Decoding of Canonical Huffman Codes with Look-Up Tables

Yakov Nekritch *

Abstract

We describe an efficient algorithm for decoding canonical Huffman codes with look-up tables. Furthermore, a connection between the probability of a symbol and the length of its codeword in a Huffman code is investigated. We design a method for estimating probabilities of codewords in a given Huffman code. We show how this method can be used to estimate a “good” size for the look-up table used in the decoding.

*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.


The traditional tree-based decoding of Huffman codes requires sequential examinations of single bits and is relatively slow. More efficient methods examine values of bit sequences at each algorithm step (see [9], [11], [2], [8]).

In this paper we describe a new method of decoding canonical Huffman codes with look-up tables. Besides that, some ideas, concerning the estimation of table size from the code parameters are discussed.

2 Decoding with look-up tables

In a case of decoding with look-up tables we read a number of bits at each step of decoding process. We look-up the value of read bit sequence in a table and, if this bit sequence contains a codeword, we output the corresponding symbol. Otherwise we proceed with the decoding, using the next look-up table or using some other method. The same principle is used in decoding of fixed-length codes: we read the codeword, look-up the symbol corresponding to this codeword and output this symbol. However in case of Huffman decoding the process is more difficult, as codewords of a Huffman code have variable length. Supposed the maximum codeword length in a Huffman code is $l_{max}$, then we will need a table with $2^{l_{max}}$ entries, if we want to decode every codeword in one step. This means high memory requirements, especially for codes with a large number of codewords. Besides that, the time required for initialization of such table can be unnecessarily high and may become the bottleneck of an algorithm in case of a small source file.
2.1 Look-up tables of fixed size

In a method, proposed by Choueka and Klein ([8]) we read bit strings of fixed length \( k \) at each step of the algorithm. The read bit 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 continue decoding using the next look-up table, corresponding to the prefix of a not yet decoded codeword, contained in the string. The number of tables 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 \( 2^k \) records in each table, 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 \( O(2^k(N - 1)) \) as for every internal tree node a table with \( 2^k \) records must be stored. This approach results in high speed decompression, provided that \( N \) and \( k \) are not so large. This technique can also be used for bytewise processing of input stream, in case of \( k = 8 \) or \( k = 16 \). In this case bit operations would not be used at all. Avoiding bit operations can simplify implementation in some high-level languages and results in faster decoding.

The drawback of this approach is high memory requirement for the large alphabet sizes \( N \). For instance in case of \( k = 8 \) and \( N = 2000 \) the space required is proportional to 512000 and for storing the tables several Megabytes would be required. In case of large source alphabets the space requirements can become prohibitive. Besides that, the time necessary for the initialization of the tables can essentially increase the overall decoding time.

A similar approach to Huffman decoding is described in [11].

2.2 Canonical Huffman codes

Canonical codes are a subclass of Huffman codes, described by Conell[3] and Schwartz and Kallick[12]. 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 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.

A property of canonical Huffman codes essentially simplifies the decoding
process. Once the length of the current codeword is known, it can be de-
coded 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 in-
tegers and the codewords themselves are binary representations of consecutive
integers. Let $l(b)$ be the length of the prefix $b$ and $\text{first}_l$ 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 $b_n + \text{first}_l$.

An efficient decoding algorithm for canonical Huffman codes is described
in [9]. Let $m_l$ be the value of the smallest code of length $l_i$ bit-shifted $l_{\text{max}} - l_i$ bits left. $m_l$ will be further called a minimal value of length $l_i$. Let $s_l$ be an integer value of some binary string with length $l_{\text{max}}$, prefix of which is a codeword with length $l$. Then $m_l < s_l < m_{l_2}$, iff $l_1 < l < l_2$.
Therefore, we can read at once $l_{\text{max}}$ bits, and then determine the length of
the current codeword by comparisons with minimal values. The index of the
current codeword can be computed as $\text{RightShift}(s_l, l_{\text{max}} - l) + \text{first}_l$, where
$\text{RightShift}(a, i)$ is a value of $a$ bit-shifted $i$ bits to the right. This approach
also allows to avoid bit-by-bit processing of encoded data and minimize use
of bit operations, thus leading to very fast decoding.

