Context-based Encoding Burrows-Wheeler Transform

Computer Science Department
CS4481b/9628b: Image Compression
Winter 2017
Instructor: Mahmoud R. El-Sakka
Office: MC-419
Email: elsakka@csd.uwo.ca
Phone: 519-661-2111 x86996

The Burrows-Wheeler Transform

- The ***Burrows-Wheeler Transform*** (BWT) algorithm utilizes the context of the symbol being encoded to achieve lossless compression, but in a very different way
- The transform itself was initially developed by ***David Wheeler*** in 1983
- Yet, the BWT compression algorithm saw the light in 1994 by ***Michael Burrows*** and ***David Wheeler***
- Currently, BWT is used in data compression techniques such as ***bzip2***
- Unlike most of the previous algorithms that we have looked at,
  - the BWT algorithm requires that the entire sequence to be encoded be available to the encoder before the encoding takes place
  - the BWT decoding procedure is not immediately obvious once we know the encoding procedure

The Burrows-Wheeler Transform

- The algorithm can be summarized as follows

  Given a sequence of letters of length N,

  - Create $N-1$ other sequences where each of these $N-1$ sequence is a cyclic shift of the original sequence
    Total number of sequence is $(N-1)+1$ (the original sequence) $= N$
  - These N sequences are sorted in lexicographic order
  - The encoder encodes the last letter in each cyclically shifted and sorted sequence (i.e., encoding a sequence of length N letters)
  - The compressed file consists of two parts
    - This sequence of last letters, L
    - the position of the original sequence in the sorted list

Example: thisΔisΔthe

0	t	h	i	s	Δ	i	s	Δ	t	h	e
1	h	i	s	Δ	i	s	Δ	t	h	e	t
2	i	s	Δ	i	s	Δ	t	h	e	t	h
3	s	Δ	i	s	Δ	t	h	e	t	h	i
4	Δ	i	s	Δ	t	h	e	t	h	i	s
5	i	s	Δ	t	h	e	t	h	i	s	Δ
6	s	Δ	t	h	e	t	h	i	s	Δ	i
7	Δ	t	h	e	t	h	i	s	Δ	i	s
8	t	h	e	t	h	i	s	Δ	i	s	Δ
9	h	e	t	h	i	s	Δ	i	s	Δ	t
10	e	t	h	i	s	Δ	i	s	Δ	t	h

These N sequences to be sorted in lexicographic order

Example: thisΔisΔthe

4	Δ	i	s	Δ	t	h	e	t	h	i	s
7	Δ	t	h	e	t	h	i	s	Δ	i	s
10	e	t	h	i	s	Δ	i	s	Δ	t	h
9	h	e	t	h	i	s	Δ	i	s	Δ	t
1	h	i	s	Δ	i	s	Δ	t	h	e	t
2	i	s	Δ	i	s	Δ	t	h	e	t	h
5	i	s	Δ	t	h	e	t	h	i	s	Δ
3	s	Δ	i	s	Δ	t	h	e	t	h	i
6	s	Δ	t	h	e	t	h	i	s	Δ	i
8	t	h	e	t	h	i	s	Δ	i	s	Δ
0	t	h	i	s	Δ	i	s	Δ	t	h	e

- The *L* sequence is:

  s s h t t h Δ i i Δ e

- The original sequence appears as sequence number 10 in the sorted list

Example: thisΔisΔthe

- The encoding result will be:
  - The sequence *L* (***s s h t t h Δ i i Δ e***)
  - An index value 10
- Notice how many repetitive letters have come together
- If we had a longer sequence of letters, the runs of like letters would have been even longer
- All elements of the original sequence are contained in *L*
- The decoding process is just to figure out the permutation that will let us recover the original sequence

