THE HUFFMAN ALGORITHM

Student BOGDAN MORMOGEAC
Student VERONICA SCARLAT
Student ADRIAN STANCIULESCU
Faculty of Economic Cybernetics, Statistics and Informatics,
Academy of Economic Studies

1. Biography

This method was used for the first time by Huffman in 1952 . The compression method shown here, based on Huffman codified static trees Huffman, has certain disadvantages.

These disadvantages due to which it is no longer actually used nowadays, have been mostly eliminated for the methods based on dynamic Huffman trees, methods drawn upon the classic algorithm. Their idea was that for each character codification, the Huffman tree should be reorganized so that that character eventually is reassigned a shorter code. The differences between these dynamic methods consist precisely of the way the tree is reorganized. The Huffman dynamic methods are able to adjust the lengths of any character's code according to their local frequency..

After this algorithm appeared, many alternative algorithms have developed. The differences between them and the initial standard algorithm appear generally in the phase of attributing each symbol a certain code of bits. Among these alternatives we mention Huffman modified code, Huffman adaptive algorithm, Fano-Shannon algorithm etc.

2. The Algorithm

Let's consider the characters of a text file, which contains a documentary written in the Romanian language. It is easy to observe that in this text some characters like 'a', 'e' or 'i' appear more often and not so often characters like 'z', 'w' or 'x' etc. In most of nowadays computers all the characters are being coded on 8 bits, in ASCII code. This way all the characters are occupying exactly 1 byte in the file, meaning 8 bits, disregarding their occurrence frequency.

From here comes the idea of compression: to attribute shorter codes to the characters that appear more often in the text and longer codes to the ones that don't appear often. It is presented below the method of obtaining these codes.

2.1. Generating the Huffman tree

Let's consider a text in which characters' appearing frequencies are:
r[4] e[6] g[1] i[2] t[3] a[6]
The characters are being sorted after their increasing frequencies:
g[1] i[2] t[3] r[4] e[6] a[6]
Each letter is considered to be a binary tree with a single node. After that the first two nodes from the above sequence are taken and put together to form one tree which root will have as frequency the sum of the frequencies of the two trees put together. Then the trees list is being sorted again. And so on until there is only one node left which will be the root for the Huffman tree.

Let's see how the considered example will look after making these operations (fig 1):

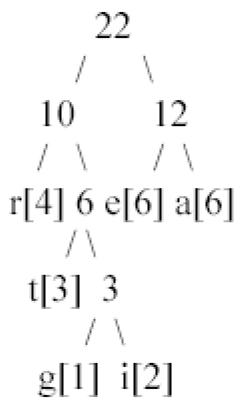


Fig 1. Huffman tree

2.2. The compression

In order to compress, the algorithm proceeds this way: the source file is being traversed and the occurrence frequency for each of the 256 characters in ASCII code is being determined.

```
fseek(fisier,0,SEEK_END);
lungime = ftell(fisier);                        // Determining file's length
fseek(fisier,0,SEEK_SET);
for(nr=lungime;nr>0;nr-)
aparitii[getc(fisier)+255]++;             // Determining characters appearing
fclose(fisier);                                           // frequencies
for(i=0;i<256;i++)                             // Creating the initial 256 nodes
arbori[i]=i+255;                                 //  characters
```

Using these numbers the Huffman tree is being generated. Using two vectors the father and the sons of each node are being memorized. To succeed in doing such operation first the trees are being sorted by the occurrence frequency of each character.

```
void OrdoneazaArbori(void)           // Sorting the characters by the frequencies
        {   int i,n,max,aux;
      //Sorting method: placing maximal number at the end of the vector
                    for(n=255;n>0;n--)
                             {  max=0;
  for(i=1;i<=n;i++)
          if  (aparitii[arbori[i]]>aparitii[arbori[max]])
max=i;
                      aux=arbori[max];
                      arbori[max]=arbori[n];
          arbori[n]=aux;    }
           }
                    // NodCurent has the index for a new, yet unused node
NodCurent=255;
for(primul=0;primul<255;primul++)              //  The first tree in the tree list
{    NodCurent--;                              //   The new NodCurent is being
  fii[NodCurent].fiu1=arbori[primul];             // linked to the first tree
 fii[NodCurent].fiu2=arbori[primul+1];           //  and to the second
 tata[arbori[primul]]=tata[arbori[primul+1]]=NodCurent; // and to his father
```

```
                              // Determining the frequency for the current node
   aparitii[NodCurent]=aparitii[arbori[primul]]+aparitii[arbori[primul+1]];
   Insereaza(NodCurent,primul+2);                    //  Placing the new node in the
 }                                                 //  list of sorted trees, so that the list
                                                          //  remains sorted
```

The procedure of adding the new node is presented below:

```
void Insereaza(int nod,int primul)
          {   for(;(primul<256)&&(aparitii[arbori[primul]]<aparitii[nod]);primul++)
                                 arbori[primul-1]=arbori[primul];
                  arbori[primul-1]=nod;
     }
```

