THE BURROWS-WHEELER TRANSFORM.

WAHOME GITHIRE

MINERVA SCHOOLS AT KGI.

Author Note

First paragraph: Overview of the algorithm and main usage.

Second paragraph: Improvement intended by algorithm implementation.

Third paragraph: Analysis of algorithm i.e time and space complexities.

Fourth paragraph: Trade offs and useful scenarios to apply algorithm.

Abstract

A lot of data in modern computation is parsed as text files be it JSON, XML among others. For generality, let's say that a lot of program, digital files and signatures and moved around on a high level packaged in some combination of Unicode characters. The universe of unicode characters however is finite and thus in order to encode our data using these limited characters, we have to do so using their combinations and permutations. Inevitably, we end up repeating characters to facilitate this encoding. At first glance this may seem like a bad thing but computationally, it is a blessing in disguise. Without going into detail, I will give an example "**AAABBBCCCAAABBBC"** can be represented as "**A3B3C3A3B3C1"**. This is a compression of the original string by the running length algorithm which we shall encounter later on.

Keywords: Unicode characters, Running length,Compression.

The Burrows Wheeler Transformation.

We have touched on compression and repetitions. By definition, compression implies shortening, abbreviation and changing the format of data or some solid such that it will require less space to store than it does in Its current state. Repetitions play a key role as the facilitate the representation of a whole set of units using only one of the units and the frequency of occurrence. This form of compression, the run length encoding is so intuitive to us that we rarely notice how much we use it. A nice example is our way of naming numbers. We call 10 units of 1s a 10 which is equivalent to 10*1. A thousand is equivalent to 1000 ones, 10 hundredths among other. We are used to bundling units into packages and their frequency of occurrence and the key to it is having repeating unit.

Repetitions however will be useless if we use them to represent the entire set but we cannot really reverse them to obtain our original data. An example, in the string "**AABABCCCDA",** we can say we have "**A4B2C3D1".** Decompressing this by the run length algorithm generates "**AAAABBCCCD"** which is not reflective of our original data. Even with repeating characters, we need to add an extra factor to maintain data integrity upon compression, the order of characters in the original string. Should we capture all characters, their order and frequency in a manner that is reversible to generate the exact original data, we say we have lossless data compression.

The Burrowers Wheeler algorithm *is a block sorting, lossless, data compression algorithm* developed by M Burrows in 1994, built on a similar but unpublished idea by D.J Wheeler. The algorithm takes in a string and returns a permutation of it such that there will be an increasing number of repeating characters, and first row index to facilitate reversal.((Burrows,Wheeler 1994))

BWT, How does it work?

BWT transformation.

The first step is performing a **rotation** of the string in a reduced manner such that you perform a one step rotation, then the second step rotation on the result of the first rotation, the third rotation on the result of the second rotation and so on and so forth. This taxing computationally and space wise as we end up generating an **N*N** matrix.This arises a worst case space complexity of **O(N^2)** given that N is the length of our matrix. The rotations themselves involve N swaps for each character along the string of length N and this operates on N strings thus an **O(N^2)** complexity since one of the dimensions of our matrix is an instance of the rotated set whose length is N. AN example is the matrix below that is generate when you rotate the words: **"Love you mama!."**

S0	**'Love you mama!'**
S1	**'ove you mama!L'**
S2	**'ve you mama!Lo'**
S3	**'e you mama!Lov'**
S4	**' you mama!Love'**
S5	**'you mama!Love '**
S6	**'ou mama!Love y'**
S7	**'u mama!Love yo'**
S8	**' mama!Love you'**
S9	**'mama!Love you '**
S10	**'ama!Love you m'**
S11	**'ma!Love you ma'**
S12	**'a!Love you mam'**
S13	**'!Love you mama'**

Lexicographical sorting: As soon as the matrix is generated as so, the list will then be lexicographically be sorted. What lexicographically implies is dictionary order. Imagine looking up the first words on the specific string in a dictionary. The order in which they are found is how they will be sorted with special characters coming first then a,b,c and so on and so forth. Note the position of **S0** In the left column. After the sorting, **the index of wherever it ends up** is the index returned by the BWT transform **along the string of TERMINAL characters**. These along with the first column of the matrix above will be used to backtrack and restore the original string.((Burrows,Wheeler 1994)) This reversibility from the three pieces of data mean the transformation is lossless with respect to order in the long run. This sorting costs ast best **O(NLogN)**.

