On Efficient Decoding of Huffman Codes

Yakov Nekritch [*]

Abstract

Two new algorithms for space- and time- efficient decoding of minimum redundancy codes with numerical sequence property are investigated. Both algorithms are particularly efficient in case of large source alphabets and limited memory resources. Practical experiments show a reduction of up to 50 % and more in the number of operations.

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

1 Introduction

Algorithm for construction of Huffman codes, i.e. prefix codes with minimum redundancy, was first presented by Huffman [8]. Huffman coding is nowadays one of the most popular and widely used compression techniques. It is a part of many data format specifications, such as JPEG [14] and PNG [1] image formats and deflate compression method [5]. For a number of applications, we are specially interested in minimizing time and memory resources for data decompression, while compression can take significantly more time and memory. An example of such application is a large information retrieval system, when compression is done only once and decompression should be performed for every information retrieval. Another example is data transmission, when the sender is a fast machine and the receiver is a much slower one and, possibly, has very limited memory resources.

Several decompression schemes for Huffman codes were offered. Table-lookup scheme allows decoding in constant time, however memory requirements are very high. Binary tree representation of Huffman codes allows decompression in time proportional to codeword length. Chung [3] also presents an algorithm working in $O(l)$ time. The code has a *numerical sequence property* if codewords of equal length form a consecutive sequence of integers. For any Huffman code an equivalent code, satisfying this property can be easily constructed. Moreover, some compression schemes (for instance, JPEG) use Huffman codes with numerical sequence property. Connell [4] constructed an algorithm decoding Huffman codes with numerical sequence property in $O(c)$ time, where c is the number of different codeword lengths. Another efficient decoding algorithm was presented by Klein [10]. In this paper a decoding scheme is presented allowing the decoding of Huffman codes in $O(\log c)$ time. Moreover this procedure requires very little memory and at most one reading operation for any codeword is necessary. Besides that, an improvement for table look-up decoding method is presented.

2 Previous methods

Throughout this paper the following notation will be used.Let l_{max} denote the length of the longest codeword and l_{avg} denote average codeword length.Let α denote a binary sequence and α^t denote a binary sequence of length t.Let

$I(\alpha)$ denote an integer, whose binary representation is α. We will say that binary sequence α is lesser than β if an integer corresponding to α is lesser than integer,corresponding to β. Let $minword(l)$ be the least codeword of length l in the given Huffman code. Let $first(l)$ denote the index of the first codeword of length l.

Decoding the Huffman codes using decoding tables was considered by Sieminski [12]. The idea is to build a table with entries, corresponding to all binary sequences of length l_{max}, where l_{max} is the length of the longest codeword. Each entry in the table, which corresponds to binary sequence $w\alpha$, where w is some codeword, contains information on symbol corresponding to w and the length of w. On each step of the algorithm we read a sequence of l_{max} yet unprocessed bits. We output symbol, corresponding to codeword, which is a prefix of this sequence. This construction allows decoding of Huffman code for a single codeword in constant time. Besides this, we don't have to process the input stream bit-by-bit, as in other methods.The main drawback of this method is high memory requirements.

The standard Huffman tree, which was used for constructing Huffman codes, can be used for decoding. To decode a single codeword, we start at the root node and traverse the Huffman tree according to the read bit sequence. At the beginning of the decoding the current node is the root node. We check if the current node is a leave. If yes, the decoding is over and we output the symbol, corresponding to the leave. Otherwise, we read the next bit from the input stream.If the next bit is 0, we assign the left child of the current node. If the next bit is 1 the current node is the right child of the current node. The average number of operations for the decoding of a single codeword is $O(l_{avg})$ In general case, we should transmit the structure of the chosen Huffman tree, which leads to increase of the size of compressed file. This increase is negligible in case of large size of compressed data, however it can be important in case of relatively small files to be compressed. Chung suggests a method of reducing storage overhead. Instead of transmitting the tree structure he suggests transmitting the array,which is sufficient for correct decoding of Huffman codes.

Another way to reduce the storage overhead is to choose for compression process the Huffman trees with some additional properties. The frequent choice in many applications is the *canonical* tree, defined by Schwartz and Kallick [13]. One property of the canonical tree is that if we list leaves leaves of the tree in pre-order, the leaves appear in non-decreasing order of their depth. A canonical tree can be unambiguously specified by the string $< n_1, n_2, ..., n_{l_{max}} >$, where n_i denotes number of codewords of lengths i, thus reducing the storage overhead for specifying the tree shape. Connell [4] describes the algorithm, allowing the decoding in $O(c)$ time. The algorithm uses the numerical property of canonical tree, i.e. the fact that the codewords of the same length are binary reprersentations of consecutive integers. Therefore after reading l bits we can determine, whether we read a codeword of length l, by comparing the integer value of l read bits with the value of last codeword of length l.