Moffat and Turpin also suggest in [9] a Huffman decoding method with
look-up tables. In this method we use the look-up table to determine or limit
the possible codeword lengths for codewords with prefix $b$. The look-up table
contains the minimum length $b_{\text{min}}$ of a codeword with prefix $b$. We start with
reading $l_{\text{max}}$ bits from the input stream, where $l_{\text{max}}$ is the maximum codeword
length. Then we examine the value $b$ of first $l_{\text{first}}$ read bits in the look-up
table. If $b_{\text{min}}$ is not bigger than $l_{\text{first}}$, then the symbol corresponding to the
current codeword is output. Otherwise we determine the codeword length by
comparing the value of current codeword with all the minimal values, starting
from $b_{\text{min}}$. Once the length of the current codeword is known, we compute
the codeword index as described above. The pseudocode description of the
algorithm is given below. Here and further array minlength[$s$] contains the
minimum length of a codeword with prefix $s$, first[$l$] is the index of the first
Procedure TableLookUp()

begin
V := value of the next l_max bits from the input string;
v := value of the first l_first bits of V;
length := min length[v];
while (minval[length] <= v)
  length := length + 1;
length := length - 1;
index := first[length] + RightShift(V - minval[length], l_max - length);
output(symbol[index]);
end;

In [10] we described a modification of the above method. If the codeword
is a prefix of the read bit string, the corresponding symbol is output directly.
Otherwise, we use the prefix of a codeword to determine the codeword length
or limit the range of possible codeword lengths. We read l_first bits and if
the length of the current codeword is less than l_first we output the symbol,
corresponding to the current codeword. Otherwise we use the value b of
the first l_first bits to determine a possible codeword length. If the codeword
length / can be uniquely determined, i.e. all codewords with prefix v have the
same length, we compute the codeword index and output the corresponding
symbol. If maximum length of codewords with prefix b l_max is less then
l_first plus a small value (a criteria l_max - l_first \leq 3 was used) we use an
additional look-up table. We then read next l_max - l_first bits and output the
symbol, that corresponds to the read bit sequence in the additional table.
Otherwise we read l_max - l_first bits and compare their value with the minimal
values until the codeword length is determined. This algorithm is described
in procedure TableLookUp1, table[v].flag indicates what kind of decoding
should be used for the bit string v. The value SHORT_CODE means that
there is a codeword, which is a prefix of \( v \), SAME_LENGTH means that all codewords with prefix \( v \) have the same length, ADD_TABLE indicates, that a small additional table should be used and LONG_CODE means that length should be determined as in [9].

Procedure TableLookUp1( )

begin
  \( v := \) value of the next 1_first bits;
  if (table[v].flag = SHORT_CODE)
    output (table[v].value);

  if (table[v].flag = SAME_LENGTH)
    shift := value of the next table[v].length bits;
    output (symbol[table[v].value + shift]);

  if (table[v].flag = MEDIUM_CODE)
    index := value of the next table[v].length bits;
    output (secondtable[table[v].value + index]);

  if (table[v].flag = LONG_CODE)
    \( v := \) value of the next table[v].length bits;
    while (minval[length] < v)
      length := length+1;
      length := length-1;
      index := first[length] + (v-minval[length]);
    output (symbol[index]);

end

The methods, described above, are particularly efficient in case of large alphabets. In case of small codes, containing codewords with small codewords length they are less efficient, than method, described in the section 2.1. The reason for this is, that we can output more than one codeword at each algorithm step.
In the next section we describe decoding method which is efficient for different types of Huffman codes without causing high memory overhead.

3 Look-up tables for multiple codewords

We start decoding by reading $l_{first}$ bits from the input stream. The bit sequence $b$ contains zero or more codewords and, possibly, a prefix of some codeword. We output symbols, corresponding to the codewords, contained in the read bit sequence. If the string contains more than one codeword and the prefix $p$ of the next codeword, we look whether the length of this codeword can be determined, read the rest bits of the current codeword and compute its index, using the property of Huffman codes, described in Section 2. Supposed, $f$ is the index of the first codeword with prefix $p$ and $v$ is the value of the rest bits of the codeword, then, as codewords with the same length are consecutive integers and the corresponding symbol indexes are also consecutive integers, the index of the next codeword equals to $f + v$.

