

Modeling

- Introduction and categorization
- Dictionary methods
- Context modeling

Introduction

- **Modeling**
 - constructing a representation of the source that generates data being compressed
- **Issues**
 - how to select elements of model
 - statistical information
- **Elements** [for text source]
 - characters
 - words
 - strings – variable length, fixed-length

Probabilistic

- Probabilistic modeling
 - collect statistics (weighted?) on el't freqs
 - used with probabilistic code (Huffman, AC)
- Non-probabilistic modeling
 - try to build “useful” strings (frequent or in some other way appealing) to be used as el'ts mapped to codewords
 - may “guess” about relative freqs without explicitly collecting statistics
- Ad hoc techniques
 - run-length encoding
 - digram/trigram coding
 - semantic techniques

Fixed, Static, Adaptive

Determine element set and working statistics

- **Fixed** model — **predetermined**
Example: assume **E** is most likely
- **Static** model — **remain constant** during compr'n process
 - based on particular file being compressed
 - requires pre-scan
 - must transmit model
(*Example:* which char is most likely in file?)
 - efficiency recoups overhead for long strings
- **Adaptive** model – progressively **modify** based on observations
 - assume initial choice
 - does not require pre-scan
Example: so far, **T** is most likely
later, **E** becomes most likely

Fixed, Static, Adaptive

Theorem

There is an adaptive model that will be only *slightly* worse than *any* static model.

A static model can be *arbitrarily* worse than an adaptive counterpart.

Examples

- Fixed model, fixed code — elements are chars, no statistics
- Fixed model, static code
elements are chars, Brown Corpus stats used to represent all files
- Static model, fixed code
elements are chars & char pairs, frequency ordered
code is: 60 6-bit codes and 256 12-bit codes
- Static model, static code
elements are chars, stats collected in pre-scan for Huffman code
- Adaptive model, fixed code
el'ts are 512 strings defined as compr'n proceeds, code is fixed-length
- Adaptive model, adaptive code
elements are chars, stats on frequency updated continuously

Dictionary Modeling

- the element set is a **dictionary** of strings
- input is parsed into substrings, each \in dictionary
output is encoding of dictionary indices
- if it is possible for input to be parsed in several ways
must have parsing strategy
- **defined-word** schemes
dictionary entries are *words*,
the meaning of which is independent of the file
Example: a *word* is delimited by a space or period
- **free-parse** schemes
dictionary determined dynamically as input is parsed

Move-to-Front

- defined-word model
- MTF data structure maintains list order
- encoder & decoder maintain identical list order
- when string M occurs and **is in** dictionary
encoder transmits M 's index, then updates list
decoder recovers M from list, then updates list
- when M **not in** dictionary
encoder transmits $k+1$ ($k =$ current list length),
transmits M , then updates list
decoder knows to receive M and then updates list

MTF Example

the car on the left hit the car I left

| <i>scan</i> | <i>transmit</i> | <i>list after transmission</i> | | | | | | |
|-------------|-----------------|--------------------------------|------|------|------|------|----|--|
| the | 1,the | the | | | | | | |
| car | 2,car | car | the | | | | | |
| on | 3,on | on | car | the | | | | |
| the | 3 | the | on | car | | | | |
| left | 4,left | left | the | on | car | | | |
| hit | 5,hit | hit | left | the | on | car | | |
| the | 3 | the | hit | left | on | car | | |
| car | 5 | car | the | hit | left | on | | |
| I | 6,I | I | car | the | hit | left | on | |
| left | 5 | left | I | car | the | hit | on | |

Implementing MTF – simple array

- **structure**

array LIST[*rank*] stores *values*

- **encoding *value***

sequentially search LIST to find that LIST[*r*] = *value*

MTF *rank* = *r*

NEWLIST[1] = LIST[*r*]

forall $i < r$, NEWLIST[$i+1$] = LIST[i]