Below is a list of the matrices after sorting:

S8	**' mama!Love you'**
S4	**' you mama!Love'**
S13	**'!Love you mama'**
S0	**'Love you mama!'**
S12	**'a!Love you mam'**
S10	**'ama!Love you m'**
S3	**'e you mama!Lov'**
S11	**'ma!Love you ma'**
S9	**'mama!Love you '**
S6	**'ou mama!Love y'**
S1	**'ove you mama!L'**
S7	**'u mama!Love yo'**
S2	**'ve you mama!Lo'**
S5	**'you mama!Love '**

Space and time analysis of naive transform: The implementations above are naive, very space inefficient and demand a lot of computational power. They do however highlight the basis of of the transformation. THe space complexity is at worst O(N^2). In regards to processing, the sorting plus the swaps take a total O(NLOGN + O(N^2)) thus O(N^2). This is a naive implementation. We can optimize the performance and resource usage. An example is as noted, instead of having to generate the entire matrices and using N^2 cells,w e could just generate the first and last column through special permutations and swaps for a final space complexity of O(2N) thus O(N). This will be especially useful for reversing the results of the transform.

Outcome of the transform: As stated there before, for the transformation to be valid, it should return a string of the terminal character after performing the lexicological sorting. The resulting index of the original index S0 will also be returned as a starting for the backtracking. A final part of this is the first column although this is easy to infer from should we know the frequency of occurrence of the individual characters before. This in our implementation is kept in memory through a hash table making use of memoization with the character occurence displaying the dynamic programming[1] properties of optimal substructure(The count of the latest of any characters we encounter is the maximum number of times we have seen the character 'till now') and also overlapping subproblems such that instead of counting the characters from scratch every time we encounter it, being composed of overlapping counts, we can build on the previous counts by just adding one to the memoized dataset.This ranking is called the **T Ranking** of characters in the columns. The matrix has a special property called the **LF-Mapping property(Last Column to first column mapping property.)** which states that "the ith occurrence of a character c in L(Last column) and the ith occurrence of c in F corresponds to the same occurence in T"((Mark Nelson nd.)) ((CMU-BWT-2017))Basically the T ranking orders are retained even after the sorting of the lists. This property is what allows us to infer the first column from the last one and avoid generating the entire matrix. In my implementation I generated the matrix yes but only because I needed it to print out some of the values and illustrate

[1] **#cs110-dynaprog: A subtle observation of how the dynamic properties manifest and taking advantage to implement a more efficient reverse algorithm.We memoize the frequency of characters occurences to prevent having to recount them when assigning the T rank so our TRANK implementation is actual DP.**

the transformations I am stating but the output is solely a string of the terminal characters and can be generated by not keeping a record of the elements as lists rather, using a variable which changes values instead of saving the values as the loop runs out and returns the last value encountered when it breaks.((CMU-BWT-2017))

The result set of running BWT(“*Love you mama!.****”):***

Terminal chars String:	'uea!mmva yLoo '
Index of S0	3

Why transform then? Now here is the juicy part, transforming by BWT has a tendency to stream similar characters together while have the reversibility somehow encoded from the T ranks and Index. See this example where **‘m’** in **‘mama’** and ‘**o**’ in ‘**love’** and ‘**you’** were brought together and by the Run Length encoding, can be represented as m2. While this is not a particularly impressive compression, this bringing together of streams of similar characters tends to be very useful. For instance, this is the results returned when you transform :**“ananas and a man bad banana man”**

