In *Proceedings IEEE ITCC Conference*, pages 206–211, 2003.

Fast Prefix Code Processing

Renato Pajarola
Information & Computer Science
University of California, Irvine
pajarola@acm.org

Abstract

As large main memory becomes more and more available at reasonable prices, processing speed of large data sets becomes more important than reducing main memory usage of internal data structures which are small compared to the available main memory capacity. In this paper we describe the use of a finite state machine for fast processing of prefix codes that significantly improves decoding performance in practice, and that is easy to implement. We present an intuitive explanation of this method, an extension to discover symbol boundaries in compressed data, implementation details, and we also provide experimental performance results.

Keywords: variable length coding, data compression, fast decoding, Huffman codes, prefix 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 [4, 8, 11, 12, 19, 20, 21, 27] and efficiently applied to text and image compression [10, 14, 24]. Currently data compression is of increasing interest again because of the growing amount of data processed in applications and transmitted over the internet. In particular compression of geometric data is currently an active research area [5, 15, 25, 26], 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, 6, 9, 17, 18].

There has been some work on memory efficient and fast construction of Huffman and prefix codes but only little on fast and efficient decoding (see also [7]). However, with increasing network transmission rates and disk access speeds, fast decoding and scanning through compressed data is more important than code construction and encoding. Prefix 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 theoretical cost estimate does not sufficiently take into account actual CPU time used to process and decode every single compressed data bit. It is quite a difference in speed if every single bit has to be processed individually, compared to processing bytes (8 bits at once) or even larger machine words. Also if a search in a compressed file only involves identifying the boundary between two symbols at some location in the compressed file it is important to quickly skip through the file – reading and processing bytes instead of single bits – and only examine individual bits adjacent to the actual symbol boundary if necessary.

Our interest in an efficient Huffman or prefix-code decoder is motivated by research in geometry compression [15], where a fast decompressor is crucial, and by research in pattern matching in compressed files [17], where it is important to quickly find symbol boundaries at arbitrary positions in the compressed data stream. The idea of the presented fast prefix code decoding data structure and algorithm is theoretically the same as the approaches presented in [3] and [23]. However, we present a much more intuitive approach based on the binary code tree itself, we extend the data structure in such a way that it allows to test for symbol boundaries in constant time, provide analysis of the time and space costs, and we present experimental run-time results. Also a very fast decoder based on a different canonical coding [22] is presented in [13].

The paper is organized as follows: In Section 2 we shortly describe Huffman and prefix coding and the problem of efficient decoding. Section 3 presents the basic data structures and algorithm for fast prefix code decoding, and Section 4 extends the approach to efficient symbol boundary recovery. Actual run-time experiments of fast prefix code decompression are presented in Section 5, and Section 6 concludes the paper.

2. Problem

Huffman coding [8] creates minimal redundancy codes for a given set of symbols of the source alphabet, and their respective occurrence frequencies in a data stream to be compressed. It constructs a binary code tree where each leaf represents a symbol of the alphabet, and the path from the root to the leaf defines the variable length code for that symbol. The connections in the binary tree represent bits in the code, here we assume that a pointer of a node to its right successor represents a 1 and the pointer to the left child is a 0. 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 reaching a leaf node which signals that a certain symbol has been fully decoded. Generally this involves testing every single bit and branching in the tree accordingly.

The basic Huffman code data structure is a binary tree with no other information in the inner nodes than two pointers to the left and right child nodes, and the leaf nodes only storing the source symbol corresponding to the binary code represented by the path from the root node to this leaf node. For m source symbols in the alphabet the binary code tree has m-1 inner nodes and m leaf nodes, thus storage cost is O(m). A theoretically optimal decoding algorithm starts at the root node, tests a single bit and branches accordingly to the left or right child node until a leaf node is reached. At the leaf node the stored source symbol is outputted, and the decoding process is reset to the root node. Therefore, decoding cost is linear $O(\bar{n})$ in the number of bits $\bar{n}$ of the compressed data stream.

