Transformations for Compressing Quality Scores of Next Generation Sequencing Data

Raymond Wan
r-wan@cb.k.u-tokyo.ac.jp

University of Tokyo and CBRC, AIST

August 2, 2011

FASTQ Data

Motivation for current research:

- Amount of next generation sequence data steadily increasing.
- Offers no new insight into the data; but focus on storing the data – somewhat of a hidden issue.
- Compression aims to improve disk storage + transmission costs.

Overview of work by others:

- Relatively less work done in this area so far.
- Most work to date is on DNA sequence compression (since 1999).
- In the last 1-2 years, some work on read compression:
  - Most describe software with pre-selected parameters.
  - Little explanation about what parameters worked and what did not.

FASTQ format

```
@SRR032209.3000 length=36
AATTAACATGTAAAGGCAGTTTGCTTCCATGCTACC
+SRR032209.3000 length=36
BA@?@@@@?@;A<9?>@>?(54423894+0+0038;
...
```

FASTQ format

```
→  @SRR032209.3000 length=36
   AATTAACATGTAAAGGCAGTTTGCTTCCATGCTACC
→  +SRR032209.3000 length=36
   BA@?@@@@?@;A<9?>@>?(54423894+0+0038;
   ...
```

FASTQ format

```
    @SRR032209.3000 length=36
→   AATTAACATGTAAAGGCAGTTTGCTTCCATGCTACC
    +SRR032209.3000 length=36
    BA@?@@@@?@;A<9?>@>?(54423894+0+0038;
    ...
```

FASTQ format

```
@SRR032209.3000 length=36
AATTAACATGTAAAGGCAGTTTGCTTCCATGCTACC
+SRR032209.3000 length=36
BA@?@@@@?@;A<9?>@>?(54423894+0+0038;
...
```

FASTQ quality scores range from "!" to "~", or ASCII code 33 to 126 (the range of printable ASCII characters).

FASTQ format

```
@SRR032209.3000 length=36
AATTAACATGTAAAGGCAGTTTGCTTCCATGCTACC
+SRR032209.3000 length=36
BA@?@@@@?@;A<9?>@>?(54423894+0+0038;
...
```

FASTQ quality scores range from "!" to "~", or ASCII code 33 to 126 (the range of printable ASCII characters).

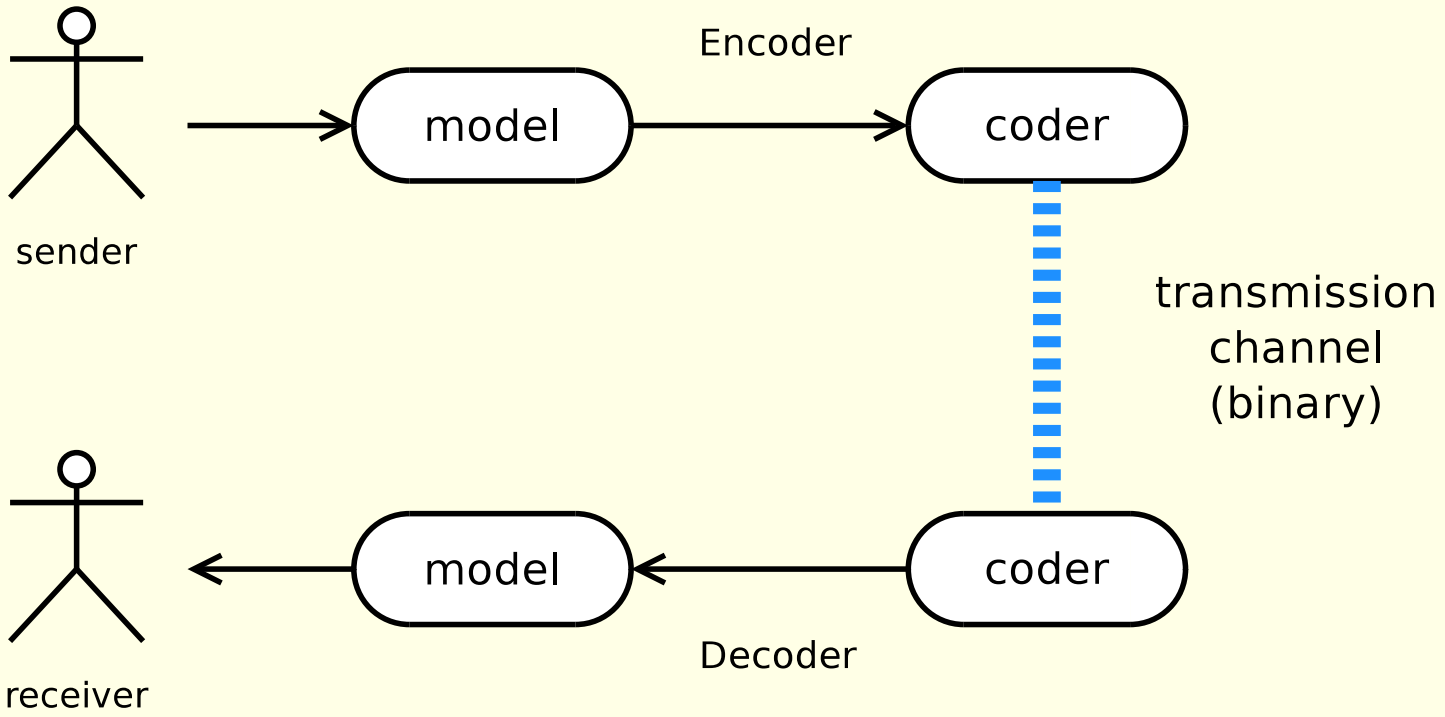
e.g.	"!"	1.000
	"+"	0.100
	"5"	0.010
	"?"	0.001

The number of possible characters is 94.

Reasons for compressing quality scores:

- Perhaps the least important, but also has attracted the least attraction (so far).
- Systems are being developed to make use of quality scores – mapping, assembly, and read trimming.