Otherwise, if no codewords were output and the bit string $b$ is a prefix of a codeword, we proceed with the decoding in the same way as in [10]. We use the value $b$ of the first $l_{first}$ bits to determine a possible codeword length. If the codeword length $l$ can be uniquely determined, i.e. all codewords with prefix $v$ have the same length, we compute the codeword index and output the corresponding symbol. If maximum length of codewords with prefix $b$ $l_{max}$ is less then $l_{first}$ plus a small value (a criteria $l_{max} - l_{first} \leq 3$ was used) we use an additional look-up table. We then read next $l_{max} - l_{first}$ bits and output the symbol, that corresponds to the read bit sequence in the additional table. Otherwise we read $l_{max} - l_{first}$ bits and compare their value with the minimal values until the codeword length is determined.

Finally, we shift the bit pointer $l_{tot}$ bits to the right, where $l_{tot}$ is the total length of the codewords, processed at this step, and start the decoding from the beginning.

The decoding process can be described in pseudocode as follows:

```
Procedure TableLookupModified( )
begin
    bitval:=value of the next l_first bits;
```
output symbols, corresponding to codewords in bitval;
if (bitval is a prefix of a codeword)
    if (length of current codeword can be
determined from bitval)
        read the rest bits of the current codeword;
        compute the codeword index and output it;
    else
        read (l_max - l_first) bits;
        determine the length of the current codeword;
        compute the codeword index and output it;
end

As an example consider the following code:

```
'1' = 000;   '6' = 0110;    '11' = 10101;
'2' = 0010;  '7' = 0111;   ...
'3' = 0011;  '8' = 1000;   '21' = 11111;
'4' = 0100;  '9' = 1001;
'5' = 0101;  '10' = 10100;
```

Supposed the length \( l_{\text{first}} \) of an initial look-up string is 8. Consider the stream of binary codewords 00000101010111001010000... . We read the first ten bit string 00000101. It contains the codewords 000 and 0010. We output symbols “1” and “2”, corresponding to these codewords. The next 8-bit string is 10101111 and we output “11”. The rest of this string, 011 is a prefix of a 4-bit codeword, therefore we read the next bit 1, and output the symbol, corresponding to 0111. Then we read the the string 00101000 and output two symbols “2” and “8”, and so on.

For implementation of this method we use a table with \( 2^{l_{\text{first}}} \) entries. A record for string \( b \) contains the symbol, corresponding to the codewords, contained in \( b \), an indication, whether further bits should be read and the number of bits processed at the step. Table I shows the beginning of the table for the code, described above in case of \( l_{\text{first}} = 8 \). The String gives a binary representation of an initial string. Field NUM specifies the number of codewords, contained in \( b \). Field SYMSTR lists the symbols, corresponding to codewords in \( b \). Field Next Bits specifies the number of additional codewords
to be read. It is used, when the end of a codeword prefix $p$, such that the length of the next codeword can be determined, or when the whole string $b$ is a prefix of a codeword. If no additional bits should be read NextBits equals to 0. Otherwise NextBits contains information about the number of bits to be read and the method to be used for decoding of the current codeword. If NextBits is less than 40, then all codewords with prefix $p$ have the same length and the field FirstInd specifies the index of the first codeword with prefix $p$. If the value of the next NextBits bits is $v$, then the index of the next codeword can be computed as FirstInd + $v$. If NextBits has value between 40 and 80, then an additional table should be used. In this case FirstInd specifies the index of this table. The value of NextInd above 80 indicates, that we should read the next $l_{max} - l_{first}$ bits and determine the codeword index by comparisons with the minimal values. The field ProcBits specifies the number of bits processed at this step. For instance the string 00000000 contains two codewords, corresponding to symbols 1 and 1, the rest of the string 00 is not enough to determine the length of the next codeword and the number of bits processed before we proceed to the next initial string is 6. For string 00000001 again two symbols 1 and 1 will be output. However 01 is sufficient to determine the length 3 of the next codeword. Hence we will read the next bit $v$ and compute the next symbol as $4 + v$. The total number of processed bits in this case will be 9.