While theoretically optimal, bit-wise processing is obviously not optimal with respect to actual hardware architectures. Due to parallelism in digital logic design, microprocessors can process multiples of bits (i.e. bytes) as fast as individual bits. For example, although the theoretical cost of reading data sequentially bit per bit is the same as reading bytes, actual hardware implementation makes reading bytes up to 8 times faster. Such constant speed-up factors which are theoretically negligible, nevertheless are very important in practical implementations. In this paper we address this problem of improving actual CPU processing time of prefix codes for decoding and retrieving symbol boundaries in the compressed data stream.

3. Fast prefix code decoding

Considering the basic binary code tree data structure and decoding algorithm outlined in the previous section it is clearly required to take into account every single bit during the decoding process. Not considering any particular bit for decoding will result in a non-deterministic traversal of the binary code tree. However, to speed up decoding performance we want to read and process the data by multiples of bits, i.e. bytes or larger words, to take advantage of microprocessor architectures, and reduce most of the testing for the tree traversal. In this section we show how a binary code tree can be enhanced for word-wise tree traversal for improved decoding performance.

Given a current node in the binary code tree and a sequence of bits from the compressed data stream, the resulting end-node from traversing the tree according to these bits is defined uniquely by actually traversing the binary code tree itself bit by bit, and restarting at the root node whenever reaching a leaf. Thus for a fixed size sequence of bits, and a start-node in the binary code tree, all possible end-nodes can be precomputed as shown in Figure 1 and stored in a node transition table as shown in Figure 2. Note that leaf nodes have all the same node transition table as the root. For a fixed word size of k bits, 2^k possible bit sequences or node transitions must be considered at every node. This allows us to *jump* efficiently from any node to another in the code tree by processing multiple bits simultaneously instead of single bits.

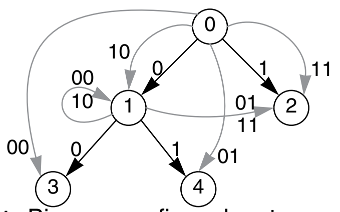

FIGURE 1. Binary prefix-code tree with node transitions using a word size of 2 bits for the codes A=00, B=01 and C=1.

	words			
start-nodes	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*

FIGURE 2. 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 node transition table as the root node and can be omitted out to save storage.

The presented data structure, binary code tree with node transition table, is not sufficient for decompression. Besides all possible node transitions for arbitrary bit-sequences, also the output symbols have to be known for every node transition. Any node transition may implicitly pass one or multiple times through a leaf of the binary code tree. For every visit of a leaf node, even implicitly given by a node transition, a decoded source symbol can be outputted. 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 as we implicitly passed through leaf node 3 (symbol A) and explicitly ended in leaf node 2 (symbol C). Decoded symbols, if any, can be captured for every node transition in a *symbol output table*. Similar to the node transition table, this table can be constructed by a bit-wise traversal of the binary code tree, initializing the traversal at the start-node for each node transition and restarting at the root node whenever reaching a leaf. An entry in the symbol output table for a given node transition consists of all source symbols corresponding to leaf nodes that have been traversed for this node transition. Figure 3 depicts the symbol output table for the same encoding and binary code tree as used in Figure 1.

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 3. 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 and outlined in Algorithm 1 below. Using the given data structures described above, the decoder can process the compressed data stream reading multiple bits in one step, 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 determined from the symbol output table and written to the output stream. Note that the binary code tree itself is not used anymore in this algorithm, the node transition table and output table fully specify the decoding process. However, the binary code tree is needed to construct the two tables at initialization in a preprocessing step as indicated above. In Algorithm 1, InputStream and OutputStream refer the file objects that read from and write to files, and Code Tree is an object that stores the node transition and symbol output tables as two-dimensional indexed arrays.

```
PROCEDURE Decode (in: InputStream; out: Output-
Stream; code: CodeTree);
VAR byte: Byte; cur: Node;
BEGIN
   cur := code.root;
   WHILE NOT in.EOF() DO
      byte := in.readByte();
      out.print(code.outputSymbols[cur][byte]);
      cur := code.nextNode[cur][byte];
   END;
END Decode;
```

ALGORITHM 1. Pseudocode for the fast prefix code decoding algorithm using bytes. CodeTree is the data structure with the transition and symbol output tables.