Klein [10] presents an algorithm for fast decoding of canonical Huffman codes, using the special data structure. The data structure is a binary tree called a *sk-tree*. The algorithm makes use of the numerical sequence property of canonical Huffman codes. Let $first(l)$ be the index of first codeword of length l in the full list of all codewords. Let $I(w)$ be the integer value of codeword w. Then for any codeword w of length l

$$first(l) + (I(w) - I(first(l))) \tag{1}$$

is an index of codeword w in the full list of all codewords. The idea of the algorithm is to read single bits untill the length of the codeword, which is currently being read is determined. The remainning bits of the codeword are then read at once and its index in the list is determined by the formula given above. The sk-tree is a data structure used for detemining the length of the current codeword as soon as it is possible. External paths of sk-tree constitute all posible binary sequences, necessary for uniquely determining the codeword lengths. The average number of operations per codeword necessary for decoding a single codeword with this algorithm can be approximately estimated by:

$$\sum_{i \in \text{leaves in sk-tree}} (d(i) 2^{-d(i)})$$

the savings in space and time depend on and significantly vary with concrete size and structure of the Huffman tree. In the worst case there is no saving. Prctical experiments on large text databases show reduction of up to 50

3 Length search tree decoding

In this work a space- and time-efficient algorithm for decoding of Huffman codes with numerical sequence property is presented. The algorithm requires maximum $\lceil \log_2(c) \rceil$ comparisons, where c is the number of different codeword lengths. The space requirements for the corresponding data structure is $O(c)$, which makes the algorithm very practical for the case, when the number of symbols is very high and memory resources are limited. The algorithm can be efficiently applied to Huffman codes with different shapes of Huffman tree. Besides that, the algorithm avoids single bit manipulation,which leads to further reduction of decoding time. The algorithm exploits the following simple property of Huffman codes with numerical sequence property.

Property 1 *For any two codewords w_1, w_2 in canonical Huffman code with lengths l_1, l_2 respectively, if $I(w_1 1^{l_{max}-l_1}) < I(w_2 0^{l_{max}-l_2})$,then for any other codewords w'_1,w'_2 with lengths l_1, l_2 respectively $I(w'_1 1^{l_{max}-l_1}) < I(w'_2 0^{l_{max}-l_2})$.*

For ease of description we consider below only the case of canonical Huffman codes. However the similar algorithm can be constructed for the general case of codes with numerical sequence property.

Property 2 *For any two codewords w_1, w_2 in canonical Huffman code with lengths l_1, l_2 respectively, if $l_1 < l_2$ then $I(w_1 1^{l_{max}-l_1}) < I(w_2 0^{l_{max}-l_2})$.*

Proof: Let $l_1, l_2, \ldots, l_n$ be the sequence of codeword lengths. Let $maxword(l)$ be the last codeword of length l. In canonical Huffman code $minword(l_1) = 0$ and $minword(l_i) = 2^{l_i - l_{i-1}} * (minword(l_{i-1}) + n_{i-1}))$ and the codewords of the same length are consecutive integers. Hence $maxword(l_i) = minword(l_i) + n_i - 1$. For any codeword length l_i

$$I(minword(l_{i+1}) 0^{l_{max}-l_{i+1}}) = I(maxword(l_i) 1^{l_{max}-l_i}) + 1$$

.

Therefore, for any codeword w_1, which length is lesser than l, $I(minword(l) << (l_{max} - l)) > I(w_1 \alpha^{l_{max}-l_1})$, where '<<' is a left shift operation. Hence if we read the sequence $\alpha^{l_{max}}$, prefix of which is some codeword w of length l, then for any $l' > l$ $I(minword(l') << (l_{max} - l')) > I(\alpha)$. In other words, if we read a sequence of l_{max} bits from the input stream, we need one comparison to determine, whether its prefix codeword has length, lesser than l, for any $l < l_{max}$.

