorithm that may be developed further for practical uses, such as data compression and search engines.

Linear algebra solution to BWT; output = W.v

League Table Weights is an algorithm that solves the Borrow Wheeler Transform without using the traditional sorting algorithms. This weighting algorithm builds a weight matrix W that reflects the power of each character in the input string. Given a string s = 'ueyhfghrygfueyhfghr..' of length n, add the character '!' at the end of the line to get 'ueyhfghrygfueyhfghr..!'. Now build a weight matrix W[n+1, n+1] with column names equal each character 'ueyhfghrygfueyhfghr..!' and row names equal to each character 'ueyhfghrygfueyhfghr..!'. To fill each cell do the fallowing. Set a flag to the value of 1 if column name > row name; example r > g, set flag to -1 if column name is less than row name; example c < d, put the value of the flag

in the cell where the target column and the target row meet. The cell with 0 value need to take their values form the bottom right cell. Example: if cell [3,9] which contain [y,y] is zero then cell [4,10] which is [h,g] will provide the value -1 as g is less than h. Now we have a Matrix W of weights. We need to sum the values of each column to get the weights of each character. This is a linear solution to the BWT. The way zero cells are filled will be obvious later when we see an example of comparing numbers.

League Table Weights, LTW R Example

For now lets see how it is working using R code:

```
> # league table  to generate a matrix W of weights
> generateWeightMatrix = function(inputStr1,zero = 1){
+   inputStr1 = paste(inputStr1,'!',sep="")
+   inputStr = paste(inputStr1,inputStr1,sep="")
+   strLen = nchar(inputStr1)
+   strLen1 = strLen - 1
+   W = matrix(0,strLen,strLen)
+   arr = strsplit(inputStr, "")[[1]]
+   for(i in 1:strLen-1){W[i,strLen] = -1;W[strLen,i] = 1}
+   #################
+   for(j in strLen1:1)
+   {
+     for (i in 1:strLen1)
+       {
+       aj = arr[j]
+       ai = arr[i]
+       if(aj == ai){if(zero == 1){W[i,j] = W[i+1,j + 1]}}else{
+         if(aj>ai){flag = 1}else{flag = -1}
+         W[i,j] = flag
+       }
+     }
+   }
+   #################
+   diag(W) = 0
+   #W[lower.tri(W)] <- 0#;W = W -t(W)
+   W = as.data.frame(W)
+   rownames(W) = paste(1:strLen,arr[1:strLen],sep = '')
+   names(W) = arr[1:strLen] #paste(1:strLen,arr[1:strLen],sep = '')
+   W
+ }
```

Using the above R function to produce the weights matrix W, then producing v as a vector of ones and calculating the characters weights and producing the final transformed string.

```
> # generate string of length n
> n = 6
> s = paste(sample(letters, n, replace = T),collapse = "")
> for(i in 1:1){s = paste(s,s,sep="")}
> s

[1] "bxyhvnbxyhvn"

> n = nchar(s)
> v = rep(1,n+1)
> v

 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1

> # generate all weights
> d = generateWeightMatrix(s,1)
> N = as.matrix(d)

> N

     b  x y  h  v  n  b  x  y  h  v  n  !
1b   0  1 1  1  1  1 -1  1  1  1  1  1 -1
2x  -1  0 1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1
3y  -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
4h  -1  1 1  0  1  1 -1  1  1 -1  1  1 -1
5v  -1  1 1 -1  0 -1 -1  1  1 -1 -1 -1 -1
6n  -1  1 1 -1  1  0 -1  1  1 -1  1 -1 -1
7b   1  1 1  1  1  1  0  1  1  1  1  1 -1
8x  -1  1 1 -1 -1 -1 -1  0  1 -1 -1 -1 -1
9y  -1 -1 1 -1 -1 -1 -1 -1  0 -1 -1 -1 -1
10h -1  1 1  1  1  1 -1  1  1  0  1  1 -1
11v -1  1 1 -1  1 -1 -1  1  1 -1  0 -1 -1
12n -1  1 1 -1  1  1 -1  1  1 -1  1  0 -1
13!  1  1 1  1  1  1  1  1  1  1  1  1  0
```

