LZW Compression

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner
Spring 2003

Recall: Compression

We want easily computable functions

$$C : \text{files} \longrightarrow \text{data}$$
$$D : \text{data} \longrightarrow \text{files}$$

For us, decompressing after compression must reproduce the original file exactly: $D(C(x)) = x$.

(But lossy compression is very useful in certain areas.)

Huffman Coding

Compression

- Scan the input file and determine character counts.
- Build the code tree.
- Write the code tree to the output file as a header.
- Scan the input file, and write the codes of the characters to the output file.

Deompression

- Read the header in the compressed file, build the code tree.
- Read the rest of the file, use code tree to replace binary sequences by their corresponding characters.
- Write the characters to the decompressed file.

Optimality

Claim: Huffman coding produces on optimal prefix code based on the frequencies of letters in the input file.

This does **NOT** mean that Huffman is an optimal compression method.

Other methods may simply use different techniques.

Optimality Proof (sketch)

Length of the compressed file is

$$\mathrm{cost}(T) = \sum_{c} \mathrm{count}(c) \cdot \mathrm{depth}_T(c).$$

where T is the code tree produced by Huffman's algorithm.

Lemma: Huffman code is optimal: no other prefix code produces a shorter compressed file.

Claim:

T full binary tree with n leaves, a and b minimal count leaves. Can merge a and b to produce new tree T' such that

$$\mathrm{cost}(T') \leq \mathrm{cost}(T) - \mathrm{count}(a) - \mathrm{count}(b).$$

Equality holds iff a and b are siblings.

Ultimate Compression

Here is a wild idea: we can think of a Huffman compressed file as a special type of program.

When executed on a special Huffman machine, this program will generate as output the original file.

How about replacing the Huffman machine by a universal computer?

Let's say we use a C compiler plus RE.

The compressed file would then be just be a C program that, when executed, produces the input file.

Example: Pi

31415926535897932384626433832795028841971693993751058209749445923078164062862089
98628034825342117067982148086513282306647093844609550582231725359408128481117450
28410270193852110555964462294895493038196442881097566593344612847564823378678316
52712019091456485669234603486104543266482133936072602491412737245870066063155881
74881520920962829254091715364367892590360011330530548820466521384146951941511609
43305727036575959195309218611738193261179310511854807446237996274956735188575272
48912279381830119491298336733624406566430860213949463952247371907021798609437027
70539217176293176752384674818467669405132000568127145263560827785771342757789609
17363717872146844090122495343014654958537105079227968925892354201995611212902196
08640344181598136297747713099605187072113499999983729780499510597317328160963185
95024459455346908302642522308253344685035261931188171010003137838752886587533208
38142061717766914730359825349042875546873115956286388235378759375195778185778053
21712268066130019278766111959092164201989380952572010654858632788659361533818279
68230301952035301852968995773622599413891249721775283479131515574857242454150695
95082953311686172785588907509838175463746493931925506040092770167113900984882401
28583616035637076601047101819429555961989467678374494482553797747268471040475346
46208046684259069491293313677028989152104752162056966024058038150193511253382430
03558764024749647326391419927260426992279678235478163600934172164121992458631503
02861829745557067498385054945885869269956909272107975093029553211653449872027559

10000 Digits

```
long a[35014],b,c=35014,d,e,f=1e4,g,h;
main(){
 for(;b=c-=14;h=printf("%04ld",e+d/f))
  for(e=d%=f;g=--b*2;d/=g)
   d=d*b+f*(h?a[b]:f/5), a[b]=d%--g;
}
```

This C program is just 143 characters long!

And it outputs the first 10,000 digits of π.

This is a compression ratio of 1.4%.

Program-Size Complexity

For every string x, let $P(x)$ be the smallest C program that generates x as output.

Let $K(x)$ be the size of $P(x)$: Kolmogorov-Chaitin complexity.

So we can compress x down to size $K(x)$.

As it turns out, up to a constant, this is the best we can do in general.

So this is the ultimate compression method: replace x by $P(x)$ for compression, and run $P(x)$ to get back x for decompression.

What's Wrong?

There is a bit of a problem: This idea disregards efficiency in a rather criminal way.

- It may take a long time to compute $K(x)$ and the corresponding $P(x)$.
- It may take a long time to compute x from $P(x)$.