- **average-case cost** = $O(\text{weighted avg of } \textit{rank})$

- typically $O(n)$

- suitable for small alphabets or skew pdf

Implementing MTF – splay tree

- **structure**

array $\text{TIME}[\text{value}]$ stores **timestamps** (index in msg)

self-adjusting b.s.t. searchable by timestamp

tree node stores **count** of its right subtree

- **encoding value**

$t \leftarrow \text{TIME}[\text{value}]$ (old timestamp)

locate node for **value** using key t

move node(**value**) to *root*, via seq of edge rotations (update **count** fields)

MTF rank $\leftarrow 1 + \text{count}$ (right subtree = nodes with timestamps $> t$)

$\text{TIME}[\text{value}] \leftarrow$ new timestamp

detach node(**value**), update its timestamp

concatenate left and right subtrees, L and R

 splay rightmost node x of L, attach R as x 's right subtree

combined tree becomes left subtree of node(**value**)

- **amortized cost** $= O(\log \text{rank}) = O(\log n)$

Lempel-Ziv Models

- very popular – good compression, fast
- adaptive dictionary modeling (free-parse)
- dictionary has *prefix property*
 $X \in \text{dictionary} \Rightarrow \text{all prefixes of } X \text{ also } \in \text{dictionary}$

LZ78 model

- greedy parse of input into strings of gradually increasing length
output is alternating seq of chars & indices
 - take longest prefix P of input that is in dictionary
 - take following char, c
 - transmit (p, c) , where $p = \text{index of } P$
 - add string Pc to dictionary

Lempel-Ziv Example

input: **aa-bbb--fffffsgggggg**

| <i>parse</i> | <i>add to dictionary</i> |
|----------------------------------|--------------------------|
| (λ, a) | a |
| (a, -) | a- |
| (λ, b) | b |
| (b, b) | bb |
| (λ, -) | - |
| (-, f) | -f |
| (λ, f) | f |
| (f, f) | ff |
| (ff, f) | fff |
| (λ, g) | g |
| (g, g) | gg |
| (gg, g) | ggg |
| (g, λ) | |

Unix Utility Compress

- uses Lempel-Ziv as outlined above
- inefficient during initial portion of input
- to achieve good compression,
must have long input to build string frequency
- dictionary size is finite
when dictionary is full, cannot add more strings
then compression performance degrades
- *Compress* monitors compression ratio
when CR deteriorates, clears dictionary

Another LZ model – LZ77

- output at each step is either a **single char** (C),
or a **string reference** in form (I, L) ,
where I = string index, L = string length

Example:

abbaabbbabab → **abba(1,3)(3,2)(8,3)**

- use relative index (**offset**) instead of absolute index,
references restricted to start within a **window**

abbaabbbabab → **abba(4,3)(5,2)(2,3)**

- decoding:

abba

abbaabb

abbaabbba

abbaabbbabab

LZ Variations – LZ77 Family

sliding window W of N chars (lookahead buffer B)
implied dictionary = all substrings $\in W$

LZ77 $P=(\text{offset,length})$ and $C=(\text{char})$ alternate

LZSS P, C distinguished by flag bit (needn't alternate)
use P only if it saves space (length ≥ 2)
char following P is beginning of next string

LZH code offsets and lengths using dynamic Huffman

ZIP 1 use Shannon-Fano coding

ZIP, GZIP use static Huffman coding
one for literals & lengths, one for offsets

- input partitioned into arbitrary blocks
(start new block when encoder thinks useful)
- trees stored in beginning of emitted blocks
- hash table for length 3 strings