N is asymmetric, with the upper triangular is the negative of the lower triangular transpose, weights are 1 for column character > row character -1 for the opposite. The equal characters values are borrowed from the right bottom cell just next to it. All diagonal elements are zeroes.

```
> det(N)

[1] 0
```

N could be full rank with det = 1 or reduced rank with det = 0, if the input string is a repeat of the same patterns. BWT process is reversible with det = 1 or det = 0.

```
> w = t(N)%*%v
> # the weights to be used for sorting the BWT array are:
> t(w)

      b x  y  h v n   b x  y  h v  n   !
[1,] -8 8 12 -4 4 0 -10 6 10 -6 2 -2 -12

> S = strsplit(paste('!',s,sep = ""),"")
> d1 = data.frame(unlist(S),as.numeric(w))
> names(d1) = c('S', 'w')
> out = as.character(as.factor(d1[order(w),][,1]))
> # producing the final transformed string.
> out

 [1] "n" "n" "!" "y" "y" "v" "v" "h" "h" "b" "b" "x" "x"
```

Now we have introduced the main features of LTW, we have to look at it in more details in order to have a successful implementation.

- First remark is: We can produce the upper triangular from the lower triangular by changing the weights signs.
- Second: We can calculate most of the values just by comparing characters.
- The problem though, is where cells are at cross between two similar values example [a, a] we don't know what weights to give them. unless we know the value at the right bottom cell. This is the only complications here. The algorithm I am going to present here, is light weigh in code and stored objects. It has the bare bone elements.

with the second parameter set to zero we can produce the simple weights matrix M using 'generateWeightMatrix'. The main target of this algrithm is to fill the non diagonal zero cells in this matrix.

```
> d = generateWeightMatrix(s,0)
> M = as.matrix(d)
```

The weights of similar characters have the value zero In M and they need to be filled with the values at the right bottom of each cell as implemented in the next C sharp function.

```
> M
```

```
     b  x y  h  v  n  b  x y  h  v  n  !
1b   0  1 1  1  1  1  0  1 1  1  1  1 -1
2x  -1  0 1 -1 -1 -1 -1  0 1 -1 -1 -1 -1
3y  -1 -1 0 -1 -1 -1 -1 -1 0 -1 -1 -1 -1
4h  -1  1 1  0  1  1 -1  1 1  0  1  1 -1
5v  -1  1 1 -1  0 -1 -1  1 1 -1  0 -1 -1
6n  -1  1 1 -1  1  0 -1  1 1 -1  1  0 -1
7b   0  1 1  1  1  1  0  1 1  1  1  1 -1
8x  -1  0 1 -1 -1 -1 -1  0 1 -1 -1 -1 -1
9y  -1 -1 0 -1 -1 -1 -1 -1 0 -1 -1 -1 -1
10h -1  1 1  0  1  1 -1  1 1  0  1  1 -1
11v -1  1 1 -1  0 -1 -1  1 1 -1  0 -1 -1
12n -1  1 1 -1  1  0 -1  1 1 -1  1  0 -1
13!  1  1 1  1  1  1  1  1 1  1  1  1  0
```

The next function is a non optimal implementation of LTW. provided for testing

```
> leagueTableBwtEncode = function(inputStr1){
+   inputStr1 = paste(inputStr1,'!',sep="")
+   inputStr = paste(inputStr1,inputStr1,sep="")
+   strLen = nchar(inputStr)/2
+   B = rep(0,strLen)
+   inputStrA = strsplit(inputStr, "")[[1]]
+   #################
+   flag = 0
+   # gap loop
+   for(s in 1:(strLen)){
+     for(i in (strLen):1){
+       ai = inputStrA[i]
+       as = inputStrA[i + s]
+       if(ai == as){}else{
+         if(ai>as){flag = 1}else{flag = -1}}
+       B[i] = B[i] + flag
+     }
+   }
+   #################
+   # shape output
+   outputStr = strsplit(inputStr,'')
+   index = 1:strLen
+   d = as.data.frame(cbind(B,index))
+   d1 = d[order(B),]
+   output = ''
+   for(i in 1:strLen){
+     output = paste(output,outputStr[[1]][d1$index[i]+strLen-1],sep = '')
```

```
+   }
+   output
+ }
> #####
> output = leagueTableBwtEncode(s)
> output
```

```
[1] "nn!yyvvhhbbxx"
```