The algorithm works as follows. First a binary sequence of length l_{max} is read.Then the length of its prefix codeword is determined, using the above mentioned property of canonical codes, by comparing the value of α with the critical values for different lengths, where under critical value for length l we understand $I(minword(l)0^{l_{max}-l})$. Then the index of the codeword is computed, using the formula (1). To accelerate the search of the length of prefix codeword binary search tree is used. As the decoder doesn't know probabilities of different symbols, we'll assume that the probability, that codeword length equals l is $1/c$. In this case, to get the length of the codeword with help of the binary search tree we need $\lceil \log_2(c) \rceil$ comparisons. Alternatively, if decoding time is more important than compression efficiency, encoder, knowing the probabilities of codewords, can construct an optimal binary length search tree and transfer it along with encoded data. Then decoder will need $\lceil \log_2(c) \rceil$ comparisons in worst case. The storage overhead for optimal search tree is $O(c)$. To construct such a tree a $c \log_2(c)$ time is needed [7].

This tree has a following structure. Each internal node N contains the critical value for some codeword length l. We traverse the tree, taking on each step left edge of current node, if decoded bit sequence is less than value, stored in current node, and right edge otherwise, until we achieve a leaf node. The leaf node contains the length of prefix codeword of decoded bit sequence. Description of decoding procedure in pseudocode is given below.

```
Procedure decode()
begin
N:= root of search tree;
w:= integer corresponding to sequence of next n undecoded bits;
repeat
if (w<N.value)
        N:=left child of N;
else
        N:=rigth child of N;
until (N is a leaf node);
length:=N.value;
```

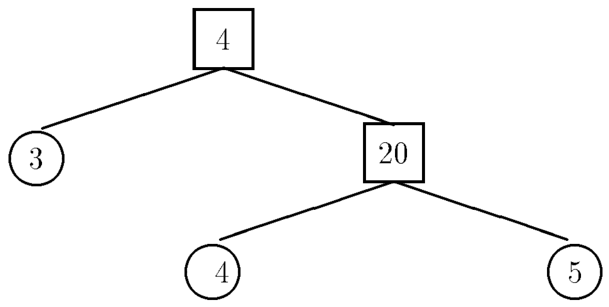

Figure 1: A tree for length search

```
index:=first[length]+
       (I(first length bits of (w-minvalue[length])) );
output symbol[index];
end
```

As 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 $minvalue[3] = I(000 << 2) = 0$; $minvalue[4] = I(0010 << 1) = 4$; $minvalue[5] = I(10100) = 20$;
first[3]=1; first[4]=2; first[5]=10;
The search tree T for this code is depicted on fig. 1. Internal nodes are denoted by squares, leaves are denoted by circles. Internal nodes contain

critical values for codeword lengths 4 and 5. Leaves contain the lengths of codeword. For instance, suppose, that we read sequence 10001. $I(10001) > minvalue[4] = 4$; $I(10001) < minvalue[5] = 20$; hence prefix codeword has length 4. Therefore, the index of the current codeword is $first[4] + ((I(10001) - minvalue[4]) >> 1) = 2 + ((17 - 4) >> 1) = 8$.

Using length search tree method we need 2 comprarisons to decode a single codeword. Number of comparisons necessary to decode the average codeword with the standard Huffman tree method is 4.25 (average codeword length of the code). Number of comparisons used in sk-tree decoding is 3*(1/8)+3*2*(1/16)+2*4*(1/16)+3*2*(1/16)+3*4*(1/32)+2*8*(1/32)=2.5 . Besides this, our method avoids bit retrieval operations, which are necessary on every tree traversal step in other two methods.

We assumed, that the root of the tree is not a leaf (otherwise all codewords have the same length and we don't need the tree to determine lengths of the codewords). Therefore our decoding procedure uses in worst case $\lceil \log_2(c) \rceil$ integer comparisons and $\lceil \log_2(c) \rceil$ checks if the current node is a leaf. Getting the sequence of yet unencoded bits can be implemented with a constant number of bit-oriented operations, independently of the code length [1]. The search tree and supplementary structures require kc bytes of memory, where k is a small constant, depending on the number of codewords [2].

Logarithmic limitations on time mean that worth case decoding time grows very slowly with growth of the size of source data . Suppose that the algorithm requires k comparisons, then the height of corresponding Huffman tree is 2^k. Itis known that if Huffman tree has a leaf on level d, then the probability of the corresponding element is less than $(1/\phi)^{d-1}$, where ϕ is the golden ratio. [9]. Therefore the minimal length of source file should be ϕ^{2^k} (the corresponding symbol would appear only once in this file,and besides that, there should be at least one leaf on every other level of the tree, which is a very unlikely situation). The memory requirements are $O(2^k)$. For instance, supposed that our algorithm requires 6 comparisons, then the

[1]For instance in C getting the next yet unencoded bits can be implemented with 2 shift and 1 exclusive OR operations, to get the index number we need one more shift operation.

