Software Explorations

Sqzng Engl Txt

BY JON BENTLEY

My colleague was working on a huge archival database that occupied an expensive farm of disk drives. His predecessors on the task had boasted that by using a commercial data compression scheme, they had reduced the space required by a factor of five (compared to the input records). My colleague was impressed but not awed: he combined a few standard data compression tricks with a couple of new ones, and ended up squeezing an additional factor of four, for a total compression factor of twenty.

That story has two important morals. The first is that standard compression schemes usually work quite well. The second is that special-purpose schemes are sometimes useful, too. In this column we will examine a few data compression techniques by using them on an interesting problem.

The Input File

Our task will be to compress a large file of English text: the biblical Book of Genesis. Here is the first line of the file (broken across two lines to fit this page):

```
GEN 1:1  In the beginning God
created the heaven and the earth.
```

Each of the 1533 verses in the book is presented in a single line in the file:

```
$ wc gen.txt
    1533   41330  212986
$
```

Exercise 1. Find an online Bible and try to compress its first book. Your techniques must be capable of restoring the file to exactly its original condition.

The book, chapter and verse at the start of each line are important when we `grep` the file, but they are redundant when we read the file from the beginning. We can remove that information entirely, and replace it with a special line at the start of each chapter; the new line might contain, for instance, the single word ‘‘CHAPTER’’.

Before we remove the information, though, we should check to ensure that all verses are in the proper order. Here is an Awk program for the task:

```
{   split ($2, x, ":")
    chap = x[1]
    vers = x[2]
    if ((chap == ochap) ? \
            (vers!=overs+1) : \
            (chap!=ochap+1 || vers!=1))
        print "BREAK", ochap, overs,
               chap, vers
    ochap = chap
    overs = vers
}
```

Because there is no pattern, the action is repeated on each input line. The first three lines pry chapter and verse from the second field. The logic is divided into two cases: If this chapter is the same as the old chapter, then this verse must be the successor to the old verse. If the chapters differ, though, then the new chapter must be succeed the old, and the new verse must be one. When I ran this program on the input file, all verses appeared in order.

Exercise 2. Run this program on an online Bible to find if any verses are missing.

Because no verses are missing, we may remove the chapter and verse. This Awk program sets both fields to blank, removes spaces at the beginning of each line, and the prints the modified line:

```
{   $1 = $2 = ""
    sub(/ˆ */, "")
    print
}
```

A complete program would also have to put in a chapter marker (such as ‘‘CHAPTER’’), and a marker for any missing verses (such as ‘‘MISSING’’).

This program reduces the number of characters in the file from 212,986 to 196,808, so the file is 92.4% of its original size. This little exercise illustrates the rules of our game. Our job is to sketch a compression scheme and measure its effectiveness. We don’t have to build a decoder, or even a complete encoder. The techniques, though, must be able to restore the file to its original condition. We may make a few approximations in evaluating compression efficiency (such as ignoring the space required by ‘‘CHAPTER’’ lines), but our answer should be in the right ballpark.

Exercise 3. Sketch techniques that you would use to compress this file.

Huffman Codes

Variable-length codes save space by assigning short codes to common messages and long codes to rare messages. Common English words like ‘‘in’’ tend to be shorter than less common words like ‘‘beginning’’. In

Morse code, the common letter ‘‘e’’ is ‘‘dot’’ while the rare letter ‘‘y’’ is ‘‘dash dot dash dash’’. In 1952, David Huffman gave a precise mathematical statement to this intuitive idea.

Figure 1 shows a binary Huffman code for the letters in Genesis (after stripping the verse numbers). To find the code for a given character, start at the root of the tree and proceed down to that character. A left branch represents a zero and a right brance represents a one. To go to the space character (‘‘sp’’) for instance, we take two left branches, so its code is ‘‘00’’. Similarly, the code for ‘‘o’’ is ‘‘0101’’, and the code for ‘‘b’’ is ‘‘010010’’. (Punctuation characters are represented by two-letter codes: ‘‘sc’’ for semicolon, ‘‘qu’’ for quote, ‘‘ap’’ for apostrophe, and so forth.)