LZ in PKWARE

from Ben Rudiak-Gould

LZSS with 3 predefined prefix codes for literal, length, offset

- **header** first two bytes
 - first byte = 0/1, define literals with normal/variable-encoding
 - second byte = 4/5/6, define window size = 1k/2k/4k
- **bitstream** is stored in little-endian order
 - **Example:** bytes 0F 15 represent bit stream 1111 0000 1010 1000
 - consists of an encoded sequence of Lempel-Ziv tokens
 - each codeword represents a literal byte, a (length,offset), or EOS
- **representation of tokens**
 - **first bit of codeword = 0** → literal byte (256 values)
 - normal-encoding is 8-bit little-endian binary numbers
 - variable-encoding: 1111 for 'blank', 5 bits for {Eaeilnorstu}

| | | | | | | | | | | |
|------------------|---|----|----|----|----|---|----|----|----|----|
| number of bits | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| number of values | 1 | 11 | 20 | 21 | 16 | 7 | 5 | 10 | 91 | 74 |

- **first bit of codeword = 1** → (length,offset) or end-of-stream

LZ in PKWARE

- representation of (length,offset) or end-of-stream

- bits for
- (1) representing length,
 - (2) representing most significant 6 bits of offset,
 - (3) representing rest of offset

- bits for lengths 2-519 (len=519 → EOS)

encodings for 518 values (3,2,4-519):

| | | | | | | | | | | | | |
|------------------|---|---|---|---|---|---|---|----|----|----|-----|-----|
| number of bits | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10 | 11 | 12 | 14 | 15 |
| number of values | 1 | 3 | 3 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |

- bits for upper six bits of offset

encodings for 64 values (0-63):

| | | | | | | |
|------------------|---|---|---|----|----|----|
| number of bits | 2 | 4 | 5 | 6 | 7 | 8 |
| number of values | 1 | 2 | 4 | 15 | 26 | 16 |

- bits for rest of offset

lowest 4/5/6 bits are catenated in little-endian order

exception: if length=2 then only two low-order offset bits instead of 4/5/6, limiting offset to 256

LZ Variations – LZ77 Family

- **inefficiency of LZ77**
if window has multiple copies of a string,
only the last copy is needed
so some (offset,length) values can never get used

LZ Variations – LZ78 Family

real dictionary = previously parsed strings
trie structure to find match
add entry = previous match + next char

LZ78 P =(pointer) and C =(char) alternate
when memory exhausted, clear dictionary

LZW only ptrs, 12 bits (fixed)
initialize dict = {all $\sigma \in \Sigma$ }

LZT free space by discarding LRU phrases
(self-organizing list)

LZMW add entry = concat previous 2 phrases

LZAP add entry = all prefixes of
(concat of prev 2 phrases)

LZ Variations – LZW subFamily

dict = previously parsed strings
trie structure to find match
add entry = previous match + next char
initialize dict = { all $\sigma \in \Sigma$ }
only pointers, dict size variable

LZC (Unix compress)

$\lceil \lg |\text{dict}| \rceil$ bits (increases 9-16)
when memory exhausted, dict becomes static
clear dictionary only if CR deteriorates

GIF graphics interchange format

b (bpp) initially 2 (b/w) or 8
dict has $b + 1$ bits, increasing to 12
mem full \rightarrow static, cleared when CR bad
(indicated by encoding $2^b = \text{clear code}$)

LZ Variations – LZW subFamily

V.42bis

begin & max dict size are negotiated
transparent/compressed modes based on CR
dict full → recycle unused strings
limit on string length

Other LZ Variations

LZJ implied dict = *unique* substrings
up to max length $h(\approx 6)$
trie of fixed depth h
fixed size dict (≈ 8192)
when dict full, prune by removing
substrings that occurred only once

LZFG hybrid of LZ77 and LZ78
77: sliding window
ptrs include length indicator
78: trie structure
insert only complete phrases

Compare LZ77 & LZ78

- both LZ77 & 78 are **universal** coding methods
the j -th order redundancy is $O(\lg \lg n / \lg n)$
therefore as good as any block code, but
 - the convergence to entropy is slow
 - practical data sources are non-stationary
so dictionary purging is useful
- LZ77 tends to parse data sequences into longer (therefore fewer) phrases than does LZ78
- commercial implementations of LZ77 cut some corners (do not optimize compression) in order to increase speed of computation
 - employ heuristic to avoid exhaustive search of dictionary for longest match by accepting a “long enough” match