Then decode it to get the original string back. The decoder is an implementation of a copy found in github. It is not my work, although I coded it in R.

```
> bwt_decode = function(strEncoded)
+ {
+   ############### transform to array of bite code
+   #strEncoded = strsplit(strEncoded,'')
+   strEncoded = utf8ToInt(strEncoded)
+   size = length(strEncoded)
+   ###############
+   F = rep(0,size)
+   strDecoded = rep(0, size)
+   buckets = rep(0,256)
+   indices = rep(0,size)
+   ###############
+   ### get frequencies of ascii codes
+   for (i in 1:(size)){buckets[strEncoded[i]] = buckets[strEncoded[i]] + 1}
+   ### sort input array
+   F = sort(strEncoded)
+   ### accumulated frequencies
+   j = 1
+   for (i in 1:256)
+   {
+     while (i > F[j] & j < size)
+     {
+       j = j + 1
+     }
+     buckets[i] = j-1
+   }
+   ###
+   for (i in 1:size){
+     buckets[strEncoded[i]] = buckets[strEncoded[i]] + 1
+     indices[buckets[strEncoded[i]]] = i
+   }
+   j = 1
+   for (i in 1:size)
+   {
```

```
+     strDecoded[i] = strEncoded[j];
+     j = indices[j];
+   }
+   g = grep('33', strDecoded)
+   strDecoded1 = intToUtf8(strDecoded)
+   strDecoded2 = paste(substr(strDecoded1,g+1,size),substr(strDecoded1,1,g-1),sep = '')
+   strDecoded2
+ }
> #####
> bwt_decode(output)

[1] "bxyhvnbxyhvn"

> ###############################################################################
> # vectorized league table method 'R friedly!' another BWt encoder
> rotateArray = function(x){c(x[-1],x[1])}
> ### use R vectorization power to save time
> getWeithts = function(a,b){
+   a1 = as.numeric(a>b)
+   b1 = -as.numeric(b>a)
+   c = a1 + b1
+   # get runs
+   y = rle(c)
+   inPosition = grep(0,y$values)
+   fromPosition = inPosition + 1
+   y$values[inPosition] = y$values[fromPosition]
+   b = rep(y$values,y$lengths)
+   b
+ }
> ######
> VleagueTableBwtEncode = function(inputStr1){
+   inputStr = paste(inputStr1,'!',sep="")
+   strLen = nchar(inputStr)
+   W = rep(0,strLen)
+   inputStrA = strsplit(inputStr, "")[[1]]
+   #################
+   secondArr = rotateArray(inputStrA)
+   W = getWeithts(inputStrA, secondArr)
+   # gap loop
+   for(s in 2:(strLen-1)){
+     secondArr = rotateArray(secondArr)
+     W = W + getWeithts(inputStrA, secondArr)
+   }
+   #### shape output
+   inputStrAA = c(inputStrA,inputStrA)
+   index = 1:strLen
```

```
+   d = as.data.frame(cbind(W,index))
+   d1 = d[order(W),]
+   output = ''
+   for(i in 1:strLen){
+     output = paste(output,inputStrAA[d1$index[i]+strLen-1],sep = '')
+   }
+   output
+ }
```

League Table Weights, LTW C Sharp Example

from the previous introduction it is clear now that C sharp implementation is going to hover over the upper triangular matrix, skipping all non equal values and writing to weight matrix only when a match is found.