Example: thisΔisΔthe

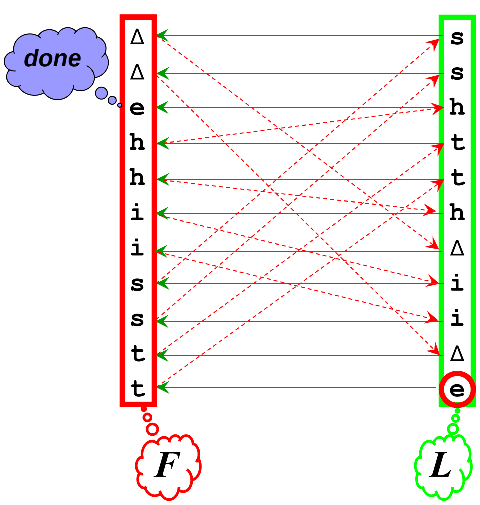
- The first step in obtaining the permutation is to generate the sequence *F*, which is the sequence of the first letter in each row, ***HOW?***
- That is a simple task to do because we lexicographically ordered the sequence
- The sequence *F* is simply the sequence *L* in lexicographic order

L = ***s s h t t h Δ i i Δ e***

lexicographic(*L*) = ***Δ Δ e h h i i s s t t***

	F										L
4	Δ	i	s	Δ	t	h	e	t	h	i	s
7	Δ	t	h	e	t	h	i	s	Δ	i	s
10	e	t	h	i	s	Δ	i	s	Δ	t	h
9	h	e	t	h	i	s	Δ	i	s	Δ	t
1	h	i	s	Δ	i	s	Δ	t	h	e	t
2	i	s	Δ	i	s	Δ	t	h	e	t	h
5	i	s	Δ	t	h	e	t	h	i	s	Δ
3	s	Δ	i	s	Δ	t	h	e	t	h	i
6	s	Δ	t	h	e	t	h	i	s	Δ	i
8	t	h	e	t	h	i	s	Δ	i	s	Δ
0	t	h	i	s	Δ	i	s	Δ	t	h	e

Example: thisΔisΔthe



	F										L
4	Δ	i	s	Δ	t	h	e	t	h	i	s
7	Δ	t	h	e	t	h	i	s	Δ	i	s
10	e	t	h	i	s	Δ	i	s	Δ	t	h
9	h	e	t	h	i	s	Δ	i	s	Δ	t
1	h	i	s	Δ	i	s	Δ	t	h	e	t
2	i	s	Δ	i	s	Δ	t	h	e	t	h
5	i	s	Δ	t	h	e	t	h	i	s	Δ
3	s	Δ	i	s	Δ	t	h	e	t	h	i
6	s	Δ	t	h	e	t	h	i	s	Δ	i
8	t	h	e	t	h	i	s	Δ	i	s	Δ
0	t	h	i	s	Δ	i	s	Δ	t	h	e

The Burrows-Wheeler Transform

- Why going through all this trouble?
- After all, we started with a sequence of length N and ended with a representation that contains $N+1$ elements!!
- It appears that we are actually causing expansion instead of compression
- The answer is:
  - this L sequence has a structure that makes it highly amenable to compression more than the original sequence
- Even in the previous small example, we have runs of repetitive symbols
- This will happen a lot more often when N is large
- Consider a large sample of text that has been cyclically shifted and sorted and consider all rows that begin with *he*Δ, for example
  - With high probability *he*Δ would be preceded by *t* to form *the*Δ
  - Consequently, in *L* we will get a long run of *t*’s

Move-to-Front Encoding

- A coding scheme that takes advantage of long runs of identical symbols is the *Move-to-Front* encoding scheme, or simply *MTF* (a.k.a. *Global Structure Transformation*, or simply *GST*)
- The scheme starts with an initial listing of the source alphabet, where each symbol in the alphabet is assigned a distinct number (code), for example, the *1st* alphabet symbol will be assigned 0, the *2nd* alphabet symbol will be assigned 1, and so on (codeword encoding)

0	1	2	3	4	5
Δ	e	h	i	s	t

- Once a particular symbol occurs,
  - The number (code) corresponding to its place in the list is transmitted
  - The symbol is *moved* to the *front* of the list
- This way, any run of repetitive symbols will be encoded by
  - a code for the symbol itself followed by a sequence of zeros

