Byte-oriented Decoding of Canonical Huffman Codes

Yakov Nekritch [*]

Abstract

We design an efficient algorithm for decoding canonical Huffman codes. The decoding is based on consequential look-up tables.

We also present a data structure for reading bit sequences without using bit operations. Our algorithm, combined with this data structure allows fast byte-oriented decoding without causing high memory requirements.

[*]Institut für Informatik , Universität Bonn, Römerstr. 164, D-53117, Bonn, Germany, email: yasha@cs.uni-bonn.de

1 Introduction

Huffman codes are prefix codes with minimum redundancy. Being a part of many popular data formats and compression utilities, such as JPEG, MPEG, PNG, gzip, pkzip, lha, zoo and arj, they are one of the most popular compression techniques in use nowadays.

Table look-up decoding of Huffman codes was considered by Sieminski [10] and Choueka,Klein, and Perl[2]. Conell[3] describes a special case of Huffman codes,the *canonical Huffman codes*. Klein[7] and Moffat and Turpin [8] describe efficient decoding methods for canonical codes. Iyengar and Chakrabarty[6] describe a finite-state machine approach to decoding Huffman codes.

Most Huffman decoding techniques are bit-oriented , i.e. they use bit-oriented operations, such as bit shifts and bitwise logical operations. The method, described in [2] is byte-oriented, i.e. no bit-oriented operations are used. However this method puts high memory requirements for large alphabets. In this paper an algorithm and a data structure for byte-oriented decoding of canonical Huffman coding are described. We show, that byte-oriented decoding of canonical Huffman codes can be achieved without high memory requirements even in case of large alphabets.

2 Canonical Huffman codes.

Canonical codes are a subclass of Huffman codes, described by Conell[3] and Schwartz and Kallick[11]. The canonical Huffman tree can be described by the following property: if we scan the leaves in pre-order they appear in non-increasing order of their depth. The canonical codes have a *numerical sequence property*, i.e. codewords with the same length are binary representations of consecutive integers. Further, the first codeword of length i f_i can be computed from the last codeword of the length $i-1$ d_{i-1} with the formula $f_i = 2(d_{i-1}+1)$. Due to these properties an explicit structure of the Huffman tree does not have to be transmitted to the decoder. It is enough if we specify (1) the length of the longest codeword and (2) the number of codewords for each length. In addition to providing a compact representation, canonical codes can also be used for efficient decoding.

Once the length of the current codeword is known, it can be decoded by several arithmetic operations in the following way. Supposed we have read the prefix b of the current codeword and all codewords with prefix b have length l. Indexes of the codewords with prefix b are consecutive integers and the codewords themselves are binary representations of consecutive integers.Let $l(b)$ be the length of the prefix b and $first_l$ be the index of the first codeword with length l and b_n be a value of the next $l - l(b)$ bits. Then the index of the current codeword can be computed as $b_n + first_l$. This idea is used in the algorithm, described in [7]. Single bits are read from the input stream, until the codeword length l can be determined, then we read another $l - l_b$ bits and compute the codeword index with the above formula. A special data structure, called an *sk-tree* is used to check, whether a codeword length can be determined from the read bits. An sk-tree is traversed like a regular Huffman tree until a leaf is reached. The leaves correspond to minimal codeword prefixes, such that all codewords with a given prefix have the same length. A leaf of the sk-tree contains information about the codeword length.

3 Finite-state machine approach to Huffman decoding.

We can also regard the nodes of the Huffman tree as states of the finite-state machine. Decoding starts at the root of the tree and depending on the input bit, the control is transfered to the state, corresponding to the left or right son (in the tree) of the current state until the state, corresponding to the parent of a tree leave is reached. After reading the next bit, we output symbol, corresponding to the tree leave and transfer control to the "root" state. The number of states in such a finite-state machine equals to the number of internal nodes in a Huffman tree, i.e. $N - 1$.

