UC Irvine
ICS Technical Reports

Title
Fast Huffman code processing

Permalink
https://escholarship.org/uc/item/3jd1c01q

Author
Pajarola, Renato

Publication Date
1999

Peer reviewed

ICS

TECHNICAL REPORT

Fast Huffman Code Processing

Renato Pajarola

October 1999

Information and Computer Science

University of California, Irvine

Fast Huffman Code Processing

Renato Pajarola
Information and Computer Science
University of California, Irvine

Keywords algorithms, data structures, data compression, decoding, Huffman codes

1. Introduction

Data compression research constantly attracts the interest of many researchers both in theoretical foundations of computing and in application oriented fields. In the last two decades the fundamentals of data compression have been laid [3, 6, 9, 10, 15, 16, 17, 22] and efficiently applied to text and image compression [8, 11, 19]. Currently data compression is of increasing interest again because of the growing amount of data processed in applications and transferred over the internet. In particular compression of geometric data is currently an active research area [4, 12, 20, 21], and it is also important to perform operations on the compressed data directly – working in the compressed domain – instead of decompressing prior to any processing [1, 2, 5, 7, 13, 14].

There has been some work on memory efficient and fast construction of Huffman and prefix codes but only very little on fast and efficient decoding. However, fast decoding and scanning through compressed data is more important than code construction and encoding with increasing network transmission and disk access speeds. Huffman decoding time cost is linear in the size of the compressed data stream and bounded by the number of symbols it has to output. However, this cost estimate does not sufficiently take into account how much time is used to process and decode every single compressed data bit. It will be quite a difference in speed if every single bit has to be read and tested compared to processing bytes (8 bits at once). Also if a search in a compressed file involves identifying the boundary between two symbols at some location in the compressed file it is important to quickly skip through the file – reading bytes – and only consider a few individual bits close to that boundary.

Our interest in an efficient Huffman decoder is motivated by research in geometry compression [12], where a fast decompressor is crucial, and by research in pattern matching in compressed files [13], where it is important to find symbol boundaries at arbitrary positions in the compressed data stream. The idea of the presented fast Huffman decoding data structure and algorithm is similar to the code tables presented in [18]. However, we present a much more intuitive approach based on the binary code tree itself, and we extend the data structure in such a way that it allows to test for symbol boundaries in constant time. Note that the presented method works for any binary prefix code.

2. Fast Huffman decoding

Huffman coding [6] creates minimal redundancy codes for a given set of symbols and their respective occurrence frequencies. It constructs a binary code tree where each leaf represents a symbol and the path from the root to the leaf defines the variable length code for that symbol. Decoding is performed by starting at the root node of the code tree, and recursively traversing the tree according to the bits from the compressed input data stream, i.e. going to the left child for a 0 and going right for a 1, until you reach a leaf which signals that a certain symbol has been fully decoded. Generally this involves testing every single bit and branching in the tree accordingly.

To speed up decoding performance we want to read and process the data by groups of bits, i.e. bytes, to eliminate most of the testing. For any start-node in the tree and a given sequence of bits, the

resulting end-node from traversing the tree according to these bits is defined uniquely by actually traversing the tree and restarting at the root whenever reaching a leaf. For a fixed word size of k bits, a table of size 2^k for every node, not including the leaves, can be used to navigate in the tree, see also Figure 1. This allows us to *jump* efficiently from any node to another in the code tree by reading bytes instead of single bits.
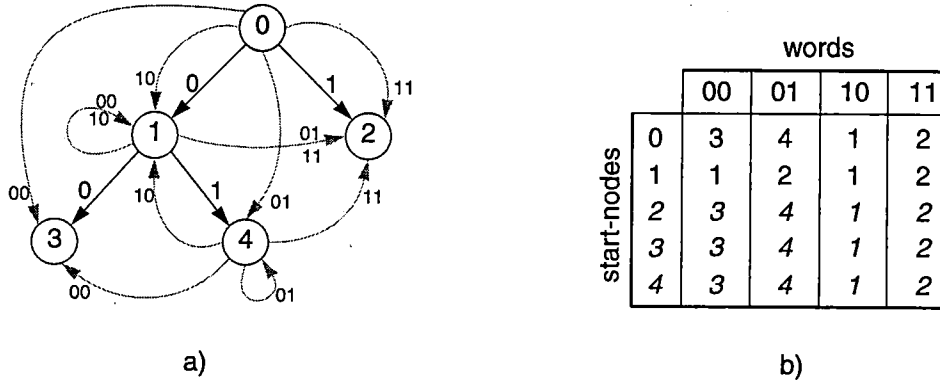

	words			
	00	01	10	11
0	3	4	1	2
1	1	2	1	2
2	3	4	1	2
3	3	4	1	2
4	3	4	1	2

a)   b)

FIGURE 1. a) Huffman code tree with some indicated node transitions using a word size of 2 bits for the codes A=00, B=01 and C=1. b) The complete node transitions table for all nodes indicating the end-node for any given combination of start-node and data word. Note that all leaves have the same *jump* table as the root node and can be left out to save storage.

