BSc thesis

Batchelor's Programme in Computer Science

something about variable-byte encoding

Jussi Timonen

February 8, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. D.U. Mind, Dr. O. Why

Examiner(s)

Prof. D.U. Mind, Dr. O. Why

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Batchelor's Programme in Computer Science

Tekijä — Författare — Author

Jussi Timonen

Työn nimi — Arbetets titel — Title

something about variable-byte encoding

Ohjaajat — Handledare — Supervisors

Prof. D.U. Mind, Dr. O. Why

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
BSc thesis	February 8, 2020	15 pages, 17 appendice pages

Tiivistelmä — Referat — Abstract

Use this otherlanguage environment to write your abstract in another language if needed.

Topics are classified according to the ACM Computing Classification System (CCS), see https://www.acm.org/about-acm/class: check command `\classification{}`. A small set of paths (1–3) should be used, starting from any top nodes referred to bu the root term CCS leading to the leaf nodes. The elements in the path are separated by right arrow, and emphasis of each element individually can be indicated by the use of bold face for high importance or italics for intermediate level. The combination of individual boldface terms may give the reader additional insight.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

ulkoasu, tiivistelmä, lähdeluettelo

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms specialisation line

Contents

1 Introduction

Enormous datasets are a common case in today's applications. Compressing the datasets is beneficial, because they naturally decrease memory requirements but also are faster when compressed data is read from disk (Zobel and Moffat, 1995).

One of the leading methods of data compression is variable-length coding (Salomon, 2007), where frequent sequences of data are represented with shorter codewords. Because the sequences of data have different lenghts when compressed, it is not trivial to determine the exact location of a certain element. If this is required, the usual data compression algorithms are inefficient. Fortunately this is not a requirement compression algorithms usually need to fulfill.

However, random access of compressed data is needed in compressed data structures. In most compression methods, the only way to this is to decompress data from the beginning. An integer compressing method with fast random access is explained later and compared existing state-of-the-art methods.

2 Variable-byte encoding

Variable-byte (VB) encoding (Williams and Zobel, 1999) is a method of compressing integers via omitting leading zero bits. In normal data sets, it loses in compression performance to generic algorithms like Huffman encoding or Lempel-Ziv encoding, but in comparison it's faster to decode (Brisaboa et al., 2009) and allows constant time random access. This is later explained in (TODO: link to later chapter).

A good data set for VB encoding is a list of mostly small numbers with a need to support larger ones. A search engine may use an inverted index of words in documents. For each word, a list of document IDs where the word appears is stored. It may also store locations of the word in document for advanced search purposes. Usually these lists are preprocessed and stored as gaps, storing the difference to previous number instead of the actual number (Manning et al., 2008). Common words have a lot of entries in these lists, but because of gap storing the numbers are small. In contrast, rare words have only a few entries but the numbers stored are larger. These lists are excellent data sets for VB encoding.

VB encoding splits each integer into blocks of b bits and adds a continuation bit to the block to form chunks of length $1+b$. The extra bit is set to 1 only on the block containing the least significant bits of the integer. This information is used in decoding to signal if next chunk continues the current integer. For example, let's assume $b = 4$ and let $n = 42$ be a 16 bit unsigned integer. The standard 8-bit representation of n is `00000000 00101000`. When split to blocks of b bits, it becomes `0000 0000 0010 1000`. Empty blocks are omitted and continuation bits are added to the remaining blocks. The result is `00010 11000`, which is the compressed data.

```
function VBEncodeNumber(n)
    bytes ← list
    while true do
        bytes.prepend(n mod 128)
        if n < 128 then
            break
        n ← n div 128
    bytes.last() += 128
    return bytes
```

```
function VBEncode(numbers)
    bytestream ← list
    for each n ∈ numbers do
        bytes ← VBEncodeNumber(n)
        bytestream.extend(bytes)
    return bytestream
```

Figure 2.1: VByte encoding

Decoding is essentially just reversing the encoding steps: chunks are read until a chunk with 1 as continuation bit is found. Continuation bits are removed from all the chunks and the blocks are concatenated to form the original number. As in the previous example, $b = 4$, encoded message is `00010 11000` and $n = 0$. Block from the first chunk is extracted and added to n, making $n =$`10`. A bitwise shift to left equal to b is applied to n, changing $n =$`100000`. Then the block is extracted from the next chunk. This block is added n, making $n = 42$. Because the previous continuous bit was 1, decoding this number has finished. An example implementation of encode and decode with block length of 7 is shown in Figure 2.1 and Figure 2.2.

```
function VBDecode(bytestream)
    numbers ← list
    n ← 0
    for each b ∈ bytestream do
        if b < 128 then
            n ← 128 × n + b
        else
            n ← 128 × n + b - 128
            numbers.append(n)
            n ← 0
    return numbers
```

Figure 2.2: VByte decoding

Small lengths of c can yield better compression rate at the cost of more bit manipulation, while longer chunks need less bit manipulation and offer less effective compression. Generally block length of 7 has been used because it gives a good average and handling chunks as bytes is convenient (Manning et al., 2008).