This approach can be extended to the case of multiple-bit inputs (see [2]). In this case we read a sequence of k bits at each step of the algorithm. This sequence contains zero or more decipherable codewords and the "rest" i.e. a prefix of a codeword, that cannot be decoded yet. We output symbols, corresponding to decipherable codewords and transfer control to the state, corresponding to the prefix of a not yet decoded codeword, contained in the string. The number of states equals to the number of different codeword prefixes, which in turn equals to the number of internal nodes in the Huffman tree, $N-1$. There are 2^k transitions at each state, that correspond to all possible binary sequences of length k, and the number of symbols, output at each transition, is between 0 and k. Thus, the space required is proportional to $2^k(N-1)$ as for every internal tree node a table of 2^k next-state pointers must be stored. This approach results in high speed decompression, provided that N and k are not so large . If $k=8$, this technique can also be used for a bytewise processing of input stream, thus completely avoiding the use of bit-oriented operations. The drawback of this approach is high memory requirements for the large alphabet sizes N. A similar approach to Huffman decoding, which also enables fast byte-oriented decompression and has high memory requirements is described in [10].

The techniques, used for decoding of canonical Huffman codes, use bit-oriented operations to manipulate single bits or bit sequences. The techniques, described in [10] and [2] allow for a byte-oriented implementation. They do not make use of numerical sequence property and can be applied to any Huffman code. However with alphabet growth space requirements grow dramatically and may become prohibitive for large alphabets (>1000 symbols). Thus in algorithm, described by Choueka et al.([2]), the space requirement is proportional to $1024 * N$ and for storing the tables for an alphabet with 2000 symbols several Megabytes would be necessary. In the next section we will show, that in case of canonical Huffman codes, fast byte-oriented decoding can be performed under reasonable space requirements.

4 Decoding with sequential look-up tables

In this work we describe a table look-up decoding method. It leads to fast decoding without causing too high memory requirements. Besides that, combined with a special data structure, it enables memory-efficient decoding without bit-oriented processing of the input stream.

4.1 Algorithm description

In the rest of this work we will use the following notation. l_{min} will denote the minimal codeword length, l_{max} will denote maximal codeword length, $l_{b_{min}}$ and $l_{b_{max}}$ denote minimal and maximal codeword lengths for codewords with prefix b.

Instead of reading a fixed number of bits, we use the already read codeword prefix to determine a possible codeword length. We do not traverse a Huffman tree bit-by-bit, but read at each stage as many bits as possible. Thus, we begin with reading l_{min} bits.If the codeword length of the current codeword equals l_{min}, than the corresponding symbol is output. If the symbol length can be identified from bits already read, next l_b bits are read, otherwise, next $l_{b_{min}} - l_{min}$ bits are read. The process is repeated until a symbol is output. This process can be implemented with a series of tables. Every table record consists of two fields. One field is used to indicate, whether a codeword has been read or the next table has to be used. The second field contains either a symbol, corresponding to a codeword or a pointer to the next table. We look up the value of the first l_{min} bits in the first table. If the bits read so far constitute a codeword we output the corresponding symbol, otherwise we read the next bit sequence.

```
Procedure Read_Next_Symbol( )

begin
bitval:=get_next_bits(l_min );
while (table[bitval].length != DIRECT_DECODE)
begin
  next_length=table[bitval].length;
  table=table[bitval].next_table;
  bitval:=get_next_bits( next_length );
end

output table[bitval].value;
end
```

As an example consider the following code:

```
a=000;          f=0110;         k=10101;
b=0010;         g=0111;         ...
c=0011;         h=1000;         u=11111;
d=0100;         i=1001;
e=0101;         j=10100;
```

In this case $l_{min} = 3$. Supposed, we read the sequence 000101000111. We read 3 first bits and output a.Then we read the next 3 bits. The control is transferred to the next table, containing suffixes of codewords, starting with 101. We read next 2 bits and output j. Next, we read 3 bits and transfer control to the table containing suffixes of codewords, starting with 011. We read the next bit and output g.