Context Modeling

- context determines codeword length for a char
 - simplest case: prev char parameterizes PDF
Example: if previous char is **q** then
extremely likely that current char is **u**
- maintain PDF for each possible “predecessor” char
 - in PDF(**q**), **u** gets 99%
 - in PDF(**t**), high prob for **h,a,e,i,o,u**
low prob for **b,d,f,g**, etc.
- more context → more accurate prediction
 - after **p**, high prob for **a,e,i,o,u,h,l,r,s,y**
 - after **philosop**, almost certain that next char is **h**

Context Modeling

- **order** is size of context
 - order-1 context model uses previous char
 - order-2 context model uses previous 2 chars
- **blending** uses contexts of several sizes
 - larger contexts: more accurate but less frequent
 - smaller context: more observed data
- **escape mechanism** – required for blending
 - encoder informs decoder when to use lower order model
- can be used as static model with a pre-scan
is also **naturally adaptive**
 - maintain frequency distribution (FD) instead of PDF

Context Modeling Example

a a a c c b a c a b a a a b c b c a a a

- **pure order-1 model** statistics after total scan

| context | Freq(a) | Freq(b) | Freq(c) |
|----------|------------------|------------------|------------------|
| a | 6/10 | 2/10 | 2/10 |
| b | 2/4 | 0/4 | 2/4 |
| c | 2/5 | 2/5 | 1/5 |

- **adaptive blended order-2 model**: use orders 2, 1, 0
 - assume initial default contexts
 - initialize order-2 model to be empty (no experience yet)
 - initialize order-1 model empty
 - order-0 model has every char equally likely

Context Model Example:

a a a c c

- assume default context of **aa**
 - initially no experience for all contexts
- **first a**: no o-2,1 experience, code using o-0
 - now have experience for context **aa**
 $\text{Freq}[\mathbf{aa}](\mathbf{a}) = 1/1$; also, $\text{Freq}\mathbf{a} = 1/1$
- **2nd & 3rd a**: each in context **aa**, code using o-2
 - now, $\text{Freq}[\mathbf{aa}](\mathbf{a}) = 3/3$
 - did not use o-1, so $\text{Freq}\mathbf{a} = 1/1$

Context Model Example:

a a a c c

- first **c**, context = **aa**
 - have o-2 experience for context, but not of **c**
so *escape* to o-1 context (**a**)
 - $\text{Freq}[\mathbf{a}](\mathbf{c}) = 0$, so *escape* to o-0
 - update \Rightarrow $\text{Freq}[\mathbf{aa}](\mathbf{a}) = 3/4$
 $\text{Freq}[\mathbf{aa}](\mathbf{c}) = 1/4$
 $\text{Freq}\mathbf{a} = \text{Freq}[\mathbf{a}](\mathbf{c}) = 1/2$
- 2nd **c**, context = **ac**
 - o-2 context empty, o-1 context **c** also empty
 - code using o-0, update FD's $\Rightarrow \text{Freq}[\mathbf{ac}](\mathbf{c}) = \text{Freq}\mathbf{c} = 1/1$

Context Modeling – PPMC Features

- **encoding** *escape* as a character symbol
frequency(*esc*) = number of times that *esc*
was invoked in that context
= number of different symbols that
were seen in that context
- **exclusion principle**
when escaping from order- k down to order $k - 1$
(because symbol σ was not found in order- k)
can **exclude** the possibilities of symbols
that were found in order- k

