CS61B Lecture 39

Wednesday, April 29, 2020

Compression

Git creates a new object in the repo each time a changed file or directory is committed, and things can get crowded as a result. To save space, it compresses each object, and every now and then, it packs objects together into a single file called a "packfile". Besides just sticking them together, Git uses a technique called **delta compression**.

Delta Compression

Typically, there will be many versions of a file in a Git repo: the latest, and previous edits of it, each in different commits. Git doesn't keep track explicitly of which file came from where, since that's hard in general. However, it can guess that files of the same name, and roughly the same size, in two commits are probably versions of the same file.

When this happens, Git stores one of them as a pointer to the other, plus a list of changes specific to that version.

	V1	V2
Without Delta Compression	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses,	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses, and don't give twopence halfpenney for any consequences.
With Delta Compression	[Fetch 1st 6 lines from V2]	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses, and don't give twopence halfpenney for any consequences.

Unix Compression Programs

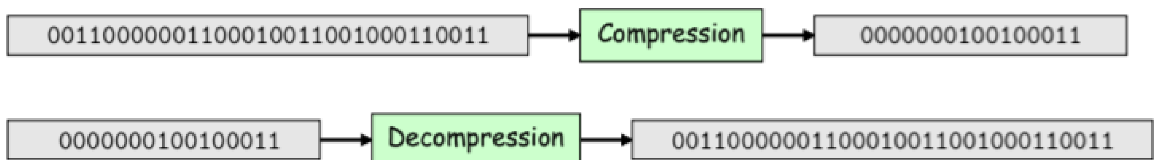
In any Unix system, you'll find two compression programs included: `gzip` and `bzip`. `gzip` is the GNU version of ZIP.

```
$ gzip -k lect37.pic.in # The GNU version of ZIP
$ bzip2 -k lect37.pic.in # Another compression program
```

```
$ ls -l lect37.pic*
#                         Size
#                        (bytes)
-rw-r--r-- 1 cs61b cs61b 31065 Apr 27 23:36 lect37.pic.in
-rw-r--r-- 1 cs61b cs61b 10026 Apr 27 23:36 lect37.pic.in.bz2 # Roughly 1/3 size
-rw-r--r-- 1 cs61b cs61b 10270 Apr 27 23:36 lect37.pic.in.gz
$
$
$
$ gzip -k lect37.pdf
$ ls -l lect37.pdf*
-rw-r--r-- 1 cs61b cs61b 124665 Mar 30 13:46 lect37.pdf
-rw-r--r-- 1 cs61b cs61b 101125 Mar 30 13:46 lect37.pdf.gz # Roughly 81% size
$ gunzip < lect37.pic.in.gz > lect37.pic.in.ungzip # Uncompress
$ diff lect37.pic.in lect37.pic.in.ungzip
$        # No difference from original (lossless)
$
$
$
$ gzip < lect37.pic.in.gz > lect37.pic.in.gz.gz
$ ls -l lect37.pic*gz
-rw-r--r-- 1 cs61b cs61b 10270 Apr 27 23:36 lect37.pic.in.gz
-rw-r--r-- 1 cs61b cs61b 10293 Apr 28 00:16 lect37.pic.in.gz.gz # Compressing a
file twice does not decrease its size
```

Compression and Decompression

A compression algorithm converts a stream of symbols into another smaller stream. It is called lossless if the algorithm is invertible (no information is lost). A common symbol is the bit:



Here, we replaced the 8-bit ASCII bit sequences for digits with 4-bit (binary coded decimal). We call these 4-bit sequences codewords, which we associate with the symbols in original, uncompressed text, and we can do better than 50% compression with English text.

Morse Code and Prefix-Free Codes

- Compact, simple to transmit.
- Actually use three symbols:
  dih, dah, and pause.
  Pauses go between codewords.

Morse code needs pauses between codewords to prevent ambiguities. Otherwise,

▬●●●●▬▬●●●● could be DEATH, BABE or BATH.

The problem is, Morse code allows many codewords to be prefixes of other ones, so that it's difficult to know when you've come to the end of one. The alternative is to devise prefix-free codes, in which no codeword is a prefix of another. Thus, one always knows when a codeword ends.

Encoding A

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

Encoding B

space	111
E	010
T	1000
A	1010
O	1011
I	1100
...	

In these encodings, no bit string is a prefix of another one: there are no bit strings except space that begin with a 1 in A, or a 111 in B.

"I ATE" is unambiguously:

- `0000011000100101` in Encoding A, or
- `110011110101000010