Things are even worse:

Theorem. *There is no algorithm to compute $K(x)$ or $P(x)$.*

This means no algorithm at all, regardless of efficiency.

Still . . .

There is an interesting idea here.

In fact, program-size complexity is a very important tool in theoretical computer science.

Can be used to define randomness (incompressibility): x is random if $K(x)$ is essentially equal to the length of x.

Random inputs are very useful to establish lower bounds for the efficiency of algorithms.

Ming Li, Paul Vitanyi: *An Introduction to Kolmogorov Complexity and its Applications*

Lempel-Ziv Compression

Back to the real world. We have to keep an eye on efficiency.

How about a learning approach?

We will build a dictionary of abbreviations, and replace words in the input by their abbreviations.

Let's say the dictionary has the format

$$C : \text{words} \longrightarrow \mathbb{N}$$

So we use numbers as codes.

Words here just means arbitrary blocks of letters, nothing more.

There are many variants, we'll discuss LZW.

Adaptive

We will build the dictionary on the fly, as we scan and compress the input.

We keep track of words we have already seen, and replace (long) words by a (short) number:

When we encounter a block w in the input that is already in C we ouput $C(w)$.

Also, unlike with Huffman, we will not transmit the dictionary C, just the sequence of code numbers $C(w_1), C(w_2), \ldots, C(w_m)$.

In other words, we have to set things up in a way so that the decompressor can rebuild the (inverse of the) compression dictionary from scratch.

For the moment, let's focus on the compression end.

Growing the Dictionary

As a running example, suppose we only deal with letters a, b, c, d.

We initialize the dictionary C with all the single letters:

$$\begin{array}{cccc} a & b & b & c \\ 0 & 1 & 2 & 3 \end{array}$$

Clearly, this can easily be duplicated on the decompressor end.

So far, we're fine.

Now we need a method to grow the dictionary as we scan the input string. And do it in such a way that the decompressor can do the same, without access to the input string.

The Basic Algorithm

We scan a block $a_1a_2\ldots a_kb$ of letters in the input.

Up to $i = k$, all the prefixes $a_1a_2\ldots a_i$ are in the dictionary.

But $a_1a_2\ldots a_kb$ is no longer in the dictionary.

At this point we

- emit the code for $a_1a_2\ldots a_k$,
- add $a_1a_2\ldots a_kb$ to the dictionary,
- continue with $a_1 = b$.

Example 1

Let $x = abbabbc$.

	emit	new	code number
$abbabbc$	0	ab	4
$abbabbc$	1	bb	5
$abbabbc$	1	ba	6
$abbabbc$	4	abb	7
$abbabbc$	1	bc	8
$abbabbc$	2	$-$	$-$

Admittedly not very impressive, but the input here is just too short.

Example 2

How about $x = abcabcabcabcabcabcabcabcabcabc$?

This compresses to $0, 1, 2, 4, 6, 5, 7, 10, 9, 12, 8, 14$.

Note how except for the first 3 letters we never code a single character.

Ratio: 30 letters vs. 12 integers.

And if we did 50 repetions of the basic block abc we would get a compressed list of length 29.

In the end, the dictionary looks like so:

a	0
b	1
c	2
d	3
ab	4
bc	5
ca	6
abc	7
bca	9
cab	8
$abca$	10
$bcab$	12
$cabc$	14
$abcab$	11
$bcabc$	13

Implementation

Since we have to make queries "is $a_1 a_2 \ldots a_i$ still in the dictionary?" it is natural to use a trie.

Initialize the trie by attaching a child to the root for each letter.

Starting at the root, we traverse a branch of the trie until we come to a leaf L.

Then we emit the corresponding code.

We also attach a new leaf node to L, determined by the next input character b.

Then we reset to the child-of-root determined by b.

Repeat till no input is left.

Pseudo Code

```
initialize C;

c = nextchar;                 // next input character
W = c;                        // a string
while( c = nextchar ) {
  if( W+c is in C )           // dictionary
      W = W + c;
  else
      output code(W);
      add W+c to D;
      W = c;
}

output code(W)                // cleanup
```

Flush

Note that for very long files this method could lead to exceedingly large code numbers.