4.2 Analysis of the algorithm.

Every Huffman decoding algorithm, independently of the decoding method employed, should store a table (or a similar data structure) with at least n records. These records set relation between n codewords of the Huffman code and the corresponding symbols of the source alphabet. In our algorithm, more than n records have to be stored. We shall call the table records, not corresponding to a source symbol *additional records*.

The number of records in all tables does not exceed the number of nodes in the Huffman tree, therefore $2n - 1$ is an upper bound for the table records and an upper bound for the additional records is $n - 1$. Let S be the set of codeword prefixes b, such that length of b equals to the length of some codeword in the Huffman code. Let $length(b)$ be the length of b. Then the number of additional records can be computed as

$$\sum_{b \in S} 2^{l_{b_{min}} - length(b)} + 2^{l_{min}} - N$$

Our algorithm can be used for any Huffman code, but it is effective (in terms of small number of "additional records" and number of table look-ups) only for canonical codes. The reason for this is that in a canonical code codewords with identical prefixes tend to have the same length. For instance, in the example above we have 7 additional records, i.e. records pointing to further tables, in the first table.

The described algorithm uses essentially less space than classical Huffman tree approach and allows for faster decompression. In case of the classical Huffman decoding the number of operations is proportional to an average code length and required space is proportional to $2N - 1$. The estimate for the number of records in our algorithm was given above. The practical experiments show, that the total number of records in all tables is between $1.1N$ and $1.2N$.Our algorithm is also faster then the $sk - tree$ decoding, for we always read groups of bits and not individual bits. For the same reason the total number of additional records is smaller than the number of nodes in an sk-tree, as some of the internal nodes in an sk-tree do not have corresponding records.

Every table record contains an additional field, indicating whether further tables should be used. This can lead to sufficient overhead in case of large source alphabets. Instead of this we can "encode" this information in the first field, but we will have to pay for it with one or two additional operations at every table look-up.

The suggested algorithm reduces the number of necessary bit manipulations.

4.3 Byte-oriented decoding

Further we suggest a special finite-automaton-based data structure, which allows reading of up to i bits from the input stream without using bit-oriented operations. Combined with our algorithm it allows efficient decoding without bit-oriented operations. This finite automaton has states corresponding to all binary sequences b with length between 1 and 8. The input alphabet consists of integers between 1 and i, the output alphabet consists of pairs (v, j), where j is an integer between 1 and i. States of the automata correspond to the "rest" of current byte, that is not yet processed. Input l indicates the number of bits to be read, v is the value of read bits, j is the number of bits that should be read at the next step. Supposed the FSM is in a state s, corresponding to the bit sequence of length k and input integer is l. If $l > k$ the automaton outputs pair $(v, l-k)$, where v is the value of b shifted $l-k$ bits left. If $l \leq k$, the automaton outputs pair $v, l-k$, where v is the value of b shifted $k-l$ bits right. Thus in the output pair (v, j) the first component v is the value of "as many as possible" read bits from the current byte and j indicates the number of bits which should be read from the next bytes.

The initial state of this finite-state machine corresponds to the first byte in the input stream. Supposed we want to assign the value of next l bits to variable val. The pseudocode description of this process is given below. The finite automaton is represented as a two-dimensional array of records with fields v, j and next_state that correspond to an output pair (v, j) and the next state of the finite automaton respectively.

```
Procedure Get_Next_Bits(l)
val:=0;
repeat
  val:=val+fsm[b][l].v;
  l:=fsm[b][l].j
  if (l > 0)
    b:=value of the next byte from the input stream;
  else
    b:=fsm[b][l].next_state;
until (l = 0)
```

The finite-state automata described above has 512 states. The structure, required to represent this automaton, requires $O((511) * i)$ bytes, where i is the maximum number of bits, which has to be read from the input stream. Thus, if we want to read, for instance, up to 8 bits, $3 * 511 * 8 = 12264$ bytes are needed.