VB encoding is a well known compression algorithm. It's origins date back to 1980's and the famous MIDI music file format. It stored some of the numbers in a "variable-length quantity" form, which was a 7-bit block VB structure (Association, 1996). Similar data types have existed for example in Apache Lucene (as vInt) and IBM DB2 database (as Variable Byte). Later, VB encoding was found efficient in compressing integer lists. It was first used to compress lists of word locations in documents (Scholer et al., 2002). It yielded excellent records, and since then many different approaches have been introduced.

history and current implementations etc here - mainitse IR book, Lem15, Pla18

TODO: a chapter about existing work

3 Directly addressable codes?

Rank and select are two functions that work on bit arrays. Rank(i) gives the sum of 1 bits from the beginning of the bit array and select(i) gives the index of ith 1 bit in the bit array. Both functions work in constant time (citation?) and they require only a few percents of extra space over the data. The extra bits c added by variable-byte encoding conveniently create a bit array, where 1's represent the endings of numbers. An effective version of random access has already been introduced (Brisaboa et al., 2009).

Random access with select query is also possible. By separating the c bit array and b block array, b contains variable-byte integers in readable form. Another upside is that functions next(i) and previous(i) are conveniently available. Rank implementation has

- explain how random access is good

4 Previous Work

bl

5 Algorithm

- modifications to basic implementation

6 Results

- comparison to basic implementation + Bri09

Table 6.1: Results with 100k entries (in milliseconds).

Experiment	128	256	32768	65536	2^{24}	$2^{32}-1$
$7bitVByteencoding$	34.97	49	53.04	52.18	53.08	76.21
$8bitVByteencoding$	33.57	32.47	42.96	43.11	46.15	65.14
$7bitVByteencodingwitharray$	33.39	46.85	51.24	49.03	48.93	66.84
$8bitVByteencodingwitharray$	32.52	31.88	41.54	39.94	41.15	52.86

Table 6.2: Results with 1M entries (in milliseconds).

Experiment	128	256	32768	65536	2^{24}	$2^{32}-1$
$7bitVByteencoding$	38.17	55.09	64.38	65.36	68.08	159
$8bitVByteencoding$	37.09	37.75	53.44	54.6	59.32	148.7
$7bitVByteencodingwitharray$	38.09	55.42	62.22	61.25	71.72	135.01
$8bitVByteencodingwitharray$	36.13	36.83	50.58	50.73	56.93	103.18

7 Conclusion

- here

8 Future work

- something to improve / research?

8.1 Figures

Figure gives an example how to add figures to the document. Remember always to cite the figure in the main text.

8.2 Tables

Table gives an example how to report experimental results. Remember always to cite the table in the main text.

9 From tex to pdf

In Linux, run `pdflatex filename.tex` and `bibtex filename.tex` repeatedly until no more warnings are shown. This process can be automised using make-command.

10 Conclusions

It is good to conclude with a summary of findings. You can also use separate chapter for discussion and future work. These details you can negotiate with your supervisor.

Bibliography

Association, T. M. M. (1996). “The Complete MIDI 1.0 Detailed Specification”. In:

Brisaboa, N. R., Ladra, S., and Navarro, G. (2009). “Directly Addressable Variable-Length Codes”. In:

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*.

Salomon, D. (2007). *Variable-length codes for data compression*. Springer, Heidelberg.

Scholer, F., Williams, H. E., Yiannis, J., and Zobel, J. (2002). “Compression of Inverted Indexes For Fast Query Evaluation”. In:

Williams, H. E. and Zobel, J. (1999). “Compressing Integers for Fast File Access”. In: *COMPJ: The Computer Journal* 42.3, pp. 193–201.

Zobel, J. and Moffat, A. (1995). “Adding Compression to a Full-text Retrieval System”. In: *Software-practice and experience* 25.8, pp. 891–903.