<table>
<thead>
<tr>
<th>String</th>
<th>NUM</th>
<th>SYMSTR</th>
<th>NextBits</th>
<th>FirstInd</th>
<th>ProcBits</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>2</td>
<td>1,1</td>
<td>0</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>00000001</td>
<td>2</td>
<td>1,1</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
<td>1,1</td>
<td>0</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>......</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101011</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>10101100</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>-</td>
<td>5</td>
</tr>
</tbody>
</table>

The difference of this method from the method, described in [9] is that we directly output symbols, corresponding to the codewords, contained in the...
first bit string without any further computations. Besides that, we output all symbols, contained in the first bit string and not just the first one. It allows an efficient decompression of codes with short codewords.

### 3.1 Table Construction

The construction of the look-up table with multiple codewords for a Huffman code can itself become a time-consuming problem. However, in case of canonical Huffman codes, the table can be constructed in linear time in the number of records. In this subsection we will outline the process of table construction.

We will call the symbol, corresponding to the zero codeword the first symbol. The first record in the table corresponds to a sequence of zeroes. It consists of zero or more codewords, corresponding to the first symbol and the “rest” sequence. We can easily compute the symbol string SYMSTR and the number of symbols, NUM, for the first record.

Now we will show how we can construct the symbol string for the \( k+1 \)-th record from the symbol string for \( k \)-th record in a constant time. Supposed the \( k \)-th record consists of \( r \) codewords and the index and length of the \( i \)-th codeword and the codeword itself are stored in index\([i]\), length\([i]\) and codeword\([i]\) respectively. First, we will presume that \( k \)-th record contains no “rest” sequence and consists only of \( r \) codewords. To construct the symbol string for the next codeword we simulate incrementing the value of \( k \)-th record in terms of codewords. The additional array maxbitstr\([r]\) stores the values of \( l << \text{length}[r] \). We start by incrementing index\([r]\) and codeword\([r]\) by 1. We check, whether codeword\([r]\) is less then maxbitstr\([r]\). If this is not the case, we decrement \( r \) by 1 and increment index\([r]\) and codeword\([r]\). This process is repeated until maxbitstr\([r]\) is greater than codeword\([r]\). Then, we check whether the length of the codeword\([r]\) has changed and fill the rest of the bit string with first symbols.

The pseudocode description of this process is given below. Function len\((i)\) computes the length of the codeword with index \( i \).

```python
begin
    while ( (codeword[r]+1) = maxbitstr[r] )
        restlen:=restlen+length[r];
        length[r]:=firstlen;
```
Now we will consider the general case, when k-th record consists of r code-
word and the “rest” bit sequence pref. We store the values of the first
codeword with prefix pref, the index of the first codeword with prefix pref
in codeword[r+1] and index[r+1]. In this case codeword[r+1] and index[r+1]
will be incremented by different values.

The pseudocode for the general case is given below. Step is an additional
variable for computing the values of index[r+1] and codeword[r+1] for the
next record. If the value of the variable step is greater than 1, it indicates
that the current record contains the “rest” bit sequence.

begin
if (step > 1)
    index[r]:=nextindex(index[r],step);
codeword[r]:=nextcodew(codeword[r],step);
if (codeword[r] == maxbitstr[r])
    r:=r-1;
while ( (codeword[r]+1) = maxbitstr[r] )
    restlen:=restlen+length[r];
    length[r]:=firstlen;
    index[r]:=0;
    restlen:=restlen + length[r];
    index[r]:=index[r]+1;
end;
length[r]:=len(index[r]);
if (len(index[r]) > len(index[r] - 1))
    codeword[r]:=LeftShift((codeword[r] + 1), length[r] - len(index[r] - 1));
    maxbitstr[r]:=LeftShift(1, length[r]);
else
    codeword[r]:=codeword[r] + 1;
restlen:=restlen - length[r];
if (restlen > 0 )
    r:=r + (restlen/firstlen) + (restlen%firstlen > 0);
    if (restlen%firstlen > 0)
        step := LeftShift(1,restlen%firstlen);
    else
        step:=1;
else
    step:= LeftShift(1,-restlen);
end;