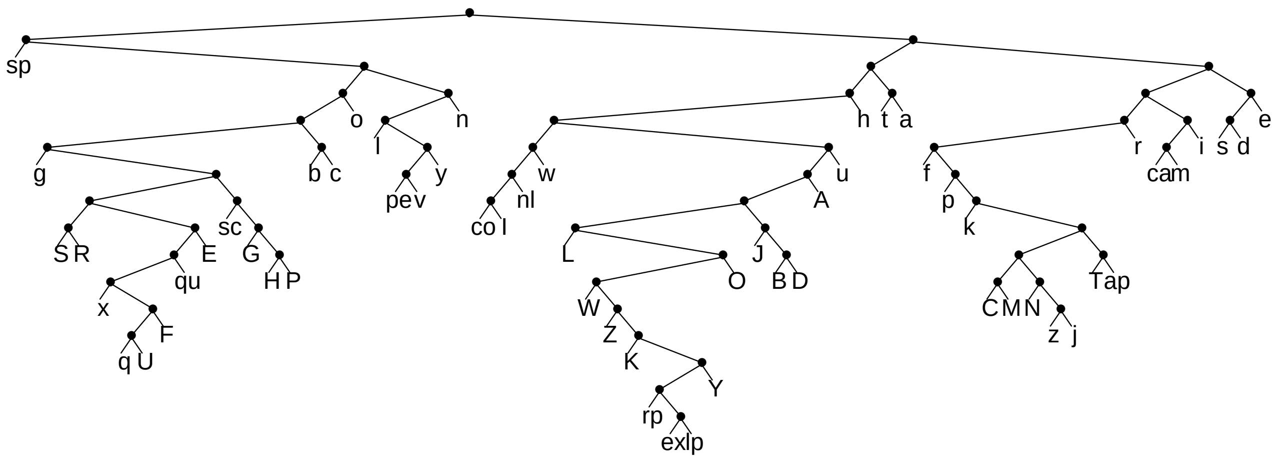


Figure 1. A byte-level Huffman code for Genesis.

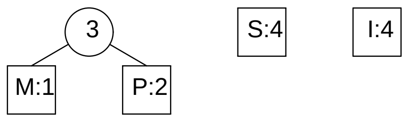
In this section we will study Huffman’s algorithm to construct trees; in the next section, we will apply the algorithm to yield the tree in Figure 1. We’ll consider the example of encoding the single word ‘‘MISSISSIPPI’’. We start by counting all letters:

```
M 1
I 4
S 4
P 2
```

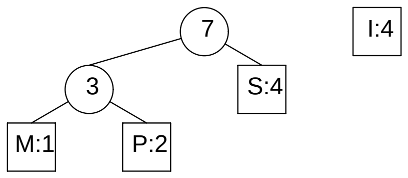
Our task is to build a tree with minimal transmission cost for those letter frequencies, which we’ll call counts.

Huffman’s algorithm starts with a forest of one-node trees and iteratively combines them to form a single tree. Here is the initial set of leaves:

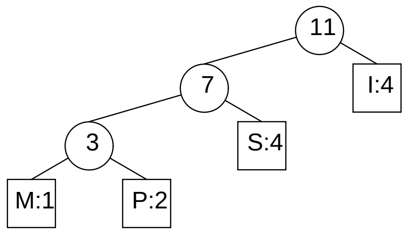
The main step of the algorithm is to choose the two subtrees with minimal count and combine them. In this case, the two smallest values are 1 (‘‘M’’) and 2 (‘‘P’’), so we combine them to make a new node with count 3:



The two smallest counts in the resulting forest are 3 and 4 (‘‘S’’ — ties may be broken arbitrarily), so we combine those into a new node with count 7:



The final step combines the two last trees to yield the optimal Huffman tree:



This tree gives us a simple encoding table:

```
M  000
P  001
S  01
I  1
```

The encoded version of ‘‘MISSISSIPPI’’ is therefore

```
0001010110101100100011
```

This example shows the essence of Huffman’s algorithm: we start with a set of counts, and iteratively sum the two smallest until a single count remains.

Exercise 4. What algorithms would you use in a program to build Huffman trees?

We could keep all counts in an array and at each iteration scan the whole length to find the two smallest counts — that requires quadratic time and is sloppy to code. If we instead store the N counts in a priority queue, we can reduce the total time to $O(N \log N)$, but that can be even more difficult to code. We will weasel around all of these difficulties by sorting the counts in $O(N \log N)$ time and then scanning through them in increasing order in $O(N)$ time.