The presented fast navigation in the binary code tree is not sufficient for decompression, also the output symbols have to be known for every node transition. In the example of Figure 1, if starting from node 1 and reading the 2bit-word 01 we end up in node 2, however, we also decoded the symbols A and C. This can be captured in a similar table as the node transition table of Figure 1 b), Figure 2 depicts this output table for the same encoding and binary code tree.

from node	word	symbols
0	00	A
0	01	B
0	10	C
0	11	C, C
1	00	A
1	01	A, C
1	10	B
1	11	B, C

FIGURE 2. Table listing the output symbols for every node transition. Note that the leave nodes are identical to the root node and can be omitted.

The decoding procedure is very simple, see also Algorithm 1. With the given data structures described above, the decoder can read the compressed data stream by words and update the current position in the binary code tree according to the node transition table. Prior to actually updating the current node, the output symbols have to be read from the output symbol table. Note that the binary code tree itself is not used anymore, the node transition table and output table fully specify the decoding. However, the binary code tree is used to construct the two tables in a preprocessing step as indicated above. For this, we traverse the tree starting from every node for all possible bit-patterns of a word, continuing at the root when reaching a leaf, and update the two tables accordingly.

```
PROCEDURE Decode (in: InputStream; code: CodeTree);
VAR byte: Byte; cur: Node;
BEGIN
    cur := code.root;
    WHILE NOT in.EOF() DO
        byte := in.readByte();
        code.outputSymbols(cur, byte);
        cur := code.nextNode(cur, byte);
    END;
END Decode;
```

ALGORITHM 1. Pseudocode for the fast Huffman decoding algorithm. CodeTree is the data structure with the transition and symbol output tables.

3. Recovering symbol boundaries

Processing compressed data files by words instead of bits is not only important for raw decompression speed but also when the task is to quickly skip through the compressed file and stop at a certain point. In particular, in [13] the problem was to sequentially read the compressed data, without actually decompressing it, and keeping track of how many symbols have been encountered. Furthermore, it was also required to stop at any arbitrary position and know where the last symbol ended in the compressed bit stream, besides knowing how many symbols have been skipped so far.

Fast processing of the compressed data stream is already outlined in the previous section, what is left is to maintain some more information for every node transition to solve the problems mentioned above. The maximal number of symbols encoded in k bits is k. Therefore, a bit-field of size k is sufficient to indicate symbol boundaries of variable length codes within a word of size k bits. Such a bit field is needed for every node transition to be able to recover the bit-positions where the last skipped encoded symbols end in the last word that has been read, see also Figure 3. To comply with the second requirement, how many symbols have been encoded at a certain position in the compressed data stream, one more entry for each node transition is sufficient. This entry is actually the number of 1s in the previously explained bit-field, representing the number of symbols ending in the last read word.

from node	word	endings field	#
0	00	[0,1]	1
0	01	[0,1]	1
0	10	[1,0]	1
0	11	[1,1]	2
1	00	[1,0]	1
1	01	[1,1]	2
1	10	[1,0]	1
1	11	[1,1]	2

FIGURE 3. Bit field of third column indicates symbol endings in processed word. The last column represents the number of encoded symbols that ended in the word that has been read.

The following Algorithm 2 shows fast scanning through a compressed file without decompressing every symbol, however, with counting the symbols that have been encoded. It also presents a way to determine in constant time whether and where a symbol ended in the last word that has been read.

```
PROCEDURE Counting (in: InputStream; code: CodeTree);
VAR count, n, i: Integer; field: BitField; byte: Byte; cur: Node;
BEGIN
    cur := code.root;
    count := 0;
    n := 0;
    WHILE NOT condition to stop DO
        byte := in.readByte();
        INC(n);
        count := count + code.numberOfSymbols(cur, byte);
        field := code.endingsField(cur, byte);
        cur := code.nextNode(cur, byte);
    END;
    out.print(count, " symbols encoded so far");
    i := 8;
    WHILE NOT field[i] DO DEC(i) END;
    IF i > 0 THEN
        out.print("last symbol at ", (n-1)*8+i, "-th bit in data stream");
    ELSE
        out.print("no symbol ends in last word");
    ENDIF;
END Counting;
```

ALGORITHM 2. Pseudocode for fast counting and symbol boundary test. The CodeTree data structure is enhanced with the described endings bit-field and symbol counts for every node transition.