Obviously the values of table fields SYMSTR, NUM, NextBits and FirstInd can be computed from variables index[], step and r.
The computation of functions nextcodew(c, j), finding the j-th codeword after the codeword c, and nextindex(c, j), finding the index of this codeword can be time consuming. Using the prefix information is not always the best option, sometimes it is better to simply shift the bit pointer and restart the decoding process by reading the new initial string. For this reason, our algorithm computes the nextcodew( ) function only for long prefix strings. We use the following criterion. If a record contains more than one codeword and the length of the end of the last codeword, beginning with pref, does not exceed 7, the functions nextcodew(c, j) and nextindex(c, j) are computed. Otherwise, we just increment values of index[r+1] and codeword[r+1] by 1, without using information, contained in pref.

4 Estimating table size from code parameters
An important issue for the algorithm, described in the previous section is selection of an appropriate table size. Choosing to construct too big table
for a small file would mean an unnecessary time and memory space overhead. On the other hand, choosing to construct a small table for a big file results in a less efficient decoding.

In many applications the size of decoded file is not known to the decoder. In case of canonical codes, as was mentioned in section 2, it is sufficient to transmit the maximal codeword length and number of codewords for each length.

In this section we will show how the file size can be estimated from the specification of a canonical Huffman code. The results also describe the connection between the probability of the least frequent source symbol and its codeword length in a Huffman code.

In this section the following notation will be used. We presume that a Huffman code is specified as a tuple \(< n_1, n_2, \ldots, n_m >\) where \(n_i\) is the number of codewords with length \(i\). For symbol \(a\) frequency \(f_a\) is the number of occurrences of \(a\) and \(f_{tot}\) is the total number of occurrences of all symbols, which is equivalent to the length of the source file. Probability \(p_a\) is defined as \(f_a/f_{tot}\).

We will also consider the Huffman tree corresponding to the Huffman code. The weight \(f_a\) of a leaf \(a\) equals to the frequency of a corresponding symbol. The weight of a node equals to the sum of weights of its descendents. We will say, that node \(N\) is on level \(l\), if \(\text{depth}(a) = l\). The total weight \(F\) of a Huffman tree is the weight of its root. The total weight equals to the size of a source file. The Huffman tree and Huffman code notations will be used alternatively.

We consider the problem of determining the minimal size of the source for a given Huffman code. In the further description a property of Huffman tree will be used extensively.

**Property 1** For any two nodes \(a, b\) in a Huffman tree. If \(\text{depth}(a) < \text{depth}(b)\), then \(p_a \geq p_b\).

*Proof:* Supposed, that \(p_a < p_b\). Then we can swap the nodes \(a\) and \(b\) and get the tree \(T'\), such that the weighted path length of \(T'\) is smaller than weighted path length of \(T\). Therefore \(T\) is not a Huffman tree.

In the following lemma we presume that the smallest source symbol has frequency 1. We will show that a certain frequency distribution for a given Huffman code corresponds to the smallest source file.
Lemma 1 A canonical tree $< n_1, n_2, \ldots, n_m >$ has minimal weight $F$ if:
1) all leaves on level $m$ have weight 1
2) all leaves on level $i$ have weight equal to the weight of the node with maximal weight on level $i+1$.

Proof: We will denote the total weight of the above tree $F_{min}$. Supposed, there is a tree $T'$ such that the total weight of $T'$ is smaller than total weight of $T$, then there is a leaf $a'$ such that for any leaf $b'$ with $\text{depth}(b') > \text{depth}(a')$ $f_{b'} = f_b$ and $f_{a'} < f_a$. Then $f_{a'}$ has weight, smaller than a weight of some node $N$, such that $\text{depth}(N) = \text{depth}(a') + 1$. Hence, by property 1, $T'$ is not a Huffman tree.