This table shows the history of the weights in the ‘‘MISSISSIPPI’’ example. The four original weights are presented in increasing order at the left of the first picture. The first step sums 1 and 2 to make 3; in the second line, 1 and 2 are therefore crossed out, and 3 is added to the right.

1 2 4 4
~~1~~ ~~2~~ 4 4 3
~~1~~ ~~2~~ ~~4~~ 4 ~~3~~ 7
~~1~~ ~~2~~ ~~4~~ ~~4~~ ~~3~~ ~~7~~ 11

The third line crosses out the (original) 4 and the (new) 3, and writes a (new) 7 on the left. In the final line only the 11 remains, corresponding to the root of the tree we saw earlier.

This linear representation makes it easy to find the two smallest elements. Because both the input and generated weights appear in increasing order, we can keep two indices into the array to point to the smallest input and generated counts. Program 1 implements this idea in an Awk program. In addition to the `count` array, the program also uses a recursive definition to compute a `cost` array to show the cost in bits of transmitting all symbols below this node in the tree. As its final act, the program prints the cost of sending the root symbol, which is the number of bits required to transmit the complete message.

Exercise 5. Program 1 computes a great deal of information and ignores most of it. Modify the program to print out the Huffman codes or to produce a picture like Figure 1.

```
    { count[NR] = $1  # Number of symbols
      cost[NR] = 0    # Transmission cost
      countsum += $1
    }
END { n = NR
      for (i = n+1; i <= 2*n; i++)
          count[i] = 1 + countsum  # Sentinel
      locheap = 1    # Least input weight
      hicheap = n+2  # Least new weight
      for (i = n+2; i <= 2*n; i++) {
          j = cheaper()
          k = cheaper()
          count[i] = count[j] + count[k]
          cost[i] = cost[j]+cost[k]+count[i]
      }
      print cost[2*n]
   }
function cheaper() {
    if (count[locheap] < count[hicheap])
        return(locheap++)
    else
        return(hicheap++)
}
```

Program 1. `hufcost`: Given counts, find cost of Huffman code.

Character-Level Huffman Codes

To apply `hufcost` to our text file, we need to count how many times each character occurs. Here is the `main` body of the `bytecount` program to do that:

```
int  c, i;
long count[BYTESIZE];

for (i = 0; i < BYTESIZE; i++)
    count[i] = 0;
while ((c = getchar()) != EOF)
    count[c]++;
for (i = 0; i < BYTESIZE; ++i)
    if (count[i])
        printf("%ld\n", count[i]);
```

The first loop initializes the counts to zero, the second loop increments the count for each character read, and the third loop prints nonzero counts.

We can now compute the cost of Huffman coding the (stripped) file with this pipeline:

```
denum gen.txt | bytecount | sort -n | hufcost
```

The final answer is 865,571 bits, which is 55.0% of the original file size.

This experiment ignores a small ‘‘implementation detail’’ that will loom large in the next section: we have accounted for sending the Huffman-encoded bit stream, but how do we send the tree itself? We will use the standard parenthesized representation for trees; the tree we saw earlier for ‘‘MISSISSIPPI’’ will be encoded as

```
(((MP)S)I)
```

Exercise 6. Show that a tree with N leaves can be represented using $N-1$ pairs of parentheses.

Because our text contains only 60 distinct characters, we can represent the tree in about 180 extra bytes.

Exercise 7. The `pack`(1) program uses Huffman codes to compress text files. Measure `pack` on this file, and compare its performance to our quick estimate.

One `pack` I use compressed the file to 877,256 bits, and another reduced it to 866,264 bits; both are close to our prediction of 865,571.

Word-Level Huffman Codes

The often employed but rarely quoted Pig Principle states that ‘‘If some is good, more is better.’’ If encoding letters gives good compression, perhaps encoding whole words will give even better compression.