Compression Background



Model:

- What if we remove unimportant information from the data?
  - ◆
- What if we use "surrounding" data to create a model?
  - ◆
- What if the data is similar to data that we just saw?
  - ◆

Model:

- What if we remove unimportant information from the data?
  - MP3 compression for music
- What if we use "surrounding" data to create a model?
  - Image compression; also fascimile transmission
- What if the data is similar to data that we just saw?
  - `gzip` system

$\rightarrow$ We select/develop a model that depends on some prior knowledge of the data.

Humans communicate in characters (Latin alphabet, kanji, hiragana, etc.) but computers use a binary alphabet $\{0, 1\}$.

Storing or transmitting data means that a conversion to binary digits (bits) is required.

That is the coder's job using the output of the model.

ASCII maps the Latin alphabet, punctuation, and numbers to bits.

i.e., "I" $\rightarrow$ ASCII code 73 $\rightarrow$ 01001001

This code assumes all 256 characters in ASCII are equally-probable.

- What if we encode a number x in two parts – (1) the largest logarithmically-sized bin and then (2) position in the bin?
  - ◆
- What if the bins are equal in size and determined by some parameter?
  - ◆
- What if we give short codes to frequently occurring symbols?
  - ◆

Coder

- What if we encode a number x in two parts – (1) the largest logarithmically-sized bin and then (2) position in the bin?
  - Gamma code
- What if the bins are equal in size and determined by some parameter?
  - Golomb code
- What if we give short codes to frequently occurring symbols?
  - Huffman coding

$\rightarrow$ Ideally, select a coder that fits the output of the model.

Example: Gamma Code

Create logarithmically-sized bins (base-2): 1, 2, 4, 8, 16, 32, 64, ...

x	Bin	Position in bin	Length
1	0		1
2	10	0	3
3	10	1	3
4	110	00	5
5	110	01	5
7	110	11	5
8	1110	000	7
...			
15	1110	111	7
16	11110	0000	9
...			
31	11110	1111	9
32	111110	00000	11

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	`woodchuck␣chuck␣if`

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	`woodchuck␣chuck␣if`
(b)	$\langle 0, 0, \mathrm{o} \rangle$	`woodchuck␣chuck␣if`

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	`woodchuck␣chuck␣if`
(b)	$\langle 0, 0, \mathrm{o} \rangle$	`woodchuck␣chuck␣if`
(c)	$\langle 1, 1, \mathrm{d} \rangle$	`woodchuck␣chuck␣if`

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	`woodchuck␣chuck␣if`
(b)	$\langle 0, 0, \mathrm{o} \rangle$	`woodchuck␣chuck␣if`
(c)	$\langle 1, 1, \mathrm{d} \rangle$	`wood chuck␣chuck␣if`
(d)	$\langle 0, 0, \mathrm{c} \rangle$	`wood chuck␣chuck␣if`

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \mathrm{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \mathrm{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \mathrm{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \mathrm{h} \rangle$	woodchuck␣chuck␣if

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \mathrm{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \mathrm{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \mathrm{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \mathrm{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \mathrm{u} \rangle$	woodchuck␣chuck␣if

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \mathrm{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \mathrm{d} \rangle$	wood chuck␣chuck␣if
(d)	$\langle 0, 0, \mathrm{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \mathrm{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \mathrm{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \mathrm{k} \rangle$	woodchuck␣chuck␣if

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \mathrm{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \mathrm{d} \rangle$	wood chuck␣chuck␣if
(d)	$\langle 0, 0, \mathrm{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \mathrm{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \mathrm{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \mathrm{k} \rangle$	woodchuck␣chuck␣if
(h)	$\langle 0, 0, \mathrm{␣} \rangle$	woodchuck␣chuck␣if

`gzip` uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \mathrm{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \mathrm{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \mathrm{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \mathrm{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \mathrm{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \mathrm{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \mathrm{k} \rangle$	woodchuck␣chuck␣if
(h)	$\langle 0, 0, \mathrm{␣} \rangle$	woodchuck␣chuck␣if
(i)	$\langle 6, 6, \mathrm{i} \rangle$	woodchuck␣chuck␣if

gzip uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \text{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \text{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \text{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \text{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \text{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \text{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \text{k} \rangle$	woodchuck␣chuck␣if
(h)	$\langle 0, 0, \text{␣} \rangle$	woodchuck␣chuck␣if
(i)	$\langle 6, 6, \text{i} \rangle$	woodchuck␣chuck␣if
(j)	$\langle 0, 0, \text{f} \rangle$	woodchuck␣chuck␣if

gzip uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \text{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \text{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \text{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \text{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \text{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \text{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \text{k} \rangle$	woodchuck␣chuck␣if
(h)	$\langle 0, 0, \text{␣} \rangle$	woodchuck␣chuck␣if
(i)	$\langle 6, 6, \text{i} \rangle$	woodchuck␣chuck␣if
(j)	$\langle 0, 0, \text{f} \rangle$	woodchuck␣chuck␣if

Also called the "sliding-window" algorithm.

Source: Wan [2003, pg. 21]

gzip uses the LZ77 algorithm to represent new data as a back-pointer into a window of recently-seen data.

	Tuple	Text
(a)	$\langle 0, 0, \text{w} \rangle$	woodchuck␣chuck␣if
(b)	$\langle 0, 0, \text{o} \rangle$	woodchuck␣chuck␣if
(c)	$\langle 1, 1, \text{d} \rangle$	woodchuck␣chuck␣if
(d)	$\langle 0, 0, \text{c} \rangle$	woodchuck␣chuck␣if
(e)	$\langle 0, 0, \text{h} \rangle$	woodchuck␣chuck␣if
(f)	$\langle 0, 0, \text{u} \rangle$	woodchuck␣chuck␣if
(g)	$\langle 3, 1, \text{k} \rangle$	woodchuck␣chuck␣if
(h)	$\langle 0, 0, \text{␣} \rangle$	woodchuck␣chuck␣if
(i)	$\langle 6, 6, \text{i} \rangle$	woodchuck␣chuck␣if
(j)	$\langle 0, 0, \text{f} \rangle$	woodchuck␣chuck␣if

