League Table Sort or

The weighting algorithm

for solving

Borrow Wheeler Transform

By

Abderrahim Hechachena

27/05/2018

abderrahimhechachena@yahoo.co.uk

In this paper we will expose the weighting algorithms to solving BWT. It is a fast algorithm that may be developed further for practical uses, such as data compression and search engines.

The League Table Sort applied to BWT; output = W.v where W is a weighting matrix and v is a vector of ones.

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 = 'ueyhfghrygfueyhfghrygf', of length n, add the character '!' at the end of the string to get 'ueyhfghrygfueyhfghrygf!'. Now build a weight matrix W[n + 1, n + 1] with column names and row names using 'ueyhfghrygfueyhfghrygf!'. 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 cells 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 in the input string. Note, diagonal elements are not counted as they will contain 0 each. Now we have W as the weight matrix, v is a vector 1 of n + 1 length. This is a linear solution to the BWT. W.v is the weighting index that will be used to sort the original input string to get the output string. This algorithm is stable, as it needs the same number of steps regardless of the randomness or presence of patterns in the input data. it beats any existing BWT sorting methods such as quick sort and suffix arrays by a heavy margin, not counting the trivial example where data is fully random. We will provide a more sophisticated example later using java and parallel processing.

An introduction example using R programming. This is not optimal but it provides a simple example.

```
> 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
+ }
> ##################
> getIndex = function(size, buckets, strEncoded, indices){
+   for (i in 1:size){
+     buckets[strEncoded[i]] = buckets[strEncoded[i]] + 1
+     indices[buckets[strEncoded[i]]] = i
+   }
+   indices
+ }
> ######################
>
> # reverse BWT process to test the above code
> 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
+   }
+   ###
+   indices = getIndex(size, buckets, strEncoded, indices)
+   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
+ }
> ###############################################
> # example:
> inputString = 'ueyhfghrueyhfghr'
> n = nchar(inputString)
> v = rep(1,n+1)
> # generae the weights matrix W
> d = generateWeightMatrix(inputString,1)
> W = as.matrix(d)
> W;det(W)

     u  e y  h  f  g  h  r  u  e  y  h  f  g  h  r  !
1u   0 -1 1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
2e   1  0 1  1  1  1  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 -1 -1 -1 -1
4h   1 -1 1  0 -1 -1  1  1  1 -1  1 -1 -1 -1  1  1 -1
5f   1 -1 1  1  0  1  1  1  1 -1  1  1 -1  1  1  1 -1
6g   1 -1 1  1 -1  0  1  1  1 -1  1  1 -1 -1  1  1 -1
7h   1 -1 1 -1 -1 -1  0  1  1 -1  1 -1 -1 -1 -1  1 -1
8r   1 -1 1 -1 -1 -1 -1  0  1 -1  1 -1 -1 -1 -1 -1 -1
9u   1 -1 1 -1 -1 -1 -1 -1  0 -1  1 -1 -1 -1 -1 -1 -1
10e  1  1 1  1  1  1  1  1  1  0  1  1  1  1  1  1 -1
11y -1 -1 1 -1 -1 -1 -1 -1 -1 -1  0 -1 -1 -1 -1 -1 -1
12h  1 -1 1  1 -1 -1  1  1  1 -1  1  0 -1 -1  1  1 -1
13f  1 -1 1  1  1  1  1  1  1 -1  1  1  0  1  1  1 -1
14g  1 -1 1  1 -1  1  1  1  1 -1  1  1 -1  0  1  1 -1
15h  1 -1 1 -1 -1 -1  1  1  1 -1  1 -1 -1 -1  0  1 -1
16r  1 -1 1 -1 -1 -1 -1  1  1 -1  1 -1 -1 -1 -1  0 -1
17!  1  1 1  1  1  1  1  1  1  1  1  1  1  1  1  1  0

[1] 0

> # calculate weights
> result = t(W)%*%v
> t(result)

      u   e  y h  f  g h r  u   e  y  h   f  g h r   !
[1,] 12 -12 16 0 -8 -4 4 8 10 -14 14 -2 -10 -6 2 6 -16
```

```
> # calculate output
> S = strsplit(paste('!',inputString,sep = ""),"")
> d1 = data.frame(unlist(S),as.numeric(result))
> names(d1) = c('S', 'w')
> outputString = paste(as.character(as.factor(d1[order(d1$w),][,1])),collapse = '')
> outputString

[1] "ruuhhffyygghhr!ee"

> bwt_decode(outputString);inputString

[1] "ueyhfghrueyhfghr"

[1] "ueyhfghrueyhfghr"

> # zero cells are the off diagonal zero cells
> zeroCells = generateWeightMatrix(inputString,0)
> zeroCells

     u  e y  h  f  g  h  r  u  e y  h  f  g  h  r  !
1u   0 -1 1 -1 -1 -1 -1 -1  0 -1 1 -1 -1 -1 -1 -1 -1
2e   1  0 1  1  1  1  1  1  1  0 1  1  1  1  1  1 -1
3y  -1 -1 0 -1 -1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1 -1 -1
4h   1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
5f   1 -1 1  1  0  1  1  1  1 -1 1  1  0  1  1  1 -1
6g   1 -1 1  1 -1  0  1  1  1 -1 1  1 -1  0  1  1 -1
7h   1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
8r   1 -1 1 -1 -1 -1 -1  0  1 -1 1 -1 -1 -1 -1  0 -1
9u   0 -1 1 -1 -1 -1 -1 -1  0 -1 1 -1 -1 -1 -1 -1 -1
10e  1  0 1  1  1  1  1  1  1  0 1  1  1  1  1  1 -1
11y -1 -1 0 -1 -1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1 -1 -1
12h  1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
13f  1 -1 1  1  0  1  1  1  1 -1 1  1  0  1  1  1 -1
14g  1 -1 1  1 -1  0  1  1  1 -1 1  1 -1  0  1  1 -1
15h  1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
16r  1 -1 1 -1 -1 -1 -1  0  1 -1 1 -1 -1 -1 -1  0 -1
17!  1  1 1  1  1  1  1  1  1  1 1  1  1  1  1  1  0
```

