MSc thesis

Computer Science

something about variable-byte encoding

Jussi Timonen

February 13, 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	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
MSc thesis	February 13, 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 **1**

2 Variable-byte encoding **2**

3 Directly addressable codes? **5**

3.1 DAC via rank 6

4 DAC with select query **7**

5 Experimental results **9**

6 Conclusion **11**

7 Future work **12**

7.1 Figures 12

7.2 Tables 12

8 Conclusions **13**

Bibliography **15**

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 sometimes preferred due to it's simplicity. 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 an inverted index or 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 experimented in compressing inverted index lists of word locations in documents (Scholer et al., 2002). That yielded excellent records, and since then many different approaches have been introduced.

Modern studies have looked into machine code for VB and applied SIMD (Single instruction, multiple data) architecture to VB (Lemire et al., 2018; Plaisance et al., 2015). The bit operations in VB are simple and therefore modifying the code to use SIMD instructions is simple and the speed improvements are significant.

TODO: might need 1-2 more chapters

3 Directly addressable codes?

Ability to handle large amounts of data fast is one of the key challenges in the field of search engines. Compressed data structures are applied to fit the data into cache, memory or even hard drive. Direct access to any element is one of the usual requirements in compressed data strutures. It is not natively possible to decode the i-th element in variable-byte compression algorithms, because the position of the element depends on the preceding data. Direct access is achievable with supporting data structures. An obvious solution is to store each element's location, but that adds a very large overhead which removes the benefit of compression, but for some compression techniques it adds a very large overhead and thus is not applicable.

Rank and select are two succinct data structure operations which operate on a bit array B. $Rank_1$(B, i) gives the sum of 1 bits between B[0] and B[i] and $select_1$(b, i) gives the index of i-th 1 bit in B. Both operations work in constant time (citation?) and they require only a few percents of extra space over the bit array B. See Table 3.1 for *rank* and *select* size requirement approximations on different sizes of bit arrays.

Locations of 1 bits in B should represent locations of the items in the encoded data. For most compression algorithms, this requires B to be created in addition to the existing data and the length of B usually has to be close to the length of the data. VB encoding has several advantages with search and rank: it's data is compressed in blocks of equal length which significantly shrinks the length of B. Also the bit array C formed from the continuation bits already stores the locations of items. In this case, $rank_1$(C, i) would give the sum of end bytes up to i-th index and $select_1$(C, i) would give the location of the ending byte of i-th compressed element.

Table 3.1: Size of support structure with SDSL-lite library

Structure	TODO	get	multiple	size	calculations	here
rank	34.97	49	53.04	52.18	53.08	76.21
select	33.57	32.47	42.96	43.11	46.15	65.14

3.1 DAC via rank

Directly addressable codes in VB was first introduced by (Brisaboa et al., 2009) in 2009. Their solution was to divide chunks in encoded bytes to separate arrays A_1 to A_n and then use $rank_1$ to get direct access. For example, if an element needed 3 chunks when encoded, the least significant bits would go to A_1, second least significant bits to A_2 and the most significant bits would go to A_3. When fetching i-th element, getting the first chunk is just fetching A_1[i]. If the continuation bit is set at 0 (meaning there are more chunks to this element), getting the correct index for A_2 is calculated with $rank_1(A_1$, i). Small values benefit from this reordering, because the first chunk does not need any calculation. This method causes a small overhead on the data, because each continuation bit array needs a support data structure for rank.

do rank/select algorithms + data structures need to be explained better? (or just referenced to)

¡introduce Bri09 better?¿

¡example of Bri09¿

- explain how random access is good ?

4 DAC with select query

Using $select_1$ on the continuation bit array to achieve direct access is more intuitive than using $rank_1$. The element locations are already marked with 1's in B and a single $select_1$ query gets the desired starting point, while the forementioned version (Brisaboa et al., 2009) used one *rank* query for each chunk beyond the first. Minimizing the amount of *select* and *rank* queries is important. They run in constant time but their impact is huge, because rest of the VB decoding consists of a few bit operations.

To use $select_1$ with VB, continuation bits need to be separated from chunks to their own bit array and a $select_1$ structure built over it. *Select*(i) returns the location of the end byte of i-th element. Therefore the start of j-th element in the block array is at block b_s = $select_1$(j-1) + 1. To simplify calculations even further, block sizes of 2, 4 and 8 were used to avoid splitting of blocks between bytes.

The blocks should be stored in an order that, depending on the endianness of the system, when reading an unsigned integer from the block array, the blocks are already in correct order. This way two bit shift operations are enough to extract the desired value. Assume original element size is 32 bits, block length is b and requested element is at position x in the block array and it's compressed within k blocks. Starting byte s of element e is x * b div 8 where div is the integer division. Offset o is the remainder of previous division, o = x*b mod 8. 32-bit value e is read from memory from s. Then we bit shift left 32-b*k-o to remove trailing bits and bit shift right 32-b*k to re-align bits and remove leading bits.

Simple way to calculate block length k of i-th element is to get $select_1$(i+1) and subtract already calculated start block index from it. This causes a second $select_1$ query, which is suboptimal. A much faster option is to iterate forward in bit array until next 1 is found. An even faster option is to read an integer from the bit array, fix it's offset and count trailing zeros. Traveling the bit array is recommended, since it is very simple to implement and is not noticably slower. See Table 4.1 for detailed times.

(maybe to future work) Storing bytes in this fashion may have benefits when reading subsequent (fast next() and previous()?) elements. With some modifications, SIMD instructions can possibly be applied.

TODO: check if rank query takes less when done on single byte array instead of multiples

Table 4.1: Time to calculate end blocks

Version	Average time over 10Mall
select(i+1)	34.97
iterate bit array	34.97
bit shift magic	34.97

- get actual results of rank vs select support size - take advantage of how data is stored

5 Experimental results

- data sets

- comparison to basic implementation + Bri09 - compare with b=2,4,8

- do we need to support search()? - SIMD?

Table 5.1: 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 5.2: 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

6 Conclusion

- here

7 Future work

- something to improve / research?

8 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: *String Processing and Information Retrieval.* Ed. by J. Karlgren, J. Tarhio, and H. Hyyrö. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 122–130. ISBN: 978-3-642-03784-9.

Lemire, D., Kurz, N., and Rupp, C. (2018). “Stream VByte: Faster byte-oriented integer compression”. In: *Information Processing Letters* 130, 1–6. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2017.09.011. URL: http://dx.doi.org/10.1016/j.ipl.2017.09.011.

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

Plaisance, J., Kurz, N., and Lemire, D. (2015). *Vectorized VByte Decoding.* arXiv: 1503.07387 [cs.IR].

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.