Corollary 1 For a canonical code $< n_1, n_2, \ldots, n_m >$ the minimal size of the source file is $F_{min}$.

The above lemmas allow to compute the lower bound on the size of the source file for a given canonical Huffman code. It can be shown, that the value of $F_{min}$ can be computed in $O(m^2)$, where $m$ is the maximum codeword length.

Generally speaking, there is no upper bound for the source file size. It can be demonstrated with an example of a binary source code $a, b$, such that $f_a = 1$ and $f_b = m$. Independently of the value of $m$ (and of the source file size, which equals to $m + 1$) the Huffman code remains the same (two codewords with length 1). However we can put an upper bound on the weight of the nodes with the same depth $d$ in case, when the nodes do not have a maximum depth and there are more than two nodes with depth $d$.

Lemma 2 Supposed, that nodes $N_1$ and $N_2$ in a Huffman tree $T$ have the same length $d$, and there is at least one another non-leaf node on level $d$. Then $f_{N_1} \leq 3 \times f_{N_2}$

Proof: Supposed, that $N_1 \geq 2 \times N_2$. There is at least one node $N_i$ with two sons $N_3$ and $N_4$. Without changing the weighted path length of $T$ we can reconstruct the $T$ in such way, that $N_3$ and $N_2$ have the same father $N_0$. As $f_{N_3} \leq f_{N_2}$ and $f_{N_4} \leq f_{N_2}$ $f_0 = f_{N_2} + f_{N_3} + f_{N_4} \leq 3 \times f_{N_0} < f_{N_i}$. As $\text{depth}(N_0) > \text{depth}(N_1)$ $T$ is not a Huffman tree. Using the lemmas listed below, we can compute the probability, that the codeword length is not less than $l$. 
Lemma 3 If non-leave nodes $N_1$ and $N_2$ have the same depth, than $f_{N_1} \leq 2 \times f_{N_2}$

Proof: Otherwise at least one son of $N_1$ would have had weight bigger than $f_{N_2}$.

With lemmas 2 and 3 we can put an upper bound on the weights of leaves on the same level $d$, if there is more than one non-leave node on level $d$.

Using the following lemmas we can compute maximum probability of the fact, that the codeword length is bigger than $l$.

In the same way as Lemma 1, the following lemma can be proved:

Lemma 4 A canonical tree $\langle n_1, n_2, \ldots, n_m \rangle$, such that the minimal weight of a leaf is $f$ has minimal weight $F$ if:
1) all leaves on level $m$ have weight $f$
2) all leaves on level $i$ have weight equal to the weight of the node with maximal weight on level $i+1$.

Corollary 2 For a given canonical tree $\langle n_1, n_2, \ldots, n_m \rangle$, such that the minimal leaf weight is $f$ the minimal tree weight is $f \times F_{\text{min}}$.

Proof: The weight of every leaf in the tree of Lemma 2 is $f$ times bigger than the weight of the corresponding leaf in the tree of Lemma 1.

Corollary 3 For a given canonical code $\langle n_1, n_2, \ldots, n_m \rangle$, the maximal probability of the least probable symbol is $1/F_{\text{min}}$.

Corollary 4 For a given canonical tree $\langle n_1, n_2, \ldots, n_m \rangle$ the maximal total probability of leaves on the level $m$ is $n_m/F_{\text{min}}$.

Let an $i$-truncated tree $\langle n_1^{(i)}, n_2^{(i)}, \ldots, n_i^{(i)} \rangle$ be a tree constructed from tree $\langle n_1, n_2, \ldots, n_m \rangle$ by replacing all nodes on level $i$ with leaves of equal weight.

Statement 1 For a given canonical tree $\langle n_1, n_2, \ldots, n_m \rangle$ the total probability of leaves with length bigger than or equal to $i$ equals to $n_i^{(i)} / F_{\text{min}}^{(i)}$, where $F_{\text{min}}^{(i)}$ is the minimal total weight of an $i$-truncated tree $\langle n_1^{(i)}, n_2^{(i)}, \ldots, n_i^{(i)} \rangle$.