[2]Again, in C *minvalue* array would require 2c bytes, if maximum codeword length is less then 16, 4c bytes otherwise, *first* array would require c,2c or 4c bytes if number of codewords is less than 2^8, 2^{16} and 2^{32} respectively.we would need $3c$ bytes of memory for search tree, if number of codeword is less than 2^8, $6c$ bytes if number of codewords is less than 2^{16} and $12c$.

minimum source file length is $\phi^{64} = 2^{0.693*64} = 2^{44.352}$ i.e the file would be at least 20,911.371 Gigabytes long. Or, in other words, for any file of size lesser than 20,911.371 Gigabytes we'll need no more than 6 comparisons to decode the codeword.For decoding of such file about 1Kbyte of memory would be required. Klein [10] argues, the Huffman trees for distributions of characters or character pairs in natural languages have depth $O(\log(n))$. In this case the number of comparisons with length search tree is $O(\log(\log(n)))$. These results can be further improved, if we store the optimal length search tree along with compressed data.

4 Improved decoding with look-up table.

In the decoding with the look-up table we first try to look-up the code in a table, containing all t-bit sequences. The decoding of the next codeword begins with reading the sequence of next t bits from the input stream. If some codeword w is a prefix of the read sequence we output the symbol, corresponding to w, otherwise decoding proceeds, using further tables or another decoding method, for instance, the method described above, but we know already that the codeword length is greater than t. In this section we propose an improvement to the look-up table method. To the best of our knowledge, this method is new. Based on ideas from [10] , we try to get additional information about the length of the current codeword from its first t bits, thus simplifying the decoding. The first t bits of the codeword can contain enough information to determine the codeword length unambiguously or to limit the range of codelengths. These ideas are used in the following way. For decoding of the next codeowrd we first read the sequence α of next t bits from the input stream. If a certain codeword w is a prefix of sequence α we output symbol corresponding to w. Otherwise we try to get information about the possible codeword length. If the codeword length can be determined unambiguously, we compute the codeword index with the formula $index(w0^{l-t}) + I(\text{next } l - t \text{ bits})$ and output the symbol with this index. If codewords with different lengths have prefix α, we grab the next $k - t$ bits from the input stream, where k is the maximal possible codeword lengrth. If $k - t \leq 3$ decoding proceeds with use of a further decoding table. If $k - t > 3$ decoding proceeds with length-search tree method, similar to one described in the previous section, except that we know already, that the codeword length is greater than t and smaller than k. The data structure, neseccary for improved look-up decoding, can be implemented as a table of records, each record contains the length field, value field and type field. Type field specifies which type of decoding should be applied to the given sequence. Length field contains the length of the current codeword, if codeword length is less than t or codeword length can be determined. Value field contains value of the encoded symbol, if codeword length is less than t, index of the first codeword with given prefix, if codeword length can be determined, or reference to corresponding search tree or decoding table, if decoding should be continued with a length search tree or decoding table respectively. An algorithm for construction of the look-up table is given below.

```
Procedure ContructDecodingTable()
begin
code:=first codeword;
while (length[code] < t)
  begin
    for (all strings w of length t such,
                 that code is prefix of t)
      begin
        look-up_table[w].value:=value[code];
        look-up_table[w].length:=length[code];
        look_up_table[w].type:=DIRECT_DECODE;
      end
    code= next code;
  end

i:=length[code];
w:=first unprocessed string of length t;
while (i <= maximum code length)
  begin
    for (all unprocessed strings w, such that all codewords
          with  prefix w have length i )
      begin
        look-up_table[w].value:=index of first codeword
                                    with prefix w;
        look-up_table[w].length:= i-t;
        look_up_table[w].type:=SAME_LENGTH;
      end
    j:=i;
    w:= next unprocessed string of length t;
    while (there are codewords of length j having prefix w)
      j:=j+1;
    if (j>i) AND (j<t+3)
      begin
        construct look-up table, containing
             all possible endings of w;
        look-up_table[w].value:=refernce to this table;
        look-up_table[w].length:= j-t-1;
        look_up_table[w].type:=NEXT_TABLE;
      end
    if (j>i) AND (j>t+3)
      begin
        construct length search tree for codewords
                 with prefix w;
        look-up_table[w].value:=refernce to this tree;
        look-up_table[w].type:= SEARCH_TREE;
      end
    if (j=i)
      i:=i+1;
    else
      i:=j;
  end while
end
```

```
Algorithm 2. Construction of the  data structure for
              the improved look-up method.
```

The construction of this data structured can be done in time, linear on number of symbols in the source alphabet.