Move-to-Front Encoding

- In the previous example, *L* = ***s s h t t h* Δ *i i* Δ *e***
  and the alphabet symbols are given by ***{Δ, e, h, i, s, t}***
- We start out with the assignment

0	1	2	3	4	5
Δ	e	h	i	s	t

- The first element in *L* is *s*, which gets encoded as a 4
- We then *moves s* to the *front* of the list, which gives us this table

0	1	2	3	4	5
s	Δ	e	h	i	t

- The next *s* is encoded as 0
- Because *s* is already at the front of the list, we do not need to make any further changes
- The next element in *L* is *h*, which gets encoded as a 3
- We then moves *h* to the front of the list

Move-to-Front Encoding

- We then moves *h* to the front of the list *L* = ***s s h t t h* Δ *i i* Δ *e***

0	1	2	3	4	5
h	s	Δ	e	i	t

- The next element in *L* is *t*, which gets encoded as a 5
- We then moves *t* to the front of the list

0	1	2	3	4	5
t	h	s	Δ	e	i

- The next *t* is encoded as 0
- Because *t* is already at the front of the list, we do not need to make any further changes
- Continuing in this fashion, we get the sequence *4 0 3 5 0 1 3 5 0 1 5*

Move-to-Front Decoding

- To decode a move-to-front encoded sequence,
  - starts with an initial listing of the source alphabet, where each symbol in the alphabet is assigned a distinct number (code), i.e., the *1st* alphabet symbol will be assigned 0, the *2nd* alphabet symbol will be assigned 1, and so on

0	1	2	3	4	5
Δ	e	h	i	s	t

- Once reading a code from the encoded sequence,
  - The symbol corresponding to this code is decoded
  - The symbol is *moved* to the *front* of the list

Move-to-Front Decoding

- In the previous example, the *encoded* sequence = *4 0 3 5 0 1 3 5 0 1 5* and the alphabet symbols are given by ***{Δ, e, h, i, s, t}***
- We start out with the assignment

0	1	2	3	4	5
Δ	e	h	i	s	t

- The first code in the encoded sequence is *4*, which is decoded as *s*
- We then *moves s* to the *front* of the list, which gives us this table

0	1	2	3	4	5
s	Δ	e	h	i	t

- The next code in the encoded sequence is *0*, which is decoded as *s*
- Because *s* is already at the front of the list, we do not need to make any further changes
- The next code in the encoded sequence is *3*, which is decoded as *h*
- We then moves *h* to the front of the list

Move-to-Front Decoding

- We then moves *h* to the front of the list *4 0 3 5 0 1 3 5 0 1 5*

0	1	2	3	4	5
h	s	Δ	e	i	t

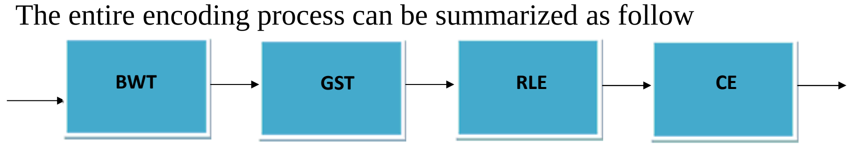
- The next code in the encoded sequence is *5*, which is decoded as *t*
- We then moves *t* to the front of the list

0	1	2	3	4	5
t	H	s	Δ	e	i

- The next code in the encoded sequence is *0*, which is decoded as *t*
- Because *t* is already at the front of the list, we do not need to make any further changes
- Continuing in this fashion, we get the sequence ***s s h t t h* Δ *i i* Δ *e***

Run-Length Encoding and Codeword Encoding

- While the resulting sequence is not too impressive, we should expect to see a large number of *0*'s and *small values* if the sequence to be encoded is larger
- The output of the *Move-to-Front* (*Global Structure Transformation*) is sent to a *Run-Length Encoder* to efficiently encode the repetitive zeros
- The *Run-Length Encoder* output is sent to a *Codeword Encoder*, e.g., Huffman or arithmetic encoder
- The entire encoding process can be summarized as follow



- Decoding is just the inverse of the above process