Lemma 1 *Using the node transition and symbol output tables, Algorithm 1 correctly decodes a bit stream of prefix codes in* $O(\bar{n}+n)$ *time with respect to the compressed file size* $\bar{n}$ *and the number n of actually encoded symbols in the bit stream. (Node transition and symbol output tables are derived from the same binary code tree that generated the prefix codes in the bit stream.)*

Proof Correctness of the decoding algorithm follows from the construction of the data structures. Due to the construction of the node transition table which incorporates all possible 2^k bit combinations of a fixed word size k at every node, the word based traversal of the binary code tree is guaranteed to be correct. Also due to the construction of the symbol output table which records for every possible node transition all symbols that have been decoded passing through a leaf node of the binary code tree, the decoding algorithm finds all encoded symbols.

Reading the input stream requires $O(\bar{n})$ time in the while-loop. Processing a byte requires constant time to determine the next node in the binary code tree using a table look-up in the node transition table. Constant time is also required to find the corresponding entry in the symbol output table, however, outputting the symbols requires time proportional to the number of symbols, amortized over the entire input stream $O(n)$ time. Thus the overall time cost is $O(\bar{n}+n)$. □

Note that the theoretical optimal running time $O(\bar{n})$ of a bit-wise traversal of the binary code tree actually incorporates also outputting $O(n)$ symbols. However, since $\bar{n}$ is the number of bits in the compressed data stream, and because no more than one symbol can be decoded per bit ($\bar{n} \gg n$), the overall running time is linear in $\bar{n}$. Although theoretically bounded by $O(\bar{n}+n)$, since constant factors are omitted, the presented algorithm practically runs much faster. This is due to the fact that reading and decoding the bits – traversing the binary code tree – from the input data stream exactly requires $\bar{n}/k$ steps compared to $\bar{n}$ steps for a bit-wise tree traversal, a speed-up factor of k for traversing the binary code tree. Additionally to reading and decoding, $O(n)$ time is required to output the n decoded source symbols. Experimental results are reported in Section 5.

The proposed algorithm does not need the actual binary code tree, however, requires main memory storage space for the node transition and symbol output tables. For a given set of m source symbols in the alphabet, and a word size of k bits the node transition table is of size $O(m \cdot 2^k)$, and the symbol output table has a worst case space cost of $O(m \cdot 2^k \cdot k)$ since at most k symbols may be decoded by a bit-sequence of length k for prefix codes. For a fixed constant word size k the storage requirement is thus linear in the number of symbols m in the source alphabet. The node transition and symbol output tables do not have to be stored along with the compressed bit stream as they can be reconstructed in a preprocessing step before decoding.

4. 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 position. In particular, in [17] the problem was to sequentially read the compressed data, without actually decompressing it, and keeping track of how many symbols have been encountered at any given position. Furthermore, it was also required to stop at any arbitrary position in the compressed data stream, and to know where the last symbol ended in the bit stream relative to the stopping position.

The basic method for fast processing of the compressed data stream has already been outlined in the previous section. Additionally, it is required to know the number of symbols encoded in the bit-stream up to the current position, and the ending of the last complete symbol code. Note that 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. The bit-field actually records every passing of a leaf node with respect to a bit-wise code tree traversal. Such a bit field is required for every node transition to allow recovery of the bit-positions where the skipped encoded symbols end in the last word that has been processed. This bit-field can be recorded in a table (see also Figure 4) similar to the symbol output table presented in the previous section. Furthermore, to count the encoded symbols without actually decoding them, the activated bits in the symbol ending bit-field could be counted for every node transition. However, an additional integer entry per node transition that stores the number of bits that are set in the bit-field provides faster access to this information, see also Figure 4.

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 4. The bit field of the third column indicates symbol endings in the processed word for a particular node transition. 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 and outputting any decoded symbol, however, counting the source symbols that have been encoded. It also shows how to determine in constant time whether and where a symbol ended in the last word that has been read. Note that Algorithm 2 can be extended to include an application dependent stopping criterion such as for example stopping after m symbols. In addition to Algorithm 1, the CodeTree data structure includes two additional two-dimensional arrays to get the information on the number of encoded symbols, and the symbol boundaries for each node transition.

