Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

Interpolative coding of integer sequences supporting log-time random access

J. Teuhola [*]

Department of Information Technology, University of Turku, Finland

ARTICLE INFO

Article history:
Received 28 August 2009
Received in revised form 25 August 2010
Accepted 12 November 2010
Available online 4 December 2010

Keywords:
Source coding
Data compression
Random access
Interpolative coding
Inverted index

ABSTRACT

Sequences of integers are common data types, occurring either as primary data or ancillary structures. The sizes of sequences can be large, making compression an interesting option. Effective compression presupposes variable-length coding, which destroys the regular alignment of values. Yet it would often be desirable to access only a small subset of the entries, either by position (ordinal number) or by content (element value), without having to decode most of the sequence from the start. Here such a random access technique for compressed integers is described, with the special feature that no auxiliary index is needed. The solution applies a method called interpolative coding, which is one of the most efficient non-statistical codes for integers. Indexing is avoided by address calculation guaranteeing sufficient space for codes even in the worst case. The additional redundancy, compared to regular interpolative coding, is only about 1 bit per source integer for uniform distribution. The time complexity of random access is logarithmic with respect to the source size for both position-based and content-based retrieval. According to experiments, random access is faster than full decoding when the number of accessed integers is not more than approximately $0.75 \cdot n/\log_2 n$ for sequence length n. The tests also confirm that the method is quite competitive with other approaches to random access coding, suggested in the literature.

1. Introduction

Ordered sets of numbers, especially integers, represent a substantial proportion of data volumes in many fields. These sets can appear either directly as application data, such as measurements recorded in *time series*, or as ancillary structures, such as *inverted indexes* for information retrieval. The two main orders of accessing elements in ordered sets are *sequential* and *random*. The corresponding data types are usually called *list* and *array*, respectively. Choosing the processing order is a matter of efficiency. In information retrieval, for example in web search engines, computing the intersection of *inverted lists* is extremely common, and represents often the heaviest part of query processing. An obvious way to compute the intersection is to apply sequential merging of the lists. However, intersecting a short and long list would benefit from direct access to elements of the long one, without having to scan through the complete list.

For long sequences of integers, *compression* can save considerable amounts of memory and, in addition, speed up retrieval because combined reading and decoding is often faster than reading uncompressed data. Compression does not present any essential problem to sequential processing. However, for random access the situation is different, because all practical source codes are of *variable length*. This makes it very difficult to determine the starting address of the i^{th} element in the compressed sequence, without decoding the prefix. What is worse, we would often really want to do random access by *content*, not by

* Fax: +358 2 3338600.
E-mail address: teuhola@it.utu.fi

0306-4573/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2010.11.006

ordinal number. This holds for the above example of intersecting inverted lists. For content-based (associative) access, the two most common techniques are *indexing* and *hashing*. Since we want to keep the original order of the sequence, hashing is not a realistic option. Building an explicit search index for the compressed sequence is of course possible, but not very tempting, because it contradicts compression itself.

Our goal is to develop an effective variable-length compression scheme for *static* arrays of integers, supporting both sequential and random access, without auxiliary indexing. In addition, random access should be possible in both of the mentioned ways, namely by *position* (=ordinal number, called also array index), and by *content* (=element value). Content-based access assumes a sorted array of integers. Random access means that we do not have to scan and decode all the elements preceding the one to be accessed. The access time should be sublinear, preferably at most logarithmic, with respect to the array size. The used storage system is assumed to support physical direct access by memory address.

Our solution is based on hierarchical (recursive) allocation of space to (sub)arrays of integers, by assuming a *worst-case distribution* at each step. The space allocation is done according to a simple formula, with array length (n) and sum of elements (s) as parameters. The same formula is applied in retrieval, recursively, leading to $O(\log_2 n + \log_2 s)$ time for both types of random access. From now on the method will be called *address calculation coding*.