The process of actual compression begins. For start the generated tree is being saved to the compressed file (more exactly only the vector in which are being memorized the sons) in the purpose of reconstructing it in the phase of decompression.

```
 fwrite(aparitii,sizeof(unsigned long),1,arhiva);            // Saving file's length
fwrite(fii,255*sizeof(struct NOD),1,arhiva);                 // Saving Huffman tree
```

After that, the source file is being traversed and for each character its codification used in Huffman algorithm is being copied to the compressed file.

```
      for(NrBiti=0;lungime>0;lungime--)                     //  Each character
              codifica(getc(fisier)+255);                   //   is being coded
       if(NrBiti != 0)                                      //   Last bites are transferred

  putc(octet << (8 - NrBiti),arhiva);                           //    to archive file
```

The procedure for codifying a character is :

```
      void codifica(int nod)
      { int nr=0,biti[255];
       for(;nod != 0;nod=tata[nod])                 // Memorizing bits towards the root
           biti[nr++]=(fii[tata[nod]].fiu2==nod);
       while(nr>0)                               // Getting the bites backwards(root-leaf)
        {      octet = (octet << 1) + biti[--nr];     //  Adding a bit in the byte
                   if(++NrBiti==8)               //  If the byte is full it now can be written
               {       putc(octet & 0x00FF,arhiva);             // in the archive and reset
                      NrBiti=0;
              }
       }
    }
```

This operation is more delicate due to the fact that the codes have an inconstant number of bites while the procedures for writing in a file generate only a number of bits that can be divided by eight.

2.3. Decompression

Decompression is less complex than the compression. First the tree saved in the archive file in the phase of compression is restored, using the vector that memorizes the sons.

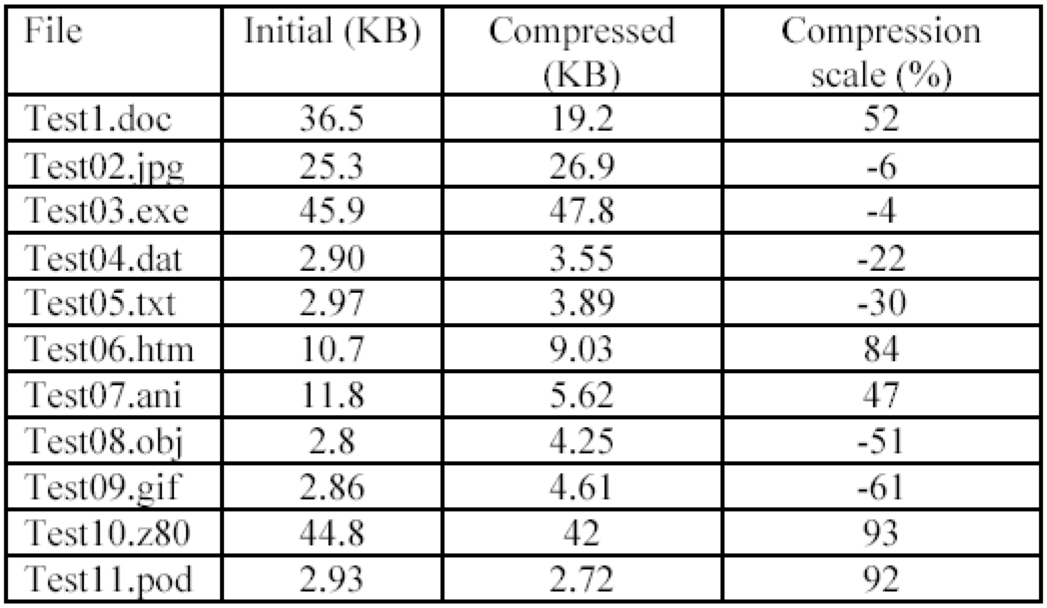
```
fread(&lungime,sizeof(unsigned long),1,arhiva);              //  Restoring the length
fread(fii,255*sizeof(struct NOD),1,arhiva);                  //  Restoring Huffman tree
```

Let's consider a cursor initially placed on the root node of the restored tree; successions of bites are being read from the archive file. For each 0 bit 0 the cursor is advancing to the left son and for each 1 bit the cursor is advancing to the right son of the current node. When the cursor reaches a node that has no sons, the character associated to that code is generated in a destination file and the cursor in being placed on the root node of the tree.

```
//  While not all the original characters have been restored
    for(NrBiti=0;lungime>0;lungime--)
//  it advances until the node is a leaf node
        { for(nod=0;nod<255;nod = (ExtrageBit()==0) ?   fii[nod].fiu1 : fii[nod].fiu2);
           putc(nod-255,fisier);            //  And then  it writes the character
                                           // associated to that node
        }
Procedure ExtrageBit( ) is presented below :
 int ExtrageBit(void)                         //  Returns the next bit from archive file
     { if(NrBiti--==0)                        //  If  the reception octet is empty
              { octet=getc(arhiva);           //   Then restore it from the archive file
                NrBiti+=8;  }
         return ((octet<<=1) & 0x0100); }  //  Returns the bit in the left
```

And it continues like that until the decompression is completed.