First lets look at how LTW works: to compare two numbers usually we start from the left till you find a missmatch, example: 35268... 35263... the first number is larger at the fifth position. In the case of BWT we are comparing more than one number at a time, so every number in the top set is larger than the number in the bottom set. I this case we moved right till we found the postion 5 where $8 > 3$, the we moved left to declare every top number as a winner. We could just started from the right and assigned the value of 8>3 the the rest till we find a new missmatch. To do this in practice lets imagine we have a string s of n characters 'jshfuheygftgd...'. lets imagine this string as a round ring on a horizontal plane, a round table. Put on top of this ring another similar ring with the same number of characters in the same positions. On top of both rings we need a third ring with the same size but contains zeroes only. Both top rings are fixed, the table is fixed as well, the bottom ring is movable. Now imagine that you are sitting on a moving char, clock wise around this table.

Here is the process, each step is one character:

- you have three rings and a flag that have value 1 or -1.
- 1) Move the bottom ring clock wise by one character.
- 2) if the second ring character in that position is bigger than the bottom ring set flag to 1
- 3) if the second ring character in that position is smaller than the bottom ring set flag to -1
- 4) if they are equal keep the flag as it is.
- 5) increment the top ring cell in the same position by the value of the flag.
- 6) move the chair by one character clock wise.

- 7) repeat steps 2 to 6 till end of cycle
- repeat steps 1 to 7 till end of string

This process is enough to build the the weights vecor W.v, It is not optimal for implementation, as more savings could be implemented for faster algorithms, as we are going to see in C sharp implementation.

- Given a string s of length n, to be processed using BWT encoder, generate a weights matrix W using 'generateWeightMatrix' function with inputStr1 = s and the second option set to 1.
- Generate the vector v using rep(1,n+1) 'The reason why n + 1 not n is the fact that we have added a special character '!' at the end of the inputStr1'.
- calculate W*v and sort '!' + s by the weights W*v, to produce the final processed string.

The implementation of this method does not need to generate the matrix W, as this is very hard on computer memory. The implementation is an algorithm that needs only the inputs string twice and an array the size of the input arrays to store the weights. This algorithm is fast enough to beat any modern implementation of BWT. it is stable as well, as the time of execution does not vary match with the level of randomness or repeated patterns, as it does with the other sorting techniques. At first we will explain how it works as a linear algebra problem, then we will give an implementation example in C sharp.

```
public int[] LeagueTableSort(char[] arr)
       {
           int strLen = arr.Length / 2; // arr is in fact two arrays concatinated to avoid
           int[] finalIndex = new int[strLen]; // where final weights are stored it is equa
           int strLen_1 = strLen - 1; // strLen -1
           int strLen_2 = strLen - 2;
           int strLen_1_s = 0;  // strLen - 1 - s
           int flag = 0;  // to carry previous sign
           int flag1 = 0; // to remember previus action
           int ipS = 0;   // i + s
           int ip1 = 0;   // i + 1
           int ipspstrLen = 0; // i + s + strLen
           // calculate first letter weights
           byte[] relPos1 = Encoding.ASCII.GetBytes(arr);
           for (int i = 0; i < strLen; i++) { finalIndex[i] = strLen * (int)relPos1[i]; }
           // loop through othe cells
           for (int s = 1; s < strLen_1; s++)
```

```
            {
                strLen_1_s = strLen_1 - s;
                ipS = strLen_2 + s;
                for (int i = strLen_2; i > strLen_1_s; i--)
                {
                    if (arr[i] == arr[ipS])
                    {
                        ip1 = i + 1;
                        if (flag1 == ip1) { }
                        else
                        {
                            if (arr[ip1] > arr[ip1 + s]) { flag = 1; } else { flag = -
1; }
                        }
                        ipspstrLen = ipS - strLen;
                        finalIndex[i] += flag;
                        finalIndex[ipspstrLen] -= flag;
                        flag1 = i;
                    }
                    ipS -= 1;
                }
            }
            return finalIndex;
        }
```

This 'LeagueTableSort' algorithm is much faster than any application of suffix array sort or quick sort. The next table provide some statistics.