Terminal chars String:	‘dsndaa nbmnbmnn n na aaaaaaaa’
Index of S0	20

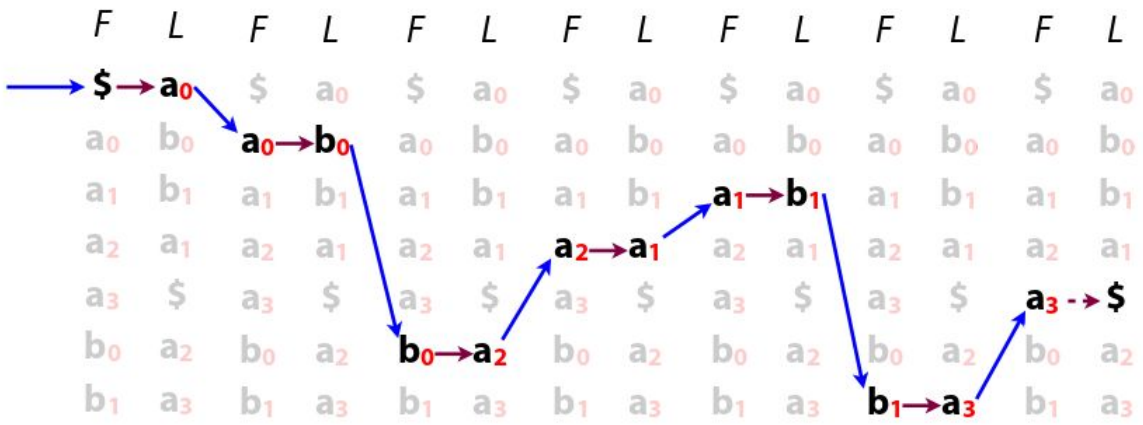
Its fascinating how all those ‘a’ are brought together and we can just call them **a8.**

Inversion of the string.

At this stage we should be compressing the returned string but the focus of this article is on the BWT so we will first discuss the inversion technique and later on discuss the Run length encoding and Its inversion to generate the string BWT returned.

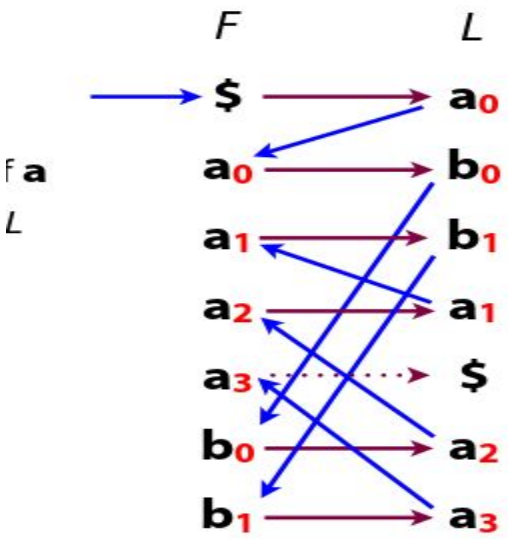
Data: BWT transformation returned the terminal strings and the starting index.

The algorithm can trace the path of the string during transformation through the T-ranks if we had the matrix. See this example of how that can happen if we had the string **"abaaba$"**:



RETRIEVED FROM (CMU-BWT-2017)

In our implementation, we will follow the corresponding Character and T ranks when shifting from the L column to the F one and directly to the immediate L when shifting F from F to L on alternative FL and columns building up to the final string as follows:



RETRIEVED FROM (CMU-BWT-2017)

Our implementation will perform the backtrack above to restore our original string as:

“$a0--b0-a2-a1-b1-a3-$”=abaaba”

We will generate the First column and assign T ranks to it, then proceed to match it to the L column as above. A naive implementation will have the tranks assigned already and saved as tuples for the L column hence the reshuffling but observe the order of the Tranks is retained. Our implementation will take the easier implementation and generate the F column is just again Lexicographically costing **O(nlogn)**

Compression: Run length algorithm

There are number of algorithms that can be used to compress the generated terminal characters string including Huffman coding,Move to front among others. In our implementation we will used the run length algorithm which keeps count of how many times we encounter a specific character and then pairs the specific character and the count till last encounter of the character before a new character is encountered and the count is restored to zero. That way, “aaaaaabbbbbccccccc” is transformed to “a5b5c6”((Imran, 2017)). This clearly is shorten than the original. Reversing this involves writing the characters as many times as the frequency post fixing them. We can do this encoding in linear time O(n) as we need only scan our list once keeping counts as stated above. The reversal too can be done is similar running time but Its O(M) where M is the length of the shortened string. The more repeating characters we have, the more efficient our compression.