3. Test Results



File	Initial (KB)	Compressed (KB)	Compression scale (%)
Test1.doc	36.5	19.2	52
Test02.jpg	25.3	26.9	-6
Test03.exe	45.9	47.8	-4
Test04.dat	2.90	3.55	-22
Test05.txt	2.97	3.89	-30
Test06.htm	10.7	9.03	84
Test07.ani	11.8	5.62	47
Test08.obj	2.8	4.25	-51
Test09.gif	2.86	4.61	-61
Test10.z80	44.8	42	93
Test11.pod	2.93	2.72	92

From the results shown above it is easy to observe that the Huffman algorithm has quite good results working on files with a reduced number of characters. For the standard algorithm, the best results are obtained working on text files (Test06.htm, Test1.doc), cursor file Test07.ani and partial on executable files (Fig. 2).

The average compression rate is 17,63. That is due to the cases in which the number of symbols in the alphabet is quite large. The maximal acceptable length in this algorithm's case for representing the symbols is 12 bites.

Being faster than other algorithms, it is recommended when the compression of files is used frequently.

4. Conclusions

The method of compression shown above, based on static codifying Huffman trees has some disadvantages due to which it is no longer actually used nowadays. The major disadvantages are:

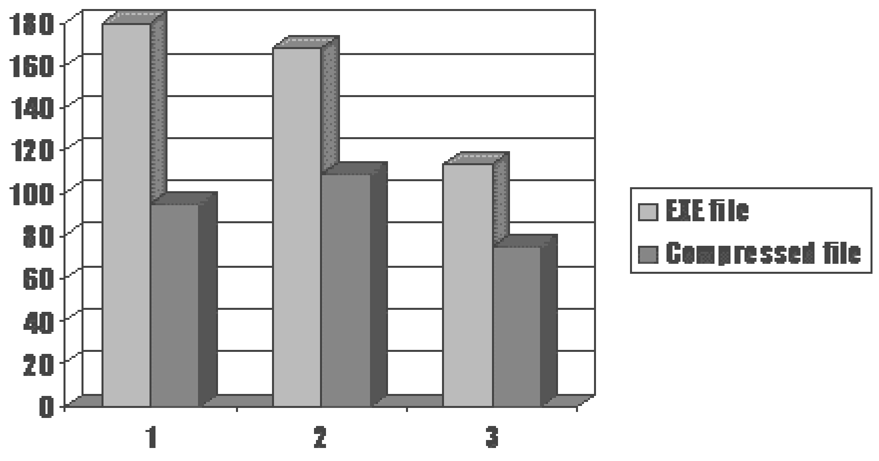
a) The file that will be compressed must be traversed two times: once to calculate the number of times each character appears, and second for the actual compression. This means that the method cannot be used, for example, for data transmission through cable or through satellite, because in this case, the data isn't usually memorized, which makes the resumption impossible. In the same time, the data having a continuous flow, one cannot put an "end" to the transmission in order to build the Huffman and the actual compression cannot be performed.

b) The generated Huffman tree must be memorized in the archive file, which generates a supplementary loading of the compressed code. In the case of the program
shown as an example, the tree that had been saved in the archive file occupies 1020 bytes, which means pretty much for relatively small files. The fact that the size of the tree can be reduced should be mentioned.

c) The method isn't able to adjust to local characters' frequency variations. For instance, for a text with 1000 code 0 characters, then, 1000 code 1 characters, then 1000 code 2 characters and so on, up to 1000 code 225 characters, this text will not be compressed at all, but, on the contrary, it will be "high-flown" due to the memorization of the tree. In spite of all this, a method that would manage to re-codify the characters "on the way" could obtain a really good compression.

However, these disadvantages have been mostly eliminated for the methods based on dynamic Huffman trees, methods drawn upon the classic algorithm. Their idea was that for each character codification, the Huffman tree be reorganized so that that character eventually is reassigned a shorter code. The differences between these dynamic methods consist precisely of the way the tree is reorganized. The Huffman dynamic methods are able to adjust the lengths of any character's code according to their local frequency.

There are advantages, too: the method is simple enough to be understood without great efforts and the compression is implemented faster than other algorithms. It is recommended in the case of frequent compressions.



References

1. [Ivan98] Ion Ivan, Daniel Verniş *Compresia de date,* Ed. Cison, Bucharest, 1998
2. [Nels95] Mark Nelson, Jean-Loup Gailly *The Data Compression Book* Hungry Minds, Inc 2nd Edition, 1995
3. http://www.pcreport.ro/pcrep35/huffman.html
4. http://www.compressconsult.com/huffman/
5. http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/huffman.html
6. http://www.cs.duke.edu/csed/poop/huff/info/
7. http://www.arturocampos.com/ac_static_huffman.html
8. http://www.11a.nu/huffman.htm
9. http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node59.html
10. http://www.cs.uidaho.edu/~karenv/cs213/cs213.useful.pages/huffman.html