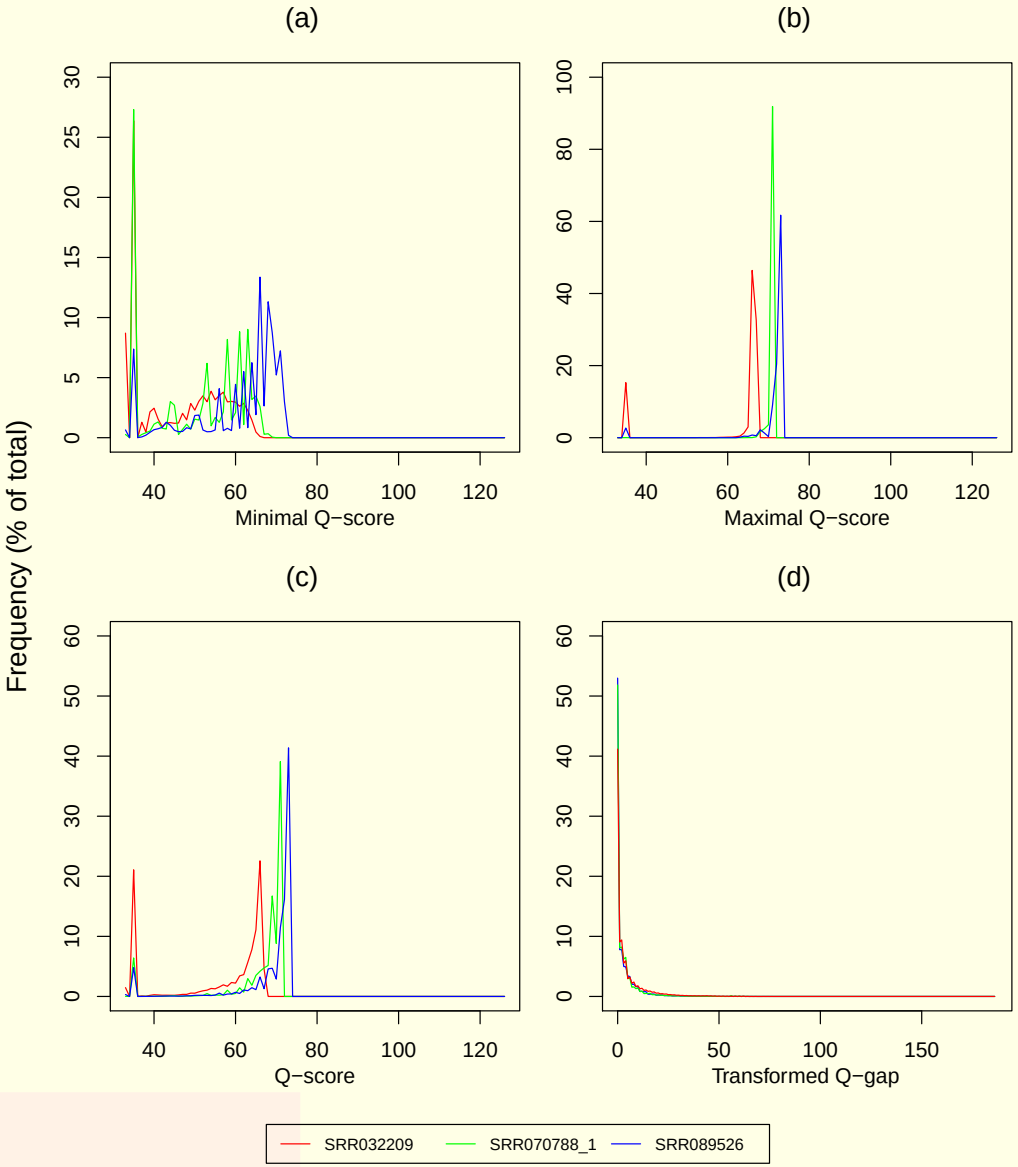
Also called the "sliding-window" algorithm.

Source: Wan [2003, pg. 21]

Quality Scores Compression

Our first step is to examine what real data looks like – something that others have already done:

Accession	Species	# reads ($\times 10^6$)	Length	Size (MiB)
SRR032209	*Mus musculus*	18.8	36	662.8
SRR070788_1	*Homo sapiens*	24.8	100	2,393.2
SRR089526	*H. sapiens*	23.9	48	1,115.7



Findings:

- About half the range of possible quality scores is unused in real data.
- Non-uniform distribution of quality scores.
- The absolute differences between adjacent quality scores is close to 0.

ReadBlocking Create blocks of reads so that each block is processed independently.

MinShifting Shift all of the quality scores in the block so that the smallest value is 1.

FreqOrdering Reorder quality scores by frequency so that the most frequent ones have the smallest value of 1.

GapTranslating Convert quality scores in a block into differences.

Example: ADBCCCEE $\rightarrow \{65, 68, 66, 67, 67, 67, 69, 69\}$

MinShifting $\{1, 4, 2, 3, 3, 3, 5, 5\}$

FreqOrdering $\{3, 5, 4, 1, 1, 1, 2, 2\}$

GapTranslating $\{65, 3, -2, 1, 0, 0, 2, 0\}$
$\rightarrow \{65, 5, 4, 1, 0, 0, 3, 0\}$
$\rightarrow \{65, 6, 5, 2, 1, 1, 4, 1\}$

The aim is to change the distribution of symbols so that small (integer values) appear more often.

Truncating Truncate the range of quality scores so that values greater than x are changed to x.

LogBinning PHRED scores places quality scores in logarithmically-sized bins. Continue this practice but use only some of the bins. So, with a parameter of 3, we have: $\{33, 34, 35, 36, 37, 38, 39, \cdots\} \rightarrow \{33, 33, 33, 36, 36, 36, 39, \cdots\}$.

UniBinning Uniformly re-bin quality scores so that each probability range maps to a bin. So, with a parameter of 5, the first bin has the probabilities $[0, 0.2)$. The next one is $[0.2, 0.4)$, and so on.

We unify the parameters to the new alphabet size for easier comparision.