NOTE:This approach is not entirely fair as it will sometimes retur a longer string and even a string of similar length. An example is parsing in the string “abcde”, this returns “a1b1c1d1” which is clearly longer. Similarly, “aabbccdd” returns “a2b2c2d2” which has a similar number of characters as the original list. A nice way avoid these pointless encoding for the single characters is not postfixing 1 if a character occurs once. We can then infer that a 1 was there if two non number characters follow each other. This again raises the question, what if we are compressing numbers? We can save the pairs and their frequencies in a hash table or a list of tuples. This will be easier to decode and will be the implementation we use.

As for the 2 common strings following each other, we can avoid the extra computation by just not swapping the second string with the frequency. We just return the sequence of 2 characters and then as above, in our decode, ensure if Its two similar characters and a different character after, we just return the string as is.

“$$$$$abbccdeffgggggg” thus returns **“$5abbccdeffg6”** which is more efficient than the naive Run compression: **“$5b2c2d1e1f2g5”.** This is less taxing computational and compresses the string even further.. The compression and decompression all cost at most **O(N) space complexity** with optimization and **O(2N)->O(N)** as is the case with purely unique characters. Time complexity is **O(N) in both cases** as we must compare the strings in reducing order.[2]

[2] **#cs110-complexity**:I have given the space and time complexity of algorithms used in the simulation and the focus section, BWT as well as the compression I deployed.

These are the results of our compression of **“ananas and a man bad banana man"**

Original:	**“ananas and a man bad banana man"**
Transformed version:	**“dsndaa nbmnbmnn n na aaaaaaaa”**
Compression tuple list:	**[('d', 1), ('s', 1), ('n', 1), ('d', 1), ('a', 2), (' ', 1), ('n', 1), ('b', 1), ('m', 1), ('n', 1), ('b', 1), ('m', 1), ('n', 2), (' ', 1), ('n', 1), (' ', 2), ('n', 1), ('a', 1), (' ', 2), ('a', 8)]**
Run length compression:	**“d1s1n1d1a2 1n1b1m1n1b1m1n2 1n1 2n1a1 2a8”**
With optimization:	**“dsndaa nbmnbmn na a8”**

Applications:

This transformation is applicable in any form of data storage or transfer networks that have a small universe of characters which can imply a lot of possible repetitions. Computers communicate in bits of 0s and 1s hence the repetition aspect tends to manifest itself a lot.

A more common application is the Bioinformatics field where the algorithms is used to shorten DNA sequences in a manner that they can be reversed. Its commonly used in LDO9, LD10, LTPS09, TSO9 among others. It is a direct application of the algorithm in DNA sequencing and targeted resequencing projects. The algorithm plays a key role in this because even when compressed, the T Ranks order are retained so with that knowledge we can still identify the matches we look for in DNA sequencing without exactly having the old sequences as were in the original strands but given only the compressed versions. This is allowed by the LF Mapping property. Alongside an appropriate compression algorithm, we can store the millions of lines of DNA sequencing codes in very short lines representing them. As an illustration, our implementation will do so on a DNA strand and see how much shorter the compressed version of the is than the original.

An illustrative application would be in text based communication systems. We can use the algorithms to permute emails, texts and package bits prior to transferring them via our networks connections. This can be especially useful if we have weak networks but still want to retain our network communication speeds. It make sense to reduce the package size of our data for them to be transferred faster.