Statement 1 puts an upper bound on the total probability of codewords with at least $i$ bits.
4.1 Algorithm for estimating file size.

The minimal size $F_{\text{min}}$ from Lemma 1 can be computed in $O(m^2)$, where $d$ is the depth of Huffman tree. To compute $F_{\text{min}}$ we set the weights of all leaves on the deepest level to 1. Then we repeat the following step for all tree levels $k$. We find the maximal node weight $f_{\text{max}}^{k+1}$ for all non-leave nodes on level $k + 1$ and set the weight of all leaves on level $k$ to $f_{\text{max}}^{k+1}$.

We will say that node $N$ is an $i$-node, if all leaves, that are descendants of $N$ have depth $i$. We will say that node $N$ is an $i$-mixed node if all leaves, that are descendants of $N$ have depth greater than or equal to $i$. Obviously, all $i$-nodes on the same level $k$ have the same weight. In a canonical Huffman tree there is at most one $i$-mixed node on any level $k$. Therefore the nodes on any tree level $k$ can take on less than $2m$ different values and the weight of the heaviest node on level $k$ can be computed in $O(m)$. Hence $F_{\text{min}}$ can be computed in $O(m^2)$. On each algorithm step at most $m$ values of $i$-nodes and at most $m$ values of $i$-mixed nodes have to be stored, therefore the algorithm requires $O(m)$ space.

However in real life distributions codewords with the same length rarely have the same frequency. To compensate for this fact and to get a more realistic size estimate, we set the weights of nodes on level $k$ to $c \times f_{\text{max}}^{k+1}$, with $c > 1$. We can also make $c$ dependent from the number of nodes on level $k$. For our experiments we have chosen $c = 3$ if the number of nodes is above 30, $c = 2$ if the number of nodes is above 8 and $c = 1$ otherwise. The results of experiments are listed in the table below. We have also compared our results with the estimates, based on the work of Katona and Nemetz([7]).

As follows from the theorem 2 in [7], the probability of a leave on level $k$ is not bigger than $F_k^{k+1}$, where $F_i$ is the $i$-th Fibonacci number. Therefore, for a maximum codeword length $m$ the file size estimate will be $F_{m+1}$. In the table below we list the results of prediction for the files of Calgary compression corpus. The minimal estimate gives much better results than Fibonacci number estimate. The results for our “realistic” procedure are even more close to actual file sizes and can be used in real applications (see the next section for an example of such application). We stress that the data corpus used in our experiments consists of different file types and no information about the frequency distribution of symbols or about the frequency of the least frequent symbol was used in this estimate. The worst estimates (for book2 and prog1) can be explained by the fact, that the frequency of the least frequent symbol
exceeds 1. Our conclusion is that we can achieve a good estimate of file size, based on lemma 1, especially if additional information about the frequency distribution is available.

Table II

| File  | Fibonacci minimal "realistic" Real size |
|-------|----------------------------------------|------------------|
|       | numbers | estimate | estimate | size |
| bib   | 2584 | 7752   | 38369    | 111261 |
| book1 | 17711  | 66392 | 270723    | 768771 |
| book2 | 2584 | 8982   | 46077    | 610865 |
| obj1  | 610 | 2169   | 46934    | 21504 |
| obj2  | 1597 | 7856   | 131800   | 246814 |
| paper1 | 1597 | 5142   | 30718    | 53161 |
| paper2 | 2584 | 8901   | 42429    | 82199 |
| paper3 | 987  | 3138   | 14336    | 46526 |
| paper4 | 610  | 2140   | 7168     | 13286 |
| paper5 | 610  | 2100   | 7680     | 11954 |
| paper6 | 1597 | 4834   | 23039    | 38105 |
| pic   | 4181  | 25471  | 135056   | 513216 |
| progc | 987  | 2728   | 13374    | 39611 |
| progl | 987  | 3183   | 15359    | 71646 |
| progp | 1597 | 6462   | 25595    | 49379 |

5 Experimental results

We have tested algorithm of section 4 on files from Calgary compression corpus (see [1]). It 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). The source alphabets for all files consist of single characters, occurring in the corresponding source file.
The results are listed in Table III. The second column in the table specifies the size of the source alphabet. As a speed criterion we have used the average number of decoded symbols per read bit sequence. The results for the look-up method from [9] and for the modified look-up method from the section 3 are listed in the third and the fourth columns. In both cases the look-up tables contain 12-bit strings.