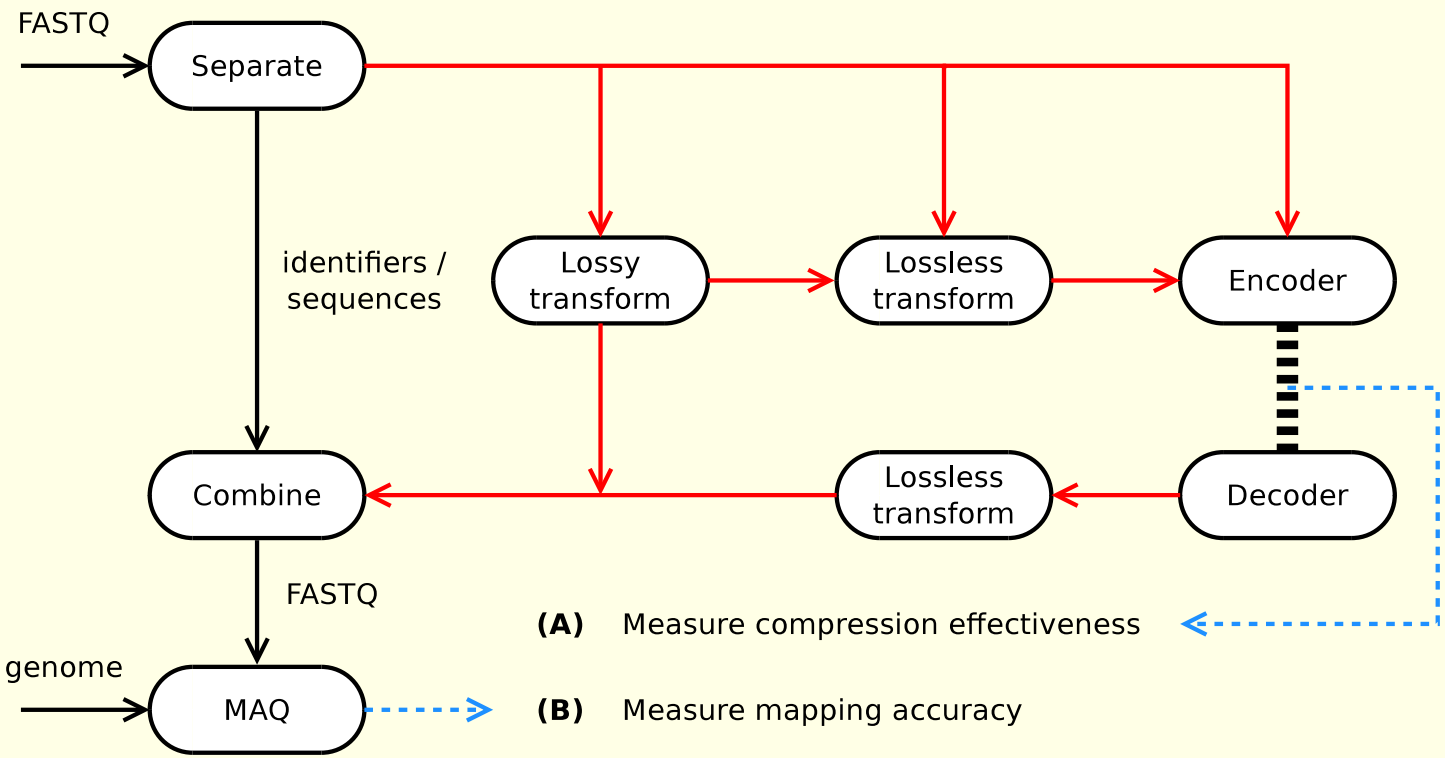
Static codes Encodes a value x the same way regardless of neighboring symbols.

Parameterized codes Encodes a value x after examining the entire block to determine overall statistics.

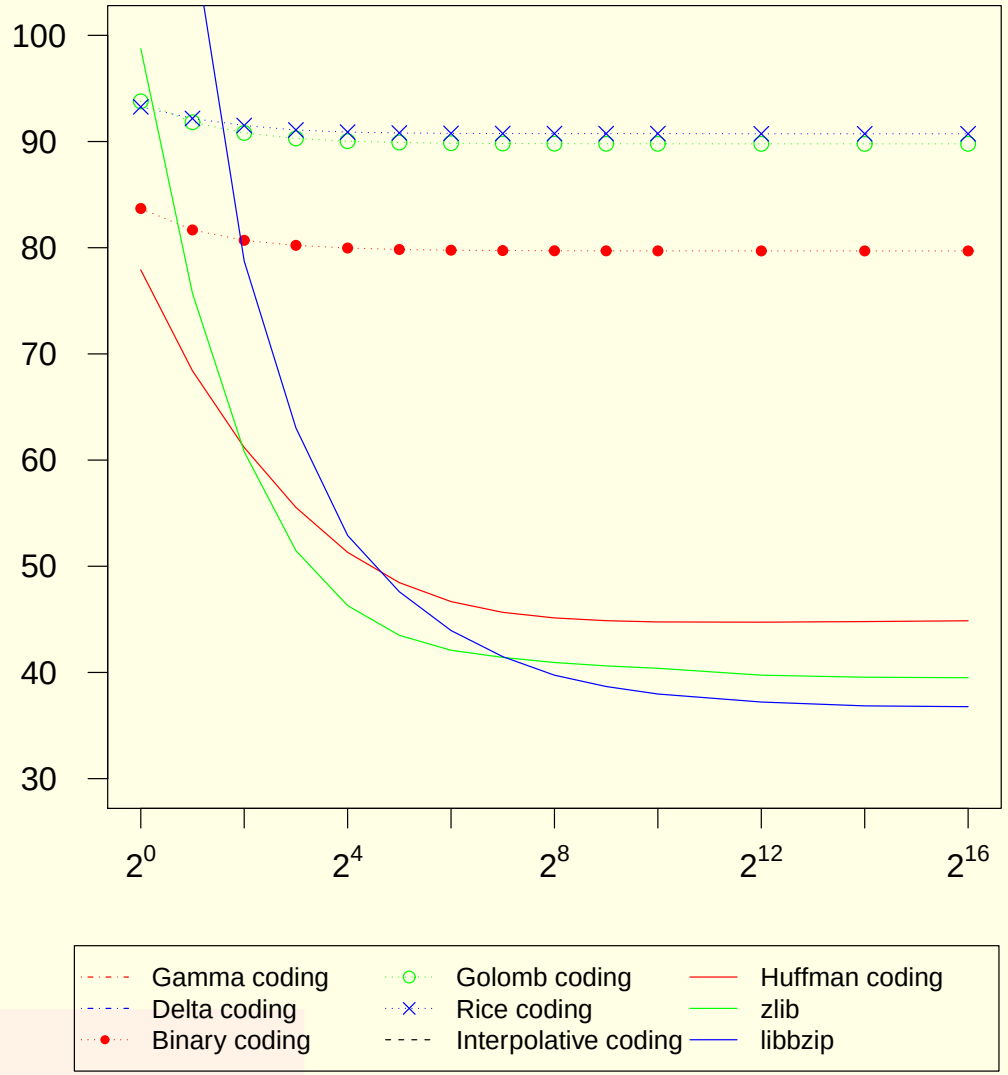
Entropy codes Encodes the values in a block close to their "zero-order" self-information. So, frequent symbols get short codes.