Implementation:

```
import random
class BWT:
  def __init__(self,string):
     self.string=string
  def transBWT(self):#Transform function
     tab1=[(self.string[i:] + self.string[:i])\
          for i in range((-len(self.string)), 0, 1)]
     finalIndSet=[x[len(self.string):]\
                for x in sorted([x[1]+str(x[0])\
                              for x in  enumerate(tab1)])]
     ind =finalIndSet.index('0')
     tab2=[x for x in sorted(tab1)]
     rotations="".join(zip(*tab2)[-1])
     return [rotations,ind]

  def inverseBWT(self, lCol, index):#Reverse finction
     def row(index):
        fCol = sorted((t, i) for i, t in enumerate(lCol))
        for _ in lCol:
           t, index = fCol[index]
           yield t
     return ''.join(row(index))

class Compression:
  def __init__(self,string):
     self.string=string
#Run Length Code based on Rosetta code implementation # 1 None REGEX based, using  tuples.
#https://rosettacode.org/wiki/Run-length_encoding#Python
  def rLencode(self):#Normal RUn length encoding
     count = 1
     occurences = []
     for ch in range(1, len(self.string)):
        if (self.string[ch] != self.string[ch - 1]):
           if self.string[ch - 1]:
              occurences.append((self.string[ch - 1],count))
              count = 1
           else:
              count += 1
        else:
           occurences.append((self.string[ch],count))
```

```
        return occurences

    def rldecode(self,compr):#Run length decode from tuples, not string
        return "".join([pair[0]*pair[1] for pair in compr])

    def rlEncodeSTR(self):#Run length encoe as string
        count = 1
        lst = []
        for ch in range(1, len(self.string)):
            if (self.string[ch] != self.string[ch - 1]):
                if self.string[ch - 1]:
                    if count>2:
                        lst.append(self.string[ch - 1] + str(count))
                    else:
                        lst.append(self.string[ch - 1] *count)
                    count = 1
                else:
                    count += 1
            else:
                lst.append(self.string[ch] + str(count))
        return "".join(lst)
    def optimaCompress(self,charSeq):#Run length encode based on suggested optimizations
        finLis=[]
        for i,j in charSeq:
            if j>2:
                finLis.append(i+str(j))
            else:
                finLis.append(i*j)
        return "".join(finLis)

def test(mainStr,BWT):#Test finction
    BWT=BWT(mainStr)
    transformed=BWT.transBWT()
    compressorTrans=Compression(transformed[0])
    compressorNorm=Compression(mainStr)
    compressionTranEnc=compressorTrans.rLencode()
    compressionNormEnc=compressorNorm.rLencode()
    compressorTransStr=compressorTrans.rlEncodeSTR()
    compresionNormStr=compressorNorm.rlEncodeSTR()

    res= {"Transformed":transformed,
          "Compressed TransEncode":compressionTranEnc,
          "Compressed TransEncode Length": len(compressionTranEnc),
          "Compressed NormEncode":compressionNormEnc,
```

```
        "Compressed NormEncode Length": len(compressionNormEnc),
        "Compressed TranString[3]":compressorTransStr,
        "Compressed TranString Length": len(compressorTransStr),
        "Compressed NorString":compresionNormStr,
        "Compressed NorString Length": len(compresionNormStr),
        "Optimized Normal length":len(compressorNorm.optimaCompress(compressionNormEnc)),
        "Optimized Transformed length":
len(compressorTrans.optimaCompress(compressionTranEnc)),
        }#Used hash table to make look up speed during iteration faster
  return res
BWT=BWT
def reps(BWT,iters,countLen):#Iterative test function
  betterBWT, betterNorm, similar = 0, 0, 0
  OPbetterBWT, OPbetterNorm, OPsimilar = 0, 0, 0
  for i in range(iters):
     mainStr=''.join(random.choice("A B C D E F G H I J K L M N O P Q R S T U V W X Y
Z".split(" ")) for _ in range(countLen))
     results=test(mainStr=mainStr,BWT=BWT)
     if results['Compressed NorString Length']>results['Compressed TranString Length']:
        betterBWT+=1
     elif results['Compressed NorString Length']< results['Compressed TranString Length']:
        betterNorm+=1
     elif results['Compressed NorString Length']== results['Compressed TranString Length']:
        similar+=1

  print("NORMAL VERSION:For strings of length {}, Normal optimal did better {}% of the
times, Transformed compress did better {}% of the times.and they were similar {}% of the times"
       .format(countLen,(betterNorm / float(iters)), (betterBWT / float(iters)),(similar / float(iters))))
iters=500
count=10
reps(BWT,iters,count)
count=20
reps(BWT,iters,count)
count=50
reps(BWT,iters,count)
count=100
reps(BWT,iters,count)
count=200
reps(BWT,iters,count)
count=500
```

[3] **#cs110-optimal algorithm: Have used hash tables because of their lookup speeds during iterations faster, lists to facilitate fast insertions and did extra optimizations to even the compressor. I identified a pattern in how BWT did its transformation and i lieu of the matrix, conducted the direct swaps using the permutations package. All these aimed at optimizing the operating costs of this simulation.**

```
reps(BWT,iters,count)
count=1000
reps(BWT,iters,count)
```