Context Model Example:

a a a c c

- contexts plus frequencies

| order-2 | order-1 | order-0 |
|--------------|-------------|---------|
| aa → a = 1 | a → a = 2 | a = 3 |
| aa → c = 1 | a → c = 1 | c = 2 |
| aa → esc = 2 | a → esc = 2 | esc = 2 |
| ac → c = 1 | c → c = 1 | |
| ac → esc = 1 | c → esc = 1 | |

- if next character is a then
 - no order-2 context for cc, so descend to order-1 context c
 - in o-1 context c, a does not occur
 - so escape (encoded using prob $\frac{1}{2}$) to o-0 context
 - in o-0 context, find a
 - exclude c (was in o-1 context)
 - encode a with prob $\frac{3}{5}$

Context Modeling

- **advantage** – very good compression, not only on text
- **disadvantages** – cumbersome (large and slow)
 - **Example:** for alphabet size = 64,
 - o-2 needs $64^2 = 4096$ contexts, each holding freqs of 64 el'ts if use 2 bytes (freq ct + codeword structure)
 - then o-2 needs 500 Kb, o-3 needs 32 Mb, higher orders worse
 - uses **probabilistic** codes
 - much slower than using fixed-length codes
 - because need to compute code for each new FD
- **can mitigate** space and time complexities
 - don't remember everything, conserve on:
 - order, number of contexts, detail of context PDF
 - use appropriate data structures

Practical Context Modeling

| | PPMC | SCM |
|-----------------|--------------------------------------|--|
| blend orders | 3, 2, 1, 0, -1 | 3, 1, 0 |
| data structure | trie | hash tables self-organizing lists |
| remembers | everything until memory exhausted | attempts to remember “most useful” info |
| memory use | 500 Kb | 100 Kb |
| execution speed | 2000 cps | 5000 cps |
| average CR | 31% | 35% |

PPMC = prediction by partial matching

SCM = streamlined context model

Context Modeling – Escape Probability

m = # of symbols coded so far

n = # of distinct symbols so far

f = # of symbol \mathbf{f} so far

t_1 = # of symbols that occurred exactly once

In **PPMC**,

$$p(\text{esc}) = n/(m + n)$$
$$p(\mathbf{f}) = f/(m + n)$$

⇒ new symbol increments denominator by 2, others by 1

PPMD reduces PPMC's overestimate of esc prob:

$$p(\text{esc}) = n/(2m)$$
$$p(\mathbf{f}) = (2f - 1)/(2m)$$

⇒ new symbol shares 2 units between f & n , others get 2

Another approach results in a slight underestimate:

$$p(\text{esc}) = (t_1 + 1)/(m + t_1 + 1)$$
$$p(\mathbf{f}) = f/(m + t_1 + 1)$$

Symbol Ranking

context predicts next letter

- avoid estimating probabilities
- avoid generating escape symbols to switch contexts
- use sliding window W as dictionary of contexts
- use list L of symbols in the alphabet (in MTF order)

Symbol Ranking

to encode symbol S

context = $C \leftarrow$ previous string of symbols

find longest match in W of $C \Rightarrow$ context order n

ranking index $r \leftarrow 0$

excluded set \leftarrow empty

while $n > 0$

from recent to older text in W

for each exact match of C_n

$P \leftarrow$ symbol immediately after the match

if P is excluded then continue

elseif $P = S$ then MTF(S) & return r

else increment r and exclude P

decrement n

for each P in list L that is not excluded

if $P \neq S$ then increment r

else move S to front of L and return r

Other Context Models

- word-based context models
- other types of context
 - word position
 - semantic information
 - using the future to predict the present
 - pre-scan to obtain total count
 - subtract past count to obtain present probability
- parts-of-speech as context in word-based system
- applications of context modeling
 - on top of Huffman →
 - context selects from among Huffman codes
 - on top of Arithmetic → improves on simple adaptive AC