Our first job is to break the input text into a string of tokens, one per line. Here is the body of the `main` function in the `token` program:

```
int  c, type, otype, started = 0;

while ((c = getchar()) != EOF) {
    if (c == '\n') c = '*';
    if (c == ' ')  c = '_';
    type = isalpha(c) != 0;
    if (type != otype && started)
            putchar('\n');
    putchar(c);
    otype = type;
    started = 1;
}
```

To ease character processing, the first two lines in the loop replace newlines with asterisks and spaces with underscores. Subsequent lines print a newline between alphabetic and nonalphabetic characters. The phrase ‘‘it was so.’’ therefore becomes

```
it
_
was
_
so
.*
```

To feed the `hufcost` program, we have to count how many times each token occurs. This `wordcount` program prints the count of how many times a token occurs followed by the token itself (the token is ignored by `hufcost` but will be used later):

```
    { count[$1]++ }
END { OFS = "  "
      for (i in count)
          print count[i], i
    }
```

It produces a file that (after it is sorted) ends with these five lines:

```
1359    of
2408    the
2428    and
3626    ,_
31822   _
```

We now have all the pieces to measure the cost of Huffman encoding the input text:

```
denum gen.txt | token | wordcount |
    sort -n | hufcost
```

The result is 439,276 bits, or 25.8% of the original file size. This is a huge compression, but it is also a huge lie: we still have to send the words and their Huffman tree.

This little experiment tells us a lot about the size of the token dictionary:

```
  $ denum gen.txt | token | wordcount |
          awk '{print $2}' | wc
     2631    2631   18210
  $
```

There are 2631 distinct tokens, and the tokens require 15,579 characters (ignoring the newlines). By Exercise 6, we can parenthesize the tree using 2630 pairs of, say, angle brackets (‘‘<>’’). The Huffman tree can therefore be represented in $2 \times 2630 + 15,579$ or 20,839 characters, or 166,712 bits. The total number of bits for both the Huffman tree and the stream is 605,988, which is 35.6% of the original file size.

We can use two old tricks to reduce the space even further. Both common sense and inspection of the output of the token program tell us that most words are followed by a single space character. We may delete those from the stream at the cost of a slightly smarter decoding program (supply a space unless the next token is punctuation). To measure this approach, I manually deleted the last line in the counts file:

```
  31822    _
```

When I ran the `hufcost` program again, the cost decreased from 439,276 bits to 362,236 bits.

The new Huffman tree (without the ‘‘_’’ word) is represented in 20,836 characters. Because there are only 62 distinct characters, though, we may represent each one in 6 bits. The cost of sending the tree is therefore 125,016 bits, and the total cost is 487,252 bits, or 28.6% of the original file size.

Exercise 8. While `pack`(1) is based on Huffman codes, `compress`(1) uses the more powerful Lempel-Ziv algorithm. How well does `compress` do on this file?

Principles

Prototyping and Measuring. We’ve learned a lot about data compression, but we’ve built only little pieces of encoders and no decoders at all. Instead, we’ve built tools to measure the performance of methods. Sometimes such measurements help us avoid the work of building the complete tool. If we do build a working system, we should be careful to compare its performance to the cheap measurements from our prototypes.

Huffman Codes. This elegant algorithm is eminently practical. We can apply the scheme to encoding individual letters, English words, or a variety of other objects. We computed new Huffman trees for this particular document, but we might compute general trees for English letters or words, and thereby avoid the cost of computing and transmitting the tree for each document.

Engineering Data Compression. Beyond its underlying mathematics, data compression has a rich heritage of engineering tricks. We exploited problem-specific knowledge in removing the spaces between words and chapter and verse numbers. We have ‘‘mixed and matched’’ a few tools, but there is more work to be done.

Exercise 9. How much further can you squeeze this file?

Solutions to Selected Exercises

1. Try the online Bible at Project Gutenberg:

```
ftp://mrcnext.cso.uiuc.edu/
  gutenberg/etext92/bible10.txt
```

2. The online Authorized Version had all verses present. Since that translation (dedicated to King James), modern scholarship has found still more ancient manuscripts, so some verses are missing in new translations (see, for instance, John 7:53-8:11).

8. On my system, `compress` squeezes the file down to 35.9% of its original size, which is about 25% larger than our method. The `gzip` program reduces the file to 29.6%.

9. We could use Huffman codes to encode the word-level Huffman tree. The June, 1991, edition of this column compressed an English dictionary; many of those techniques are useful in sending the Huffman tree.

Jon Bentley is a Member of Technical Staff in the Computing Science Research Center at AT&T Bell Laboratories.