Linear algebra solution

to

Borrow Wheeler Transform

By

Abderrahim Hechachena

27/05/2018

abderrahimhechachena@yahoo.co.uk

In this paper we will expose two different algorithms 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 = 'ueyhfghrygfueyhfghrygf', of length n, add the character '!' at the end of the line 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 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 cell. 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.

```
> # league table  method weighting matrix generator.
> # the inputStr1 could be 'ueyhfghrygfueyhfghrygf!'
> # the zero = 1 option generate the full matrix
> # the zero = 0 option do not fill the zero cells
> 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
+ }
> ###################
> # 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
+   }
+   ###
+   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
+ }
> ##############################################
> # 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

     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

> # 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 cell 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
```

Fast application of the LTW to solve BWT

For optimal algorithms lots of steps need to be cut to speed up the process. this algorithm is highly competitive against the fastest sorting algorithms. some examples of this solution implemented in C sharp are bellow.

```
 public int[] LeagueTableSort(char[] arr)
{
    int strLen = arr.Length / 2;
    int strLen1 = strLen - 1;
    int[] relPos = new int[strLen];
    char ai;
    char aA;
    int flag = 0;
    //Parallel causes havoc
    for (int s = 1; s < strLen; s++)
    {
        for (int i = (strLen1); i >= 0; i--)
        {
```

```
            ai = arr[i];
            aA = arr[i + s];
            if (ai == aA) { }
            else
            {
                if (ai > aA) { flag = 1; } else { flag = -1; }
            }
            relPos[i] += flag;
        }
    }
    return relPos;
}
```

The time of this algorithm could be halved by looping through the upper triangular weight matrix only. It could be reduced further by writing only to zero cells.

The code below may work, but still has a bug, enjoy the fun. To process the string 'abldldldlljd' add to it '!' at the end, then concatenate to get 'abldldldlljd!abldldldlljd!', split this string to get the input array arr. The reason for this is to avoid referencing complexities while we compare characters. This code is derived from the above solution. It may need some optimization, but as is, is already faster than quick sort and suffix array. It used very little RAM and it is stable time wise.

```
 public int[] LeagueTableSort(char[] arr)
{
    int strLen = arr.Length / 2;
    int[] finalIndex = new int[strLen];
    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 other 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;
}
```