Lemma 2 *Algorithm 2 processes the compressed bit-stream in* $O(\bar{n})$ *time with respect to the compressed file size* $\bar{n}$. *It correctly counts the encoded symbols, and reports the last bit of the last entirely represented symbol in the compressed bit stream with respect to the stopping position.*

Proof Reading the input stream requires $O(\bar{n})$ time in the while-loop. Updating the current node in the binary code tree requires constant time for a node transition. Furthermore, also updating the symbol count and keeping track of the current bit-field are performed in constant time per node transition, both operations only require a table look-up. After halting the file processing, at most k steps are required using a word size of k bits to find the ending of the last entire symbol in the current bit-field, and k is an implementation dependent constant. Thus the overall time cost is $O(\bar{n})$.

The algorithm is correct since for each possible node transition the bit-field records all endings of symbols in the k-bit word of the node transition, thus also the derived symbol counter per node transition is exact. □

The theoretical running time $O(\bar{n})$ is optimal, however, the actual number of steps performed in the while-loop is only $\bar{n}/k$ for a compressed bit stream of $\bar{n}$ bits. Note that in practice this is much faster than decoding the prefix code bit by bit which would require actually $\bar{n}$ steps. The storage requirement of the bit-field table is $O(m \cdot 2^k \cdot k)$, which for a fixed constant k is linear in the number m of symbols in the source alphabet.

```
PROCEDURE Counting (in: InputStream; out: Out-
putStream; 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.

5. Experiments

We conducted several experiments using various data types, and sources. The main purpose is to show the decoding time performance improvement of the presented prefix codes decoding algorithm compared to standard bit-wise decoding. For reference, we also included comparisons to the *pack* and *unpack* programs, available on unix machines, which are based on canonical Huffman codes [22], and which are known to be very fast Entropy coders (see also [28] for comparisons among several compression methods). All tests were performed on a SGI O2 workstation with a 300MHz MIPS R12000 microprocessor running an IRIX 6.5 operating system.

The first experiment presented in Table 1 reports decoding time performance of a bit-wise Huffman decoder, called HBit, compared to our word based approach, called HByte, and compared to the unpack program. HByte uses a word size k of 8 bits. The data structures holding the Huffman code tree in HBit are three integer arrays for the left, right and parent relations of the nodes in the binary code tree, and decoding is performed traversing this tree top-down starting at the root for every variable-length symbol. The node transition information for HByte is maintained in a table of nodes with each having an array of node transition records. A node transition record consists of two integers for the next node and the number of symbols, and a list of output symbols. The decoding performance is improved by a factor between 4 and 5, and our approach is also competitive in decoding speed with unpack. As expected, efficiency in terms of compression ratio is equal to the pack program.

TABLE 1. Decompression time performance, and compression efficiency.

Files	HBit	HByte	Unpack	Size	Compressed	Packed
Binary	0.25s	0.07s	0.09s	268K	177K	178K
Man pages	5.59s	1.32s	1.53s	5782K	3555K	3554K
PDF	1.05s	0.28s	0.28s	754K	728K	729K
Postscript	2.8s	0.62s	0.74s	2860K	1847K	1847K

Note that Table 1 does not include set-up time of the HBit and HByte decoders for constructing or loading the binary code tree. The decompression time that includes loading of the code tree data structures are 0.3s, 5.8s, 1.2s and 2.9s for HBit, and 0.1s, 1.38s, 0.33s and 0.68s for HByte. Thus even if the data structures have to be read from disk, the decoding performance of HByte is still competitive to unpack, and highly superior to the bit-wise Huffman decoder.

Compared to a binary prefix-code tree, the presented decoding approach imposes a significant space overhead for the node transition tables which is inefficient for fairly small files, and in the case that every file uses different variable-length codes. For large files, or environments where the same variable-length codes are applied to multiple files the proposed method is very efficient. An uncompressed binary representation of the node transition tables used in the experiments above requires 640K bytes. However, using unix *gzip* compression, a LZ77 [11] implementation, the node transition table only needs 156K bytes and can be loaded and decompressed in 0.07s, almost as fast as loading the uncompressed tables which takes about 0.05s. For larger files such as the man pages example, this enlarges the compressed file insigificantly to 3711K, and decoding in 1.39s, including reading and decompressing the node transition table, is still faster than the unpack program in that case. Note that the tables do not necessarily have to be transmitted in case of a network communication because they can efficiently be reconstructed by sender and receiver from the binary code tree, which requires only 2 bits per node to encode it.