The output above provided two weighing matrices. The first one is final as it contains all weights except the diagonal elements. This is the matrix used to derive the final BWT using W.v. The second matrix is the one we are going to use to explain more advanced techniques for speed and performance.

Java solution and parallel processing

```
> # zero cells are the off diagonal zero cells
> zeroCells = generateWeightMatrix(inputString,0)
> zeroCells
```

```
     u  e y  h  f  g  h  r  u  e y  h  f  g  h  r  !
1u   0 -1 1 -1 -1 -1 -1 -1  0 -1 1 -1 -1 -1 -1 -1 -1
2e   1  0 1  1  1  1  1  1  1  0 1  1  1  1  1  1 -1
3y  -1 -1 0 -1 -1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1 -1 -1
4h   1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
5f   1 -1 1  1  0  1  1  1  1 -1 1  1  0  1  1  1 -1
6g   1 -1 1  1 -1  0  1  1  1 -1 1  1 -1  0  1  1 -1
7h   1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
8r   1 -1 1 -1 -1 -1 -1  0  1 -1 1 -1 -1 -1 -1  0 -1
9u   0 -1 1 -1 -1 -1 -1 -1  0 -1 1 -1 -1 -1 -1 -1 -1
10e  1  0 1  1  1  1  1  1  1  0 1  1  1  1  1  1 -1
11y -1 -1 0 -1 -1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1 -1 -1
12h  1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
13f  1 -1 1  1  0  1  1  1  1 -1 1  1  0  1  1  1 -1
14g  1 -1 1  1 -1  0  1  1  1 -1 1  1 -1  0  1  1 -1
15h  1 -1 1  0 -1 -1  0  1  1 -1 1  0 -1 -1  0  1 -1
16r  1 -1 1 -1 -1 -1 -1  0  1 -1 1 -1 -1 -1 -1  0 -1
17!  1  1 1  1  1  1  1  1  1  1 1  1  1  1  1  1  0
```

- 1) The weight matrix should contain non zero values only, either 1 or -1 except the diagonal elements. Note that the cell (1u, u) = 0, the cell (2e, e) = 0 as well, to calculate the value of this cell we need to move down to cell (9u, !) which contains -1. This means all cells between them should have the value -1. in short every zero cell (i, j) should have its value from cell (i+1, j+1) if it is not zero otherwise we keep moving down till we find a non zero cell. This move is from left to right, it is not optimal as it requires to keep track of what is being done. This task may prove to be chanlanging if we are dealing with a huge matrix. Instead we are going to move from right to left to fill the zero cells.

- 2) The above remark is just saying columns zero value are dependent on the values of the columns on the right.

- 3) When we calculated the weight matrix we created a matrix of weights W, in fact we don't need this matrix at all, we can just increment the final weighting array W.v. So there is very low demand on RAM for calculations.

- 4) We notice that the weighting matrix is symmetric with opposite signs values. this may reduce calculation time further, as we need only to work with the upper triangular matrix.

- 5) The only column with all values not equal to zero is the right most column. So we have to start calculation from the right to fill zero cells.

- 6) If we have to split the weight matrix into vertical blocks of columns then every block has to start with the right column fully calculated first.

this could only happen if we move from the left to right to fill the target column.

- 7) For parallel processing we choose the braking points where columns have to meet the condition 5: example a set of 10,000,000 columns has to be split into 10 parallel chunks to be processed by 10 processors. At the end of each million column we fill the last column by moving right to find the adequate values for zero cells as stated in condition 1. Then we calculate each block using separate process starting from the right. Example: if we split the weighting matrix into three blocks, the block on the right start with ! and the other two blocks start with f and f, since ! is always a full column we need only to fill the columns f and f then start from !, f and f to calculate the three block on three different processes.

- 8) Since the values of all non zero cells do not change, we need only to calculate those values for a set of unique vectors and save these unique vectors for filling zero positions on all columns. This process may required some RAM for speed, step 3 is still valid though as it does not necessarily require to store objects in RAM.

The attached Java example explain how we would work with single block 'one process'. extension to multiple processes is straight forward as explained above in step 7