We can see that improved look-up method leads to more then twofold reduction in the number of bit reading operations. For the look-up method this criterion can not exceed 1. For the improved look-up method number of symbols per bit string ranges from 1.4 to 6.3. The most impressive advantage is achieved for the pic file, the Huffman code for which contains several very short codewords.

In the table IV we have listed the results of decoding with variable table size. The algorithm from the section 4.1 is used to predict the length of the source file. We set table size to 256 (8-bit strings), if the predicted file size does not exceed 10000 Bytes. We use 10-bit strings, if predicted file size does not exceed 100000 Bytes. Otherwise we set table size to 4096 records. Using this variation of the algorithm we can reduce memory resources for small files, with only small reduction in speed. We list the size of decoding table used in modified look-up method in column 4. In column 2 we list file sizes in thousands of bytes. Choice of the table size based on code parameters allows us to spare memory resources and table construction time for small files.

6 Future work

We have constructed an algorithm for estimating the source file size from the specifications of canonical Huffman code. This algorithm makes no assumptions about the probability distribution of the source symbols. For instance, probabilities of words in large texts can be approximated by Zipf distribution \( p_i = 1/(iH_n) \), where \( H_n \) is the \( n \)-th harmonic number (see [13]). In other cases (for instance, in image compression) symbol probabilities can be approximated by geometric distribution. Using this information to compute the coefficient \( c \) could further improve our estimates.

In some applications several Huffman codes are used to to encode alternating symbols. For instance, in gzip compression utility (see [4]) two different
codes are used to compress results of LZ77 compression. One code is used for literals and length components of duplicated string pointers and another one for distance components of duplicated string pointers. The decoding method, described above can be easily extended to that situation.

**References**


Table IV

<table>
<thead>
<tr>
<th>File</th>
<th>File size</th>
<th>look-up table size</th>
<th>Modified look-up table size</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111.2</td>
<td>0.9996</td>
<td>1024</td>
</tr>
<tr>
<td>book1</td>
<td>768.7</td>
<td>0.9987</td>
<td>4096</td>
</tr>
<tr>
<td>book2</td>
<td>610.8</td>
<td>0.9985</td>
<td>1024</td>
</tr>
<tr>
<td>obj1</td>
<td>21.5</td>
<td>0.9998</td>
<td>1024</td>
</tr>
<tr>
<td>obj2</td>
<td>246.8</td>
<td>0.9969</td>
<td>4096</td>
</tr>
<tr>
<td>paper1</td>
<td>53.1</td>
<td>0.9996</td>
<td>1024</td>
</tr>
<tr>
<td>paper2</td>
<td>82.2</td>
<td>0.9988</td>
<td>1024</td>
</tr>
<tr>
<td>paper3</td>
<td>46.5</td>
<td>0.9997</td>
<td>1024</td>
</tr>
<tr>
<td>paper4</td>
<td>13.3</td>
<td>0.9993</td>
<td>256</td>
</tr>
<tr>
<td>paper5</td>
<td>11.9</td>
<td>0.9997</td>
<td>256</td>
</tr>
<tr>
<td>paper6</td>
<td>38.1</td>
<td>0.9998</td>
<td>1024</td>
</tr>
<tr>
<td>pic</td>
<td>513.2</td>
<td>0.9991</td>
<td>4096</td>
</tr>
<tr>
<td>progc</td>
<td>39.6</td>
<td>0.9996</td>
<td>1024</td>
</tr>
<tr>
<td>progl</td>
<td>71.6</td>
<td>0.9998</td>
<td>1024</td>
</tr>
<tr>
<td>progp</td>
<td>30.8</td>
<td>0.9994</td>
<td>1024</td>
</tr>
</tbody>
</table>