Compression systems Employ commonly-used systems that already combine a model and a coder.

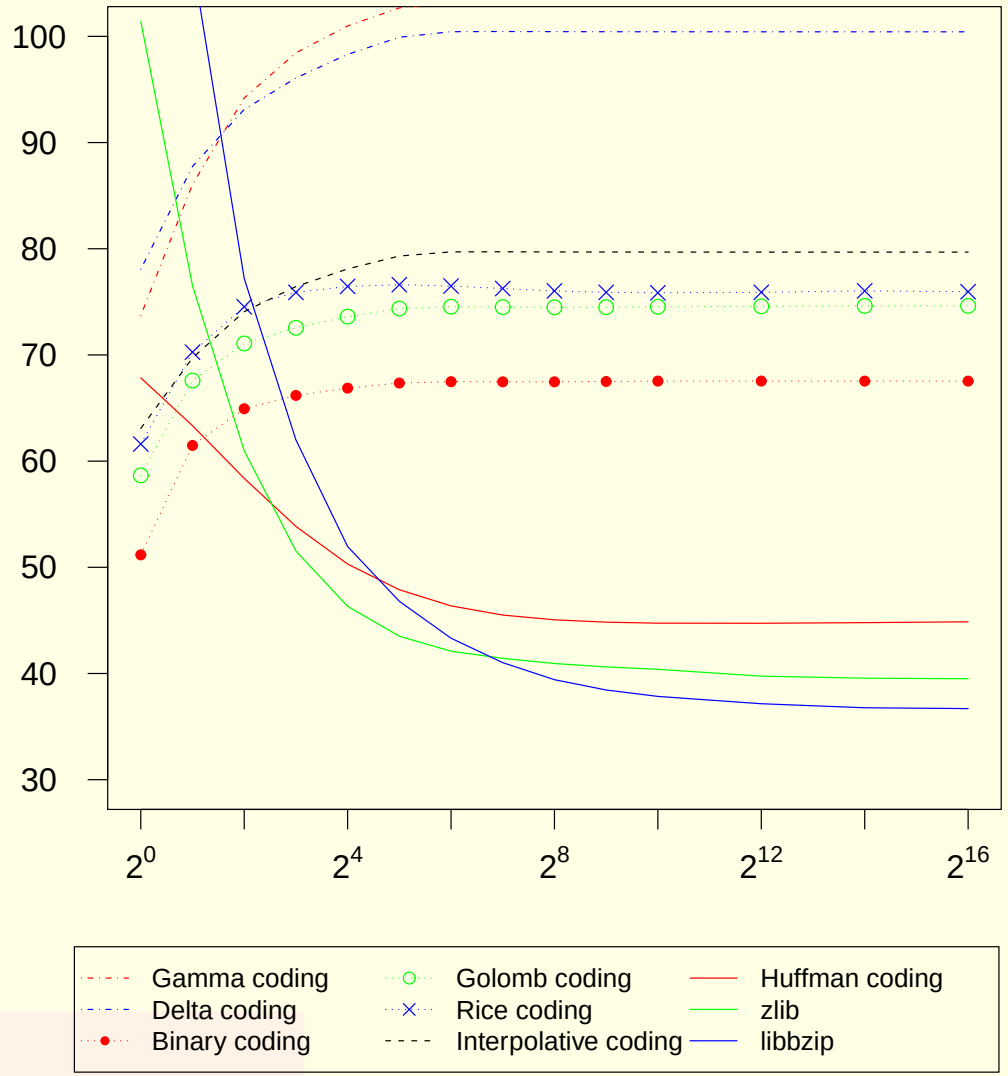