In reality, one usually only allows at most 16 bits for the codes.

Hence, when the dictionary becomes too large, we flush it and reset to the original situation.

We will simply ignore these details.

Decompression

Recall that we do not transmit the dictionary.

So the real problem is going backwards: all we have is a list of code numbers.

Of course, the alphabet is fixed in advance.

For decompression we can initialize a dictionary

$$D : \mathbb{N} \longrightarrow \text{words}$$

just as in the compression phase: numbers $0, 1, \ldots, k-1$ are mapped to the the k single letters.

The first code number is less than k, so we can decode it.

Decompression 2

From then on, we mimic the process of compression: we get the next code number c and look up $w = D(c)$.

We also keep track of the previous word v so found.

We add vw_1 to the dictionary: the compressor would have done exactly the same.

Repeat until all code numbers are taken care of.

No problem, right?

Example

$x = ababbcabb$ compresses to $0, 1, 4, 1, 2, 6$.

The decompression goes like this:

code	word	new number	new word
0	a	$-$	$-$
1	b	4	ab
4	ab	5	ba
1	b	6	abb
2	c	7	bc
6	abb	8	ca

Only entries 4 and 6 are used in this case.

Near Disaster

In the last example, entry $D(4) = ba$ is already available when code 4 appears for the first time.

Could it happen that c appears when $D(c)$ is still undefined?

Sadly, yes.

Note that this is potentially fatal: if we cannot decompress in general, the whole algorithm is useless.

Near Counterexample

$x = aabbbaa$ compresses to $0, 0, 1, 6, 4$.

The decompression goes like this:

code	word	new number	new word
0	a	—	—
0	a	4	aa
1	b	5	ab
6	$??$	$??$	$??$
4	aa	$??$	$??$

We need to look up $D(6)$ before we have entered it!

Narrow Escape

A closer look reveals that this problem (code number appears before it has been entered in D) can only appear in very limited circumstances:

- On the compressor end, we just emitted $p = C(w)$ and set $C(ws) = q$.
- The decompressor dictionary has entries for all $r < q$, but not for q itself.
- The next code number (after p) is none other than r.

The only way this could have happened, though, is if

$$x = \ldots svsvs \ldots$$

where $w = sv$.

But then we only need to enter $D(r) = svs$ into the decompression dictionary and use this value right away.

Near Counterexample, contd.

In the example from above, $s = b$ and v is emtpy.

Hence, we can continue the decompression as follows:

code	word	new number	new word
0	a	$-$	$-$
0	a	4	aa
1	b	5	ab
6	bb	6	bb
4	aa	7	bba

Another Bad Input

How about an input $x = aaaaa \dots aaaa$?

For example, for 50 a's we get $0, 4, 5, 6, 7, 8, 9, 10, 11, 7$.

(Recall that we still use alphabet a, b, c, d).

In this case all the decompression steps are of the bad kind (except for the first, and possibly the last).

But our recovery method still works: just tack on another a.

The words in the dictionary (other than the initial ones) are all of the form $aaa \dots aaa$.

Yet Another Example

$x = aabbbaabbaaabaababb$

compresses to $0, 0, 1, 6, 4, 7, 8, 10, 6$

Both 6 and 10 are “bad” code words that pop up before the appropriate entry in the decompression dictionary has been generated.

Make sure you understand the decompression process for this example before you start writing code.

Pseudo Code for Decompression

```
initialize D;
pc = nextcode;          // first code word
pw = word(pc);          // corresponding word
output pw;
```

Remember pc (previous code) and pwy (previous word).

First code word is easy: codes only a single symbol.

The Easy Case

```
while( c = nextcode )
{
 if( c is in D )                    // easy case
 {
  cw = word(c);
  pw = word(pc);
  ww = pw + first(cw);
  insert ww in D;
  output cw;
 }
 else
 { ... }

 pc = c;
}
```

The Hard Case

```
else
{
 pw = word(pc);
 cw = pw + first(pw);          // construct missing entry
 insert cw in D;
 output cw;
}
pc = c;

}                                          // end main loop
```

Summary of LZW

- LZW is an adaptive, dictionary based compression method.
- Encoding is easy in LZW, but uses a special data structure (trie).
- Decoding is slightly complicated, requires no special data structures.