League Table Weights, LTW for BWT

by

Abderrahim Hechachena

27/05/2018

abderrahimhechachena@yahoo.co.uk

BWT, Theory and Practice

In this paper we shed some light on borrow wheeler transform as a method used mainly for compression and genetics. This transform has being implemented using different techniques, some are experimental some are professional applications, tuned for speed. Among these methods; Quick sort, bucket sort and suffix arrays. The problem with the first two is when data have high level of repeated patterns they tend to loose speed, as they need more time to compare strings. suffix array may needs more memory to store objects, hence when strings get larger the need for more RAM memory become a problem. In this paper we will explain how a new method dedicated to BWT may be more relevant for this transform. I dubbed this method 'league table weights', LTW for short. The reason for this name is the way the algorithm works. BWT according to LTW is a simple problem of Linear algebra W*v where W is the weights matrix and v is a vector of ones.

First lets look at how LTW works: 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 weights matrix W. 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.

League Table Weights, LTW R Example

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

- 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.

```
> # 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 and calculating the characters weights and producing the final transformed string.

```
> # generate string of length n
> n = 5
> s = paste(sample(letters, n, replace = T),collapse = "")
> for(i in 1:2){s = paste(s,s,sep="")}
> s

[1] "crfyncrfyncrfyncrfyn"
```

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

 [1] 1 1 1 1 1 1 1 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

     c  r  f y  n  c  r  f  y  n  c  r  f  y  n  c  r  f  y  n  !
1c   0  1  1 1  1 -1  1  1  1  1 -1  1  1  1  1 -1  1  1  1  1 -1
2r  -1  0 -1 1 -1 -1 -1 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1  1 -1 -1
3f  -1  1  0 1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1
4y  -1 -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
5n  -1  1 -1 1  0 -1  1 -1  1 -1 -1  1 -1  1 -1 -1  1 -1  1 -1 -1
6c   1  1  1 1  1  0  1  1  1  1 -1  1  1  1  1 -1  1  1  1  1 -1
7r  -1  1 -1 1 -1 -1  0 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1  1 -1 -1
8f  -1  1  1 1  1 -1  1  0  1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1
9y  -1 -1 -1 1 -1 -1 -1 -1  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
10n -1  1 -1 1  1 -1  1 -1  1  0 -1  1 -1  1 -1 -1  1 -1  1 -1 -1
11c  1  1  1 1  1  1  1  1  1  1  0  1  1  1  1 -1  1  1  1  1 -1
12r -1  1 -1 1 -1 -1  1 -1  1 -1 -1  0 -1  1 -1 -1 -1 -1  1 -1 -1
13f -1  1  1 1  1 -1  1  1  1  1 -1  1  0  1  1 -1  1 -1  1  1 -1
14y -1 -1 -1 1 -1 -1 -1 -1  1 -1 -1 -1 -1  0 -1 -1 -1 -1 -1 -1 -1
15n -1  1 -1 1  1 -1  1 -1  1  1 -1  1 -1  1  0 -1  1 -1  1 -1 -1
16c  1  1  1 1  1  1  1  1  1  1  1  1  1  1  1  0  1  1  1  1 -1
17r -1  1 -1 1 -1 -1  1 -1  1 -1 -1  1 -1  1 -1 -1  0 -1  1 -1 -1
18f -1  1  1 1  1 -1  1  1  1  1 -1  1  1  1  1 -1  1  0  1  1 -1
19y -1 -1 -1 1 -1 -1 -1 -1  1 -1 -1 -1 -1  1 -1 -1 -1 -1  0 -1 -1
20n -1  1 -1 1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1  1 -1  1  0 -1
21!  1  1  1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0
```

N is asymmetric, with 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 value is 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. BWT 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)

      c  r  f  y n   c  r  f  y n   c r  f  y n   c r   f  y  n   !
[1,] -12 12 -4 20 4 -14 10 -6 18 2 -16 8 -8 16 0 -18 6 -10 14 -2 -20

> 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" "n" "n" "!" "r" "r" "r" "r" "y" "y" "y" "y" "c" "c" "c" "c" "f" "f"
[20] "f" "f"
```

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 weight 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'.

```
> 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
```

```
     c  r  f y  n  c  r  f y  n  c  r  f y  n  c  r  f y  n  !
1c   0  1  1 1  1  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1 -1
2r  -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1
3f  -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1
4y  -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1
5n  -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1
6c   0  1  1 1  1  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1 -1
7r  -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1
8f  -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1
9y  -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1
10n -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1
11c  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1 -1
12r -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1
13f -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1
14y -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1
15n -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1
16c  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1  0  1  1 1  1 -1
17r -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1  0 -1 1 -1 -1
18f -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1  1  0 1  1 -1
19y -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1 -1 -1 0 -1 -1
20n -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1  1 -1 1  0 -1
21!  1  1  1 1  1  1  1  1 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] "nnnn!rrrryyyyccccffff"
```

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] "crfyncrfyncrfyncrfyn"

> ################################################################################
> # 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. Hope some body may develop it further. Best wishes.