Results



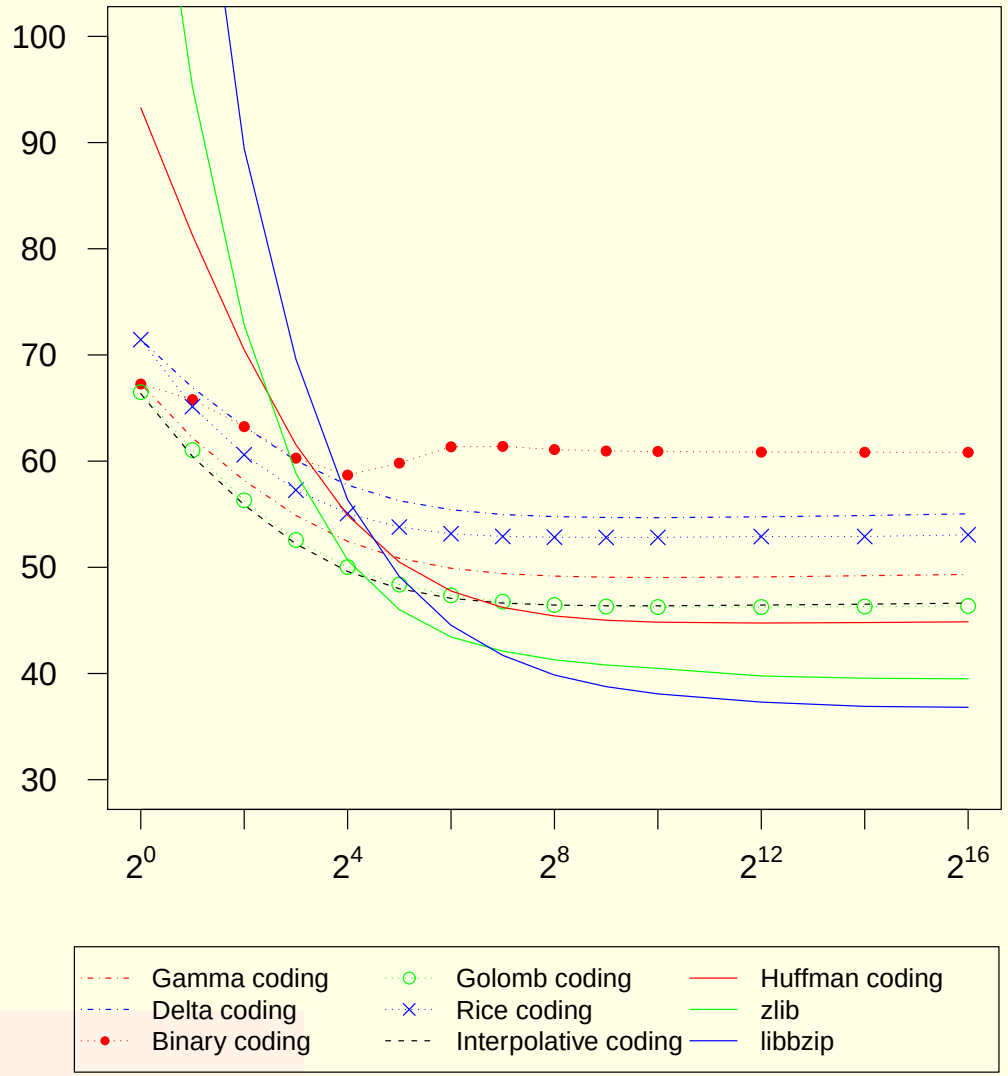
No Transformation:



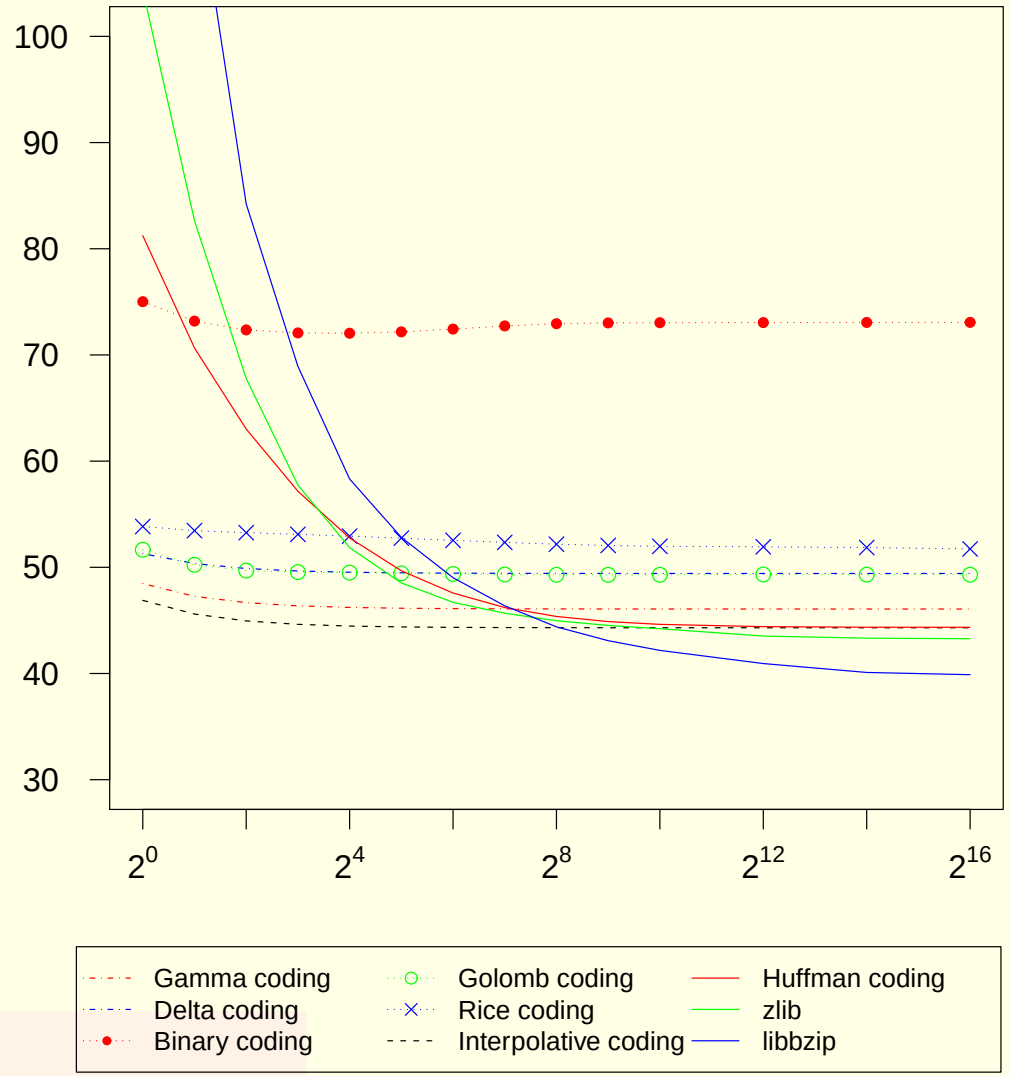
MinShifting:

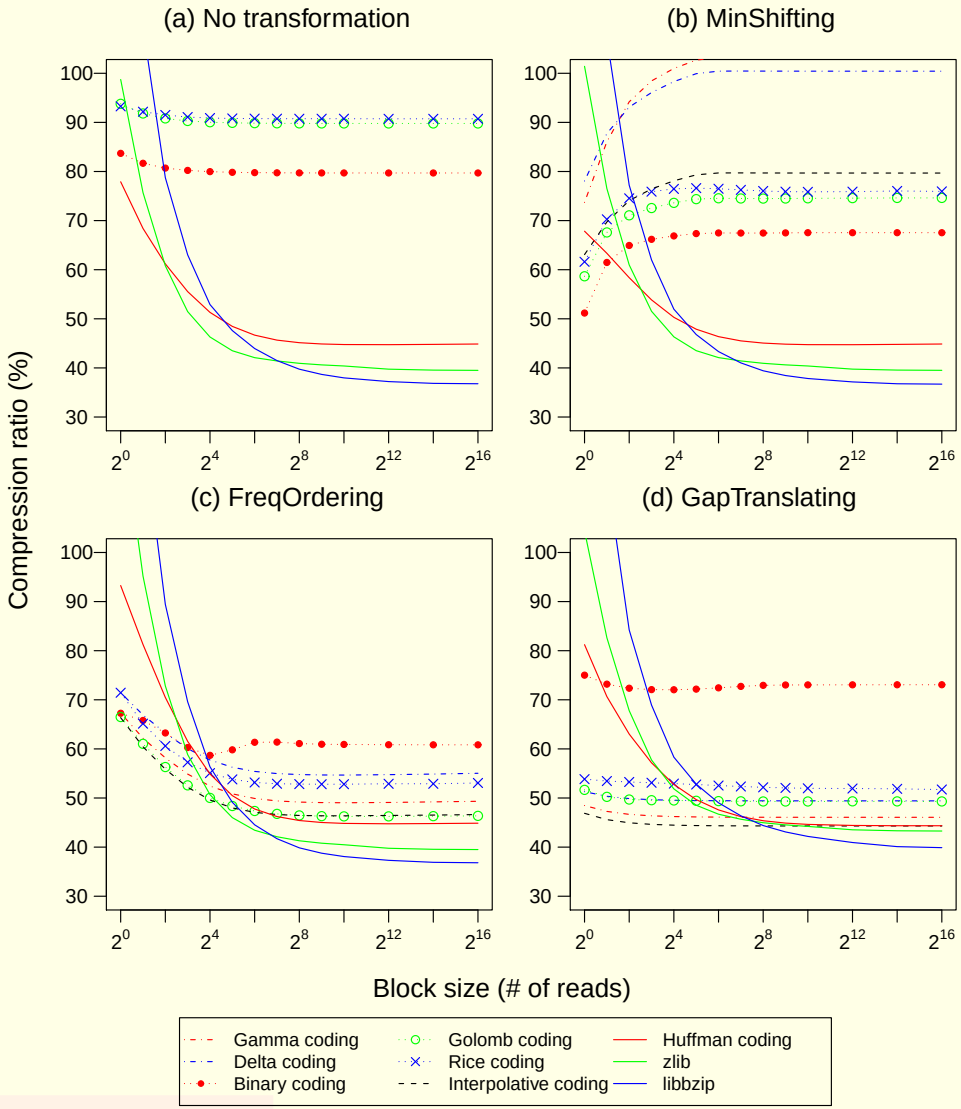


FreqOrdering:

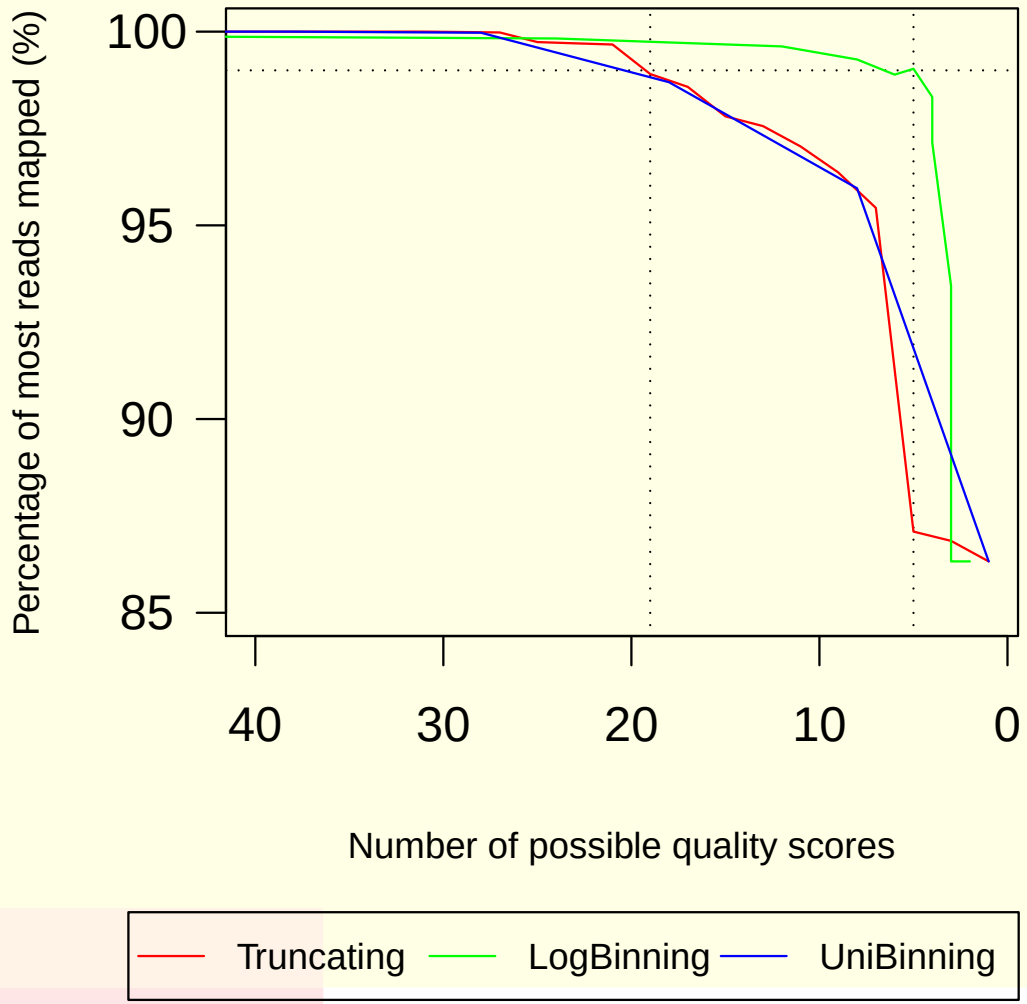


GapTranslating:

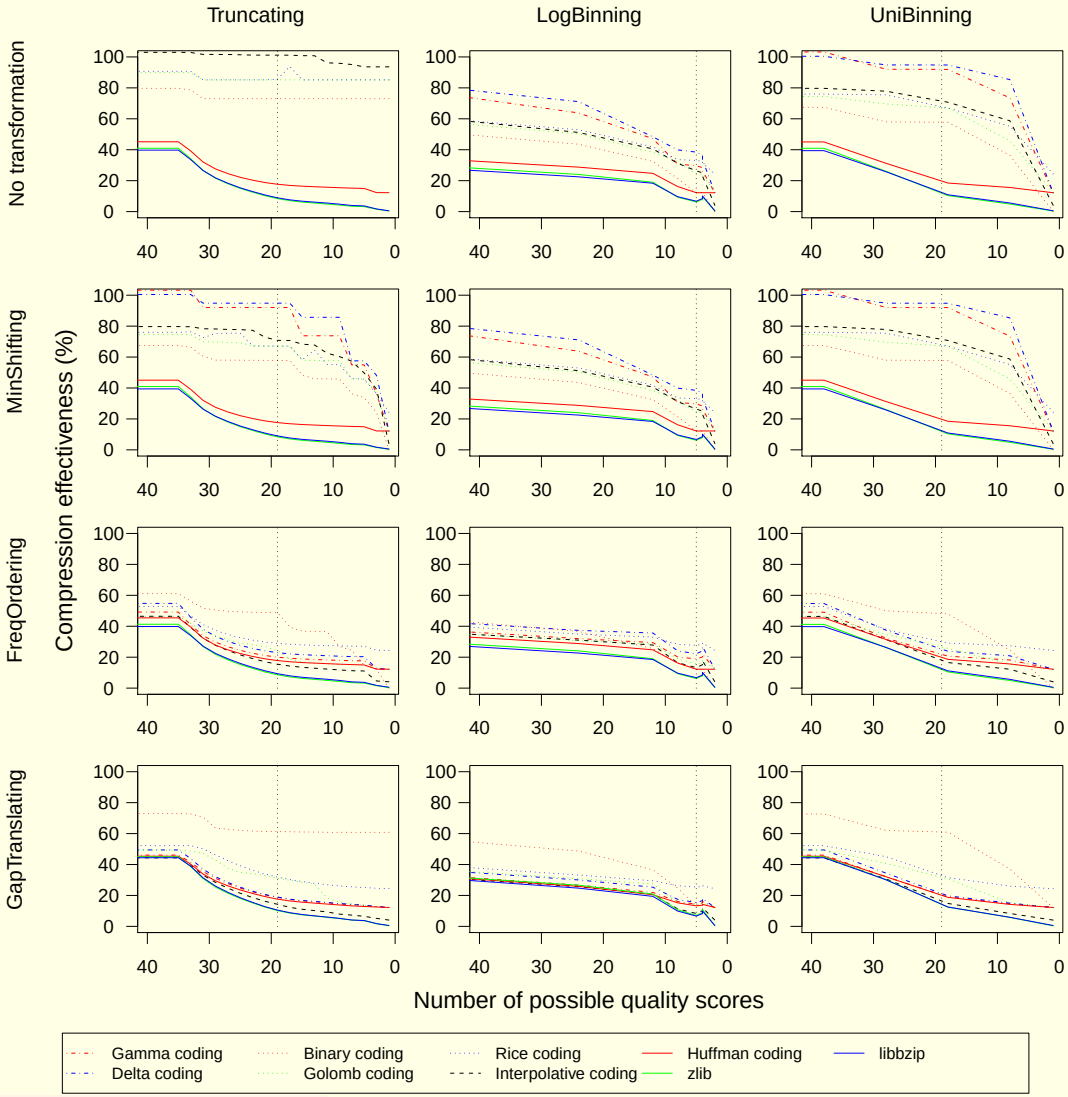




Of the 18.8×10^6 reads, 12.1×10^6 were unambigiously mapped (about 64.5%).



Bringing it All Together



Conclusion

Advantages

- Simple coding schemes perform as well as more complicated compression systems.
- To attain this performance, lossless transformations are necessary.
- Small blocksizes reduce memory usage (the amount of data in memory), minimizes the effects from data corruption, and allows random access.

- Their compression time is less than `zlib` and `libbzip2`.
- Compression ratios and compression/uncompression time of these "non-compression systems" for FreqOrdering and GapTranslating are unaffected by block size, essentially removing one parameter.

- `zlib` and `libbzip2` is well-known to general users; the other methods are more popular in the index compression field.
- `zlib` still outperforms all other methods, if a large block size is chosen.

- `zlib` and `libbzip2` is fast at decoding. This is partly due to their underlying methods, but due to their age, we expect them to be well-tuned for performance.

The most likely questions in your head...

The most likely questions in your head...

1. "Huh???"

The most likely questions in your head...

1. "Huh???"
2. "Yes, I kind of see – but there is already BAM...what is the point of this work?"

"Yes, I see – but there is already BAM...what is the point of this?"

- SAM/BAM stores mapped reads, including location information, "control information" to indicate changes to the sequence, etc.
- Can be used to store FASTQ data, but some overhead is added on top.
- BAM uses an "algorithm" called BGZF – essentially variable-length gzipped blocks.

So, our comparative results is roughly equivalent to BGZF, and thus BAM. Of course, our `gzip` results are not fully equivalent because of the above differences.

Collaborators

- Dr. Vo Ngoc Anh [University of Melbourne]
- Prof. Kiyoshi Asai [University of Tokyo and CBRC, AIST]

Appendix

References

R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, University of Melbourne, Australia, December 2003