5 Experimental results

The described decoding methods were tested on files from Calgary text compression corpus.(see [2]). Table 1 compares the results of decoding,using binary search tree with decoding, using standard Huffman tree. Table 2 compares decoding with standard Huffman tree and decoding using optimal binary search tree. The third column displays the average length of Huffman code, which corresponds to number of comparisons, using decoding method with a standard Huffman tree. Sixth column displays the depth of binary length search tree, which corresponds to number of comparisons in presented method. Fourth and seventh columns give the number of operations per codeword in both methods. We assumed that for standard Huffman tree method we need 5 operations for each traversed tree node (namely, we should check if current node is a leaf node, read the next bit, increment position in the bit stream,check whether bit's value is 1 or 0 and descend on one level in tree) and for presented method we need 3 operations (check if node is a leaf,compare its value with critical value of the node and descend one level deeper) plus 7 additional arithmetic operations. Fifth and eighth columns contain information about the size of respective tree structures for both methods.

Tables 3 and 4 show the results of encoding the same files, but instead of assigning a code to every single symbol, sequences of two symbols are encoded. Table 3 and 4 show results for standard and optimal binary length search respectively. The time and memory advantages grow dramatically with the growth of alhabet.

File	File Size	Standard tree			Length search tree		
		Average depth	Number of op-s	Tree size	Average depth	Number of op-s	Tree size
bib	111261	5.23	26.15	161	3.69	18.07	25
book1	768771	4.56	22.8	163	4	19	33
book2	610856	4.82	24.1	191	3.84	18.52	27
obj1	21504	5.97	29.85	511	3.40	17.2	21
obj2	246814	6.29	31.45	511	3.60	17.8	25
paper1	53161	5.01	25.05	189	3.67	18.01	25
paper2	82199	4.63	23.15	181	3.74	18.22	27
paper3	46526	4.68	23.4	167	3.23	16.69	21
paper4	13286	4.73	23.65	159	3.62	17.86	23
paper5	11954	4.97	24.85	181	3.68	17.04	23
paper6	38105	5.04	25.2	185	3.63	17.89	25
pic	513216	1.66	8.3	317	3.12	16.36	29
progc	39611	5.23	26.15	183	3.62	17.86	23
progl	71646	4.80	24.0	173	3.75	18.25	23
progp	49379	4.89	24.45	177	3.58	17.74	25

Table 1:

File	File Size	Standard tree			Length search tree		
		Average depth	Number of op-s	Tree size	Average depth	Number of op-s	Tree size
bib	111261	5.23	26.15	161	2.67	15.01	25
book1	768771	4.56	22.8	163	2.46	14.38	33
book2	610856	4.82	24.1	191	2.52	14.56	27
obj1	21504	5.97	29.85	511	3.03	16.09	21
obj2	246814	6.29	31.45	511	3.10	16.30	25
paper1	53161	5.01	25.05	189	2.62	14.86	25
paper2	82199	4.63	23.15	181	2.45	14.35	27
paper3	46526	4.68	23.4	167	2.49	14.47	21
paper4	13286	4.73	23.65	159	2.51	14.53	23
paper5	11954	4.97	24.85	181	2.62	14.86	23
paper6	38105	5.04	25.2	185	2.66	14.98	25
pic	513216	1.66	8.3	317	1.33	10.99	29
progc	39611	5.23	26.15	183	2.64	14.92	23
progl	71646	4.80	24.0	173	2.41	14.23	23
progp	49379	4.89	24.45	177	2.75	15.25	25

Table 2:

File	File Size	Standard tree			Length search tree		
		Average depth	Number of op-s	Tree size	Average depth	Number of op-s	Tree size
bib	111261	8.58	42.9	2645	3.62	17.86	23
book1	768771	8.14	40.7	3267	3.95	18.85	29
book2	610856	8.56	42.8	5477	3.94	18.82	27
obj1	21504	9.17	45.85	6127	3.26	16.78	17
obj2	246814	8.93	44.65	12339	3.78	18.34	27
paper1	53161	8.64	43.2	2705	3.53	17.59	21
paper2	82199	8.13	40.65	2243	3.45	17.35	21
paper3	46526	8.23	41.15	2021	3.46	17.38	21
paper4	13286	8.13	40.65	1409	3.07	16.21	17
paper5	11954	8.43	42.15	1623	3.00	16.00	15
paper6	38105	8.61	43.05	2435	3.44	17.32	19
pic	513216	2.38	11.9	4641	3.11	16.33	25
progc	39611	8.80	44.0	2885	3.42	17.26	19
progl	71646	8.00	40.0	2063	3.67	18.01	23
progp	49379	8.06	40.3	2507	3.52	17.56	23

Table 3:

File	File Size	Standard tree			Length search tree		
		Average depth	Number of op-s	Tree size	Average depth	Number of op-s	Tree size
bib	111261	8.58	42.9	2645	2.96	15.88	23
book1	768771	8.14	40.7	3267	3.02	16.06	29
book2	610856	8.56	42.8	5477	3.17	16.51	27
obj1	21504	9.17	45.85	6127	3.19	16.57	17
obj2	246814	8.93	44.65	12339	3.63	17.89	27
paper1	53161	8.64	43.2	2705	3.08	16.24	21
paper2	82199	8.13	40.65	2243	2.99	15.97	21
paper3	46526	8.23	41.15	2021	2.99	15.97	21
paper4	13286	8.13	40.65	1409	2.92	15.76	17
paper5	11954	8.43	42.15	1623	2.89	15.67	15
paper6	38105	8.61	43.05	2435	3.04	16.12	19
pic	513216	2.38	11.9	4641	1.56	11.68	25
progc	39611	8.80	44.0	2885	3.06	16.18	19
progl	71646	8.00	40.0	2063	3.21	16.63	23
progp	49379	8.06	40.3	2507	3.26	16.78	23

Table 4:

Table 5 contains results of decoding the Huffman codes with table look-up method with $t = 8$. The same set of files combined as in table 3 and 4 was used. in the standard table-lookup method we read t bits and then, if current codeword length is greater than t, read following bits until current codeword is read, in a way, similar to algorithm described by Connell(s. section 2).Improved look-up method uses the algorithm described in section 4.

Improved look-up method gives the best results in case of decoding data with large source alphabet when memory is limited. In case when memory resources are extremely limited, optimal length search tree seems to be the best choice.

References

[1] T. Boutell et. al. PNG(Portable Network Graphics) Specifications, RFC

File	File size	Alphabet size	Number of operations	
			Look-up method	Improved look-up method
bib	111261	1324	11.73	7.95
book1	768771	1634	10.34	7.54
book2	610856	2739	11.92	7.74
obj1	21504	3064	19.34	9.06
obj2	246814	6170	16.13	8.24
paper1	53161	1353	11.98	7.85
paper2	82199	1122	10.23	7.30
paper3	46526	1011	10.46	7.47
paper4	13286	705	10.16	7.25
paper5	11954	812	11.18	7.62
paper6	38105	1218	11.81	7.82
pic	513216	2321	7.37	5.99
progc	39611	1443	12.73	8.04
progl	71644	1032	10.44	7.53
progp	49379	1254	12.04	7.95

Table 5:

2083,ftp://wuarchive.wustl.edu/doc/rfc/rfc2083.txt, March 1997.

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

[3] Kuo-Liang Chung. Efficient Huffman decoding. Information Processing Letters, 61(2),1997, 97-99.

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

[5] P. Deutsch, Deflate compression data format specifications, RFC 1951,ftp://ftp.isi.edu/in-notes/rfc1951.txt, May 1996.

[6] Hirschberg D.S., Lelewer D.A. Efficient decoding of prefix codes, Communication of the ACM 33(1990),449-459.

[7] T.C.Hu, A.C. Tucker. Optimal computer search trees and variable-length alphabetical codes.. SIAM Journal on Applied Mathematics, 21(1971), 514-532.

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

[9] Katona G.H.O., Nemetz T.O.H. Huffman codes and self-information, IEEE Transactions on Information Theory, 11(1965),284-292.

[10] 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.

[11] Lelewer D.A., Hirschberg D.S., Data compression, ACM Computing Surveys, 19(1987), 261-296.

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

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

[14] Gregory K.Wallace. JPEG still image compression standard., Communications of the ACM, 34(1991),46-58.