Other Context Models

- most compression methods work in *streaming* mode
(process until EOF)
block mode: iterate processing a defined-size n of input (**block**)
- **Burrows-Wheeler Transform** (BWT), a.k.a. **block sorting**
 $S \leftarrow$ string of n symbols
if S has long runs of identical characters
then pre-filter by using RLE
permute S into L having a property
(symbols sorted by their 1-sided future contexts)
reverse transform requires extra info I
compress L, I (use MTF, RLE, Huffman/AC)
- **to recover S**
reverse order (Huffman, RLE, MTF) $\Rightarrow L, I$
reconstruct S from L, I

BWT

Determine permutation L of S , and auxiliary info I

- let $M = n \times n$ matrix, with:
(S rotated cyclically left i) in row i , $0 \leq i \leq n - 1$
- sort M lexicographically by rows

| | |
|---------------|--------------|
| possess | essposs |
| sposses | ossessp |
| ssposse | possess |
| essposs | sesspos |
| sesspos | sposses |
| ssesspo | ssesspo |
| ossessp | ssposse |
| <i>before</i> | <i>after</i> |

Note: every row and column is a permutation of S

- $L \leftarrow$ last column of M (spsssoe in example)
- $I \leftarrow$ row number of S in M (2 in the example)

BWT – Reconstruct S from L, I

- determine first column F of M
 - F and L are both permutations of S , and F is in sorted order
 - obtain F by sorting L
- construct auxiliary array T so that $T[i] = k \Rightarrow L[i] = F[k]$
this is done while sorting L

| F | L | i | T |
|---------|-----|-----|-----|
| essposs | | 0 | 3 |
| osessp | | 1 | 2 |
| possess | | 2 | 4 |
| sesspos | | 3 | 5 |
| sposses | | 4 | 6 |
| ssesspo | | 5 | 1 |
| ssposse | | 6 | 0 |

- reconstruct S from L, I , and T
 $S[n - 1 - i] \leftarrow L[T^{(i)}[I]]$, for $i = 0, 1, \dots, n - 1$
where $T^{(0)}[k] = k$, $T^{(i+1)}[k] = T[T^{(i)}[k]]$

BW – Reconstruction Example

| F | L | i | T |
|---------|-----|-----|-----|
| essposs | | 0 | 3 |
| ossessp | | 1 | 2 |
| possess | | 2 | 4 |
| sesspos | | 3 | 5 |
| sposses | | 4 | 6 |
| ssesspo | | 5 | 1 |
| ssposse | | 6 | 0 |

$$S[6 - 0] = L[T^{(0)}[2]] = L[2] = s$$

$$S[6 - 1] = L[T^{(1)}[2]] = L[T[2]] = L[4] = s$$

$$S[6 - 2] = L[T^{(2)}[2]] = L[T[4]] = L[6] = e$$

$$S[6 - 3] = L[T^{(3)}[2]] = L[T[6]] = L[0] = s$$

- why reconstruction works
 - $T[i] = k \rightarrow L[i] = F[k]$
so $F[T[i]] = L[i]$
 - $L[i]$ cyclically precedes $F[i]$ in string S
so $L[T[i]]$ cyclically precedes $F[T[i]] = L[i]$
 - start with last symbol and work backwards

BWT – Compressing L

- apply MTF to L (spsssoe in example)
init $A \leftarrow$ list of alphabet (eops in example)
for $i = 0$ **to** $n - 1$
 encode L_i as # of symbols preceding it in A
 move L_i to beginning of A

| L | A | code |
|-----|------|------|
| s | eops | 3 |
| p | seop | 3 |
| s | pseo | 1 |
| s | speo | 0 |
| s | speo | 0 |
| o | speo | 3 |
| e | ospe | 3 |