The actual representation of integers is here done by so-called *interpolative coding* (Moffat & Stuiver, 1996, 2000). It is one of the most efficient *non-statistical*, *non-parametric* coding methods in practice, and suits perfectly to our purposes, because it also functions in a hierarchical, recursive manner. Address calculation coding differs from interpolative coding mainly in the lengths reserved for code segments. Allocating space always for the worst case means that in non-worst-cases we sometimes reserve too much. Thus small gaps of unused bits occur here and there, and this is the penalty that we pay for the access features. However, for reasonable distributions of source integers the waste is rather small; roughly 1 bit per integer, compared to basic interpolative coding.

Partial decoding of compressed collections has been studied by various researchers before; related work is reported briefly in Section 2. Interpolative coding is the foundation of our work and therefore reviewed in more detail in Section 3. Our space allocation scheme for sequences of interpolative-coded integers is developed in Section 4. The random access algorithms are explained and their complexity analysed in Section 5. The coding efficiency is estimated theoretically in Section 6 and experimentally in Section 7, where also timing results are reported. Section 8 draws the conclusions.

2. Related work

Most of the existing data compression methods share the property that processing is done in large units, such as files. As for retrieval, this means that usually the whole file, or a large part of it, has to be decompressed before the wanted part of the original data can be accessed. In many applications, such as image processing and retrieval, this is usually not a problem. In some other cases, such as retrieval of text documents by keywords, it would often be desirable to do matching directly in the compressed form. Manber (1997) presented a method supporting this kind of processing, based on the idea that also the search pattern is encoded, enabling the usage of regular string matching algorithms. For structured text, such as XML documents, partial decompression and retrieval is even more desirable because the hierarchical structure defines natural subparts for access. Quite many approaches in this direction have recently been suggested. Example tools include XGRIND (Tolani & Haritsa, 2002), XPRESS (Min, Park, & Chung, 2003), XQZip (Cheng & Ng, 2004), and XSeq (Lin, Zhang, Li, & Yang, 2005).

Ordered sets of numbers constitute another major category of data types. The traditional ways of compressing numbers, especially integers, can be classified in several ways, such as *statistical* and *non-statistical* techniques, *parametric* and *non-parametric* techniques, as well as *context-dependent* and *context-independent* techniques. Further, the source sequence can be *sorted* or *unsorted*. The former can trivially be converted to a sequence of smaller, unsorted values by taking pairwise differences, called d-gaps (Anh & Moffat, 1998), of the elements. Several non-statistical, variable-length codes for integers were surveyed by Fenwick (2003). Various approaches to compressing sequences and sets of integers were reported by Moffat (2008).

Most of the common source coding methods share the property that the code lengths are variable, defined as the number of bits. Examples are the well-known *Huffman code* (Huffman, 1952), *Golomb code* (Golomb, 1966), *gamma* and *delta codes* (Elias, 1975), and *Fibonacci code* (called also *Zeckendorf code*) (Apostolico & Fraenkel, 1987). Tight concatenation of variable-length bit strings implies rather heavy bit manipulation in accessing. A faster alternative is to use a variable number of 8-bit *bytes* to represent each source value, as suggested, e.g. by Williams and Zobel (1999) and Scholer et al. (2002). Access speed can be improved also by aligning groups of variable-length binary codes at machine word boundaries. Tailored implementations of this kind were suggested by Anh and Moffat (2005, 2006).

Accessing and decoding singular numbers in the middle of a sequence of variable-length codes, without decoding the front part of the sequence, requires some *access points* to the compressed data. A theoretically oriented study in this direction was performed by Gupta, Hon, Shah, and Vitter (2006), presenting optimality considerations for compressed sets supporting fast membership, rank, selection, and predecessor queries. In practice, the random access problem has been attacked mainly in two ways: *indexing* and *blocking*. Indexing means sampling of (cumulative) numbers at regular intervals and storing the sampled values (plus related pointers) separately for fast access. Early examples of the indexing approach are methods suggested by Moffat and Zobel (1993, 1996). Blocking, in turn, means gathering codewords into fixed-length chunks with some