4.4 Experimental results.

We've tested the described method on files from Calgary compression corpus (see [1]). The Calgary compression corpus consists of two books (book1 and book2), six papers (paper1 through paper6), one bibliography (bib) and three programs (progc,progp and progl). Non-ASCII files are represented by a black-and-white picture (pic) and two object files (obj1 and obj2). As a source alphabet a sequences of two characters were considered. The results are presented in the table below. The second column gives the number of symbols in the source alphabet. The third and fourth column give information about the Huffman code: the number of nodes in a Huffman tree and average number of bits processed. The next three columns contain information about the decoding with consequent tables method. The fifth and sixth columns give the total number of records and the number of additional records. The seventh column gives an average number of look-ups per symbol.

The results of practical experiments with files from the Calgary compression corpus (see [1]) are given in the table below. We can see, that, compared with conventional Huffman tree decoding 40 to 45 percent of memory re-

File	Alphabet size	conventional tree		Consequent tables		
		Tree size	Number of bit manipulations per symbol	Total number of records	Additional records	Number of table look-ups per symbol
bib	1323	2645	8.58	1586	263	2.10
book1	1634	3267	8.14	1916	282	2.06
book2	2739	5477	8.56	3128	389	2.11
obj1	3064	6127	9.17	3562	498	2.53
obj2	6170	12339	8.93	6988	818	2.24
paper1	1353	2705	8.64	1660	307	2.13
paper2	1122	2243	8.13	1418	296	2.09
paper3	1011	2021	8.23	1270	259	2.09
paper4	705	1409	8.13	928	223	2.12
paper5	812	1623	8.43	958	146	1.92
paper6	1218	2435	8.61	1592	374	2.10
pic	2321	4641	2.39	2794	473	1.37
progc	1443	2885	8.80	1774	331	2.12
progl	1032	2063	8.00	1242	210	2.23
progp	1254	2507	8.06	1524	270	2.34

sources can be saved, and essentially less operations are used. The algorithm and data structure described in this work allow fast decoding of Huffman codes, that can be efficiently implemented without using bit operations.

References

[1] T.C. Bell, J.G. Cleary, I.H. Witten, "Text Compression", Prentice Hall, Englewood Cliffs, NJ,1990.

[2] Y. Choueka, S.T.Klein, Y.Perl, "Efficient variants of Huffman codes in high-level languages", Proc. 8th ACM-SIGIR Conference on Information Retrieval, Montreal, Canada, ACM, New York, 1985, 122-130.

[3] Connell J.B., "A Huffman-Shannon-Fano Code", Proc. of IEEE 61,7(July),1973, 1046-1047.

[4] J.L. Gailly, "Gzip program and documentation", 1993, ftp://prep.ai.mit.edu/pub/gnu/gzip.

[5] D.A.Huffman, "A method for construction of minimum redundancy codes", Proc IRE,40(1951),1098-1101.

[6] V.Iyengar,K.Chakrabarty, "An efficient finite-state machine implementation of Huffman decoders", Information Processing Letters, 64(6),1997, 271-275.

[7] Shmuel T.Klein, "Space- and time- efficient decoding with canonical Huffman trees", 8th Annual Symposium on Combinatorial Pattern Matching,Aarhus,Denmark,30 June-2 July 1997, Lecture Notes in Computer Science,vol. 1264, 65-75.

[8] Moffat A. and Turpin A., " On the Implementation of minimum redundancy prefix codes", IEEE Transactions on Communications, vol. 45 No. 10, 1200 - 1207.

[9] Nekrich Y., "On efficient decoding of Huffman codes", Technical Report No. 85190-CS, Department of Computer Science, Bonn University, April 1998.

[10] Sieminski A., "Fast decoding of the Huffman codes", Information Processing Letters, 26(1988), 237-241.

[11] Schwartz E.S. Kallick B., "Generating a canonical prefix encoding", Communications of the ACM 7(1964), 166-169.