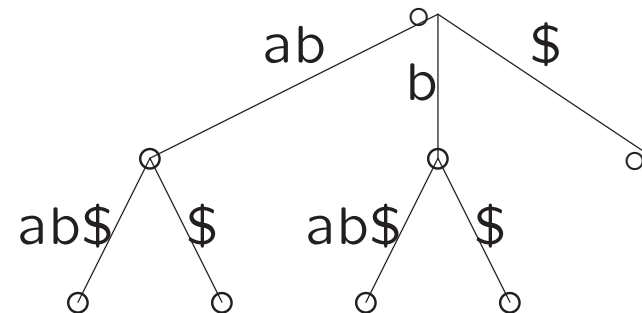
- the resulting sequence of codes are in range $[0, n-1]$
 (as they would be without MTF) but tend to have smaller values
- if the sequence is top-heavy with runs of 0's
 applying another RLE can improve compression
- use a Huffman code or arithmetic coding

BWT – Implementation

- choose large n (at least 4000, bigger is better)
- can calculate L, T using linear time and space
 - use **suffix tree** with edges stored in linked list
 - a depth-first traversal gives lexicographic order

BWT – Implementation

Example suffix tree for $x = abab$



- suffix tree (T) for string x (having length n)
 - stores $words = \{w\$ \mid w\$ \text{ is a suffix of } x\$ \}$
 - 1-to-1 map: $leafs(T) \Leftrightarrow \text{non-empty suffixes}$
 - $n + 1$ leaves $\rightarrow \leq n$ inner nodes $\rightarrow \leq 2n + 1$ nodes $\rightarrow \leq 2n$ edges
 - edges labeled by substr's of $x\$$ via 2 ptrs into $x\$$
so can represent T in $O(n)$ space

BWT - Implementation – Decoding

- pass 1: **find** $count[\sigma] = \#$ of occurrences of σ in x
 $offset[i] = \#$ of prior occurrences of y_i
for $\sigma \leftarrow 1$ **to** $|\Sigma|$ **do** $count[\sigma] \leftarrow 0$
for $i \leftarrow 0$ **to** $n - 1$ **do**
 $\sigma \leftarrow y_i$
 $offset[i] \leftarrow count[\sigma]$
 $count[\sigma]++$
- pass 2: **evaluate** $base[\sigma] = \sum_{\beta < \sigma} count[\beta]$
 $base[1] \leftarrow 0$
for $\sigma \leftarrow 2$ **to** $|\Sigma|$ **do**
 $base[\sigma] \leftarrow base[\sigma - 1] + count[\sigma - 1]$
- pass 3: **decode** x from right to left
 $r_i = k$ s.t. suffix at position i
 is rank k of all suffixes of $x\$$
 $r \leftarrow$ index provided by BWT encoding
 for $i \leftarrow n - 1$ **downto** 0 **do**
 $x_i \leftarrow y_r$
 $r \leftarrow base[x_i] + offset[r]$

BWT – Implementation – Improvements

- **index only characters that appear**
avoids providing for possibility of unseen characters
- **use modified MTF**
if next symbol has position z
then after coding change number by:
 - if $z = 1$ then shift z to position 0
 - if $z > 1$ then shift z to position 1
- **decay frequencies of MTF**
divide all frequencies in a context by 2
whenever largest frequency > 50

BWT – Limited Context Variant

- **stable sort symbols** by their $o-n$ contexts, $n \geq 2$
same result as BWT **except**, when equal $o-n$ context,
BWT resolves based on further context
variant resolves based on position in input
- **reconstruction using BWT almost good**
produces correct length $n + 1$ phrases
but sometimes not in the right places

BWT – Limited Context Variant

- **prepare T vector** via pass reconstructing n -contexts
s.t. $T[i] = k \Rightarrow i$ -th symbol has $o-n$ context rank k
where k is last rank used by that $o-n$ context
loop: prepend $L[i]$ to output
 $k \leftarrow T[i]$
 $T[i] \leftarrow T[i] - 1$
 $i \leftarrow k$
- **performance comparison**
 - much faster compression, slower decompression
 - almost equally effective (0-5% using $o-4$ context)
 - speed independent of block size