Entropy coders are most efficient for prediction error coding as widely used in image compression. We used the approach presented in this paper in our work [16] to speed up the overall decompression performance of decoding compressed progressive triangle meshes [15] by a factor of two. Efficient image and geometry compression methods using prediction error coding generate prediction error frequency distributions that are exponentially decreasing with larger errors. Such prediction error distributions are very similar to the Laplace distribution $L_\upsilon(x) = (1/\sqrt{2\upsilon})e^{-\sqrt{2/\upsilon}|x-\mu|}$ with variance υ and mean μ (0 for symmetric error distributions). Figure 5 shows the Laplace distribution for variances 0.7, 0.3, and 13.0.

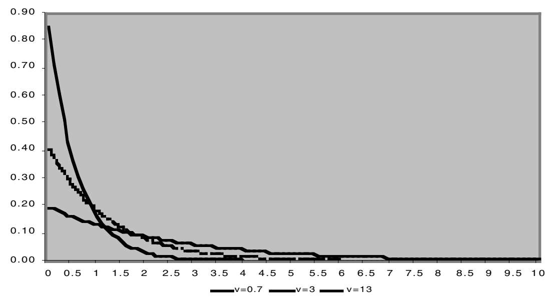

FIGURE 5. Laplace distribution $L(x) = 1/\sqrt{2v}e^{-\sqrt{2/v}|x|}$ for three different variances 0.7, 3.0 and 13.0.

In our second experiment reported in Table 2, we applied the presented fast decoding method to random numbers generated according to the Laplace distribution $L_\upsilon(x)$. This experiment simulates the use of the presented approach in a prediction error coding system as used in image and geometry compression. The simulation generated 1'000'000 random numbers for various Laplace distributions with variances υ ranging from 0.01 up to several hundreds. The performance improvements vary from a factor of 2 for very large variances (extremely flat distributions) to a factor of 9 for small variances (very skewed distributions). As with compression efficiency, the decoding speed improvements increase with smaller variances (which is equal to better approximation accuracy of prediction calculations in prediction error coding).

TABLE 2. Prediction error coding decompression time performance.

Method	υ=0.03	υ=0.6	υ=1.7	υ=13.2	υ=99.5
HBit	0.67s	1.0s	1.17s	2.15s	2.87s
HByte	0.08s	0.15s	0.17s	0.75s	1.16s

6. 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 proposed fast prefix code decompression method significantly improves decoding performance at the expense of main memory usage.

Nevertheless, the space cost for the node transition tables is still small compared to typically available main memory configurations. While reading the compressed data stream in bytes using the presented approach still requires $O(\overline{n})$ time, for $\overline{n}$ being the size of the compressed data in bits, the main CPU time spent for testing and branching in the binary code tree is reduced by a factor of k for word sizes of k bits.

The proposed algorithms and data structures are very easy to implement, and significantly improve the processing speed of prefix code decompression as shown in our experiments. In particular, the presented method supports the development of real-time geometry decompression, a current issue in graphics, multimedia and internet based computing. Furthermore, operations performed in the compressed data domain, such as pattern matching, counting or random access, can also take advantage from the proposed algorithms.

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] Y. Choueka, S.T. Klein, and Y. Perl. Efficient variants of huffman codes in high level languages. In *Proceedings of 8th ACM SIGIR conference*, pages 122–130. ACM, 1985.

[4] 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.

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

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

[7] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, 1990.

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

[9] 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.

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

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

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

[13] Alistair Moffat and Andrew Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.

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

[15] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January-March 2000.

[16] Renato Pajarola and Jarek Rossignac. Squeeze: Fast and progressive decompression of triangle meshes. In *Proceedings Computer Graphics International CGI 2000*, pages 173–182. IEEE, Computer Society Press, Los Alamitos, California, 2000.

[17] 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.

[18] 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.

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

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

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

[22] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

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

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

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

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

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

[28] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, 1999.