4. Conclusions

As large main memory becomes more and more available at reasonable prices, processing speed of large data sets, i.e. from secondary storage, becomes more important than techniques for memory efficient internal data structures which are small compared to the available main memory size. The space needed for the node transition and symbol output tables described in this paper depends on the word size and the number of different symbols. For example, for bytes and 256 symbols, the node transition table for every node has 256 entries, and each entry consists of a field for the next node (one byte) and an array for the output symbols, often not more than two or three per transition, which depends on the shape of the binary code tree. If on the average using 10 bytes for an entry, including the symbol array and a pointer to it, the data structure consumes less than 1MB of main memory ($255 * 256 * 10 \approx 0.6MB$). The additional fields for the symbol endings and the number of symbols encoded in one word would account for another 2 bytes per entry in our example, thus not significantly enlarging the overall data structure. Note that the tables do not have to be transmitted in case of a network communication because they can efficiently be reconstructed by sender and receiver if necessary from the binary code tree, which requires only 2 bits per node for encoding it.

Reading the compressed data stream in bytes still requires $O(n)$ time for n being the size of the compressed data, however, the constant is reduced; instead of testing 8 bits, only one access to the node transition table is required. Nevertheless, outputting the encoded symbols still requires $O(k)$ time for k symbols. On the other hand, counting the k symbols only takes $O(n)$ time (*with $n < k$*), since updating the number of symbols for every node transition can be done in constant time. Note also that the preprocessing cost for constructing the node transition tables is constant with fixed number of symbols and word size.

The proposed algorithms and data structures could significantly improve the processing speed of prefix code decompression methods, and in particularly support the development of real-time geometry decompression, a current issue in graphics, multimedia and internet based computing. Further-

more, also performing operations in the compressed data domain, such as pattern matching or random access, benefits from the proposed methods in this paper.

References

[1] Amihood Amir and Gary Benson. Efficient two-dimensional compressed matching. In James A. Storer and John H. Reif, editors, *Proc. Data Compression Conference*, pages 279–288. IEEE, 1992.

[2] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in Z-compressed files. In *Proc. of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 705–714. ACM, 1994.

[3] John G. Cleary, Radford M. Neal, and Ian H. Witten. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[4] Michael Deering. Geometry compression. In *Proceedings SIGGRAPH 95*, pages 13–20. ACM SIGGRAPH, 1995.

[5] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proc. Symposium on Theory of Computing*, pages 703–712, 1995.

[6] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. Inst. Electr. Radio Eng.*, pages 1098–1101, 1952.

[7] Guy Jacobson. Random access in huffman-coded files. In James A. Storer and John H. Reif, editors, *Proc. Data Compression Conference*, pages 368–377. IEEE, 1992.

[8] Weidong Kou. *Digital Image Compression: Algorithms and Standards*. Kluwer Academic Publishers, Norwell, Massachusetts, 1995.

[9] Abraham Lempel and Jacob Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[10] Abraham Lempel and Jacob Ziv. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

[11] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation, Compression and Standards*. Plenum Press, New York and London, second edition, 1995.

[12] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. Technical Report GIT-GVU-99-05, GVU Center, Georgia Institute of Technology, 1999.

[13] Renato Pajarola and Peter Widmayer. Pattern matching in compressed raster images. In *Third South American Workshop on String Processing WSP 1996*, International Informatics Series 4, pages 228–242. Carleton University Press, 1996.

[14] Renato Pajarola and Peter Widmayer. Spatial indexing into compressed raster images: How to answer range queries without decompression. In *Proc. Int. Workshop on Multi-Media Database Management Systems*, pages 94–100. IEEE, Computer Society Press, Los Alamitos, California, 1996.

[15] Jorma Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664, September 1983.

[16] David Salomon. *Data compression: the complete reference.* Springer-Verlag, New York, 1998.

[17] Khalid Sayood. *Introduction to data compression.* Morgan Kaufmann Publishers, San Francisco, California, 1996.

[18] Andrzej Sieminski. Fast decoding of the huffman codes. *Information Processing Letters*, 26(5):237–241, Januery 1988.

[19] James A. Storer, editor. *Image and Text Compression.* Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

[20] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[21] Costa Touma and Craig Gotsman. Triangle Mesh Compression. In *Proceedings Graphics Interface 98*, pages 26–34, 1998.

[22] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, June 1984.