[4]

SAMPLE RESULTS FROM ABOVE

```
/usr/bin/python2.7 "/home/brianwahome254/PycharmProjects/Final Project BWT/rotation.py"
NORMAL VERSION:For strings of length 10, Normal optimal did better 0.238% of the times, Transformed compress did better 0.192% of the times.and they were similar 0.57% of the times
NORMAL VERSION:For strings of length 20, Normal optimal did better 0.292% of the times, Transformed compress did better 0.292% of the times.and they were similar 0.416% of the times
NORMAL VERSION:For strings of length 50, Normal optimal did better 0.364% of the times, Transformed compress did better 0.402% of the times.and they were similar 0.234% of the times
NORMAL VERSION:For strings of length 100, Normal optimal did better 0.402% of the times, Transformed compress did better 0.436% of the times.and they were similar 0.162% of the times
NORMAL VERSION:For strings of length 200, Normal optimal did better 0.44% of the times, Transformed compress did better 0.468% of the times.and they were similar 0.092% of the times
NORMAL VERSION:For strings of length 500, Normal optimal did better 0.428% of the times, Transformed compress did better 0.498% of the times.and they were similar 0.074% of the times
NORMAL VERSION:For strings of length 1000, Normal optimal did better 0.417% of the times, Transformed compress did better 0.581% of the times.and they were similar 0.002% of the times
```

[4] **#cs110-novel application: Illustrated how the BWT algorithm alongside a compression one can be used to optimize storage space by compressing text files. In my application section and examples, I also explained how it is used with real life data like bioinformatics.**

With 1000 iterations:

```
/usr/bin/python2.7 "/home/brianwahome254/PycharmProjects/Final Project BWT/rotation.py"
NORMAL VERSION:For strings of length 10, Normal optimal did better 0.223% of the times, Transformed compress did better 0.176% of the times.and they were similar 0.601% of the times
NORMAL VERSION:For strings of length 20, Normal optimal did better 0.319% of the times, Transformed compress did better 0.293% of the times.and they were similar 0.388% of the times
NORMAL VERSION:For strings of length 50, Normal optimal did better 0.392% of the times, Transformed compress did better 0.395% of the times.and they were similar 0.213% of the times
NORMAL VERSION:For strings of length 100, Normal optimal did better 0.419% of the times, Transformed compress did better 0.434% of the times.and they were similar 0.147% of the times
NORMAL VERSION:For strings of length 200, Normal optimal did better 0.429% of the times, Transformed compress did better 0.443% of the times.and they were similar 0.128% of the times
NORMAL VERSION:For strings of length 500, Normal optimal did better 0.478% of the times, Transformed compress did better 0.456% of the times.and they were similar 0.066% of the times
NORMAL VERSION:For strings of length 1000, Normal optimal did better 0.46% of the times, Transformed compress did better 0.501% of the times.and they were similar 0.039% of the times
```

From the above simulations,BWT tends to do better as the size of the string gets larger. This is because the number of possible repeated sequences get larger thus Its strength gets showcased. This is not the case with smaller values hence they will tend to be similar or normal Run Length will do better. BWT relies on increased probability of repetitions. As to why it performs better as the data gets bigger? Think of it. We keep adding characters making the possibility of the permutation to be such that there are more runs without losing the order increase. THis thus implies BWT has a higher probability of generating runs and it does so indeed which makes it more and more effective with longer strings.

References

COS 226 Burrows-Wheeler Data Compression Algorithm. (n.d.). Retrieved from http://www.cs.princeton.edu/courses/archive/spr03/cs226/assignments/burrows.html (COS 226,nd)

Mark Nelson. (n.d.). Retrieved from http://marknelson.us/1996/09/01/bwt/ (Mark Nelson nd.)

Burrows Wheeler Transform. (n.d.). Retrieved December 15, 2017, from https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/bwt.pdf (CMU-BWT-2017)

Burrows, Michael; Wheeler, David J. (1994), A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, Retrieved December 12, 2017, from http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf (Burrows,Wheeler 1994)

Hoque, Imaran.(2017) "Run Length Encoding/Decoding." UCLA, Ucla.edu, https://www.ihoque.bol.ucla.edu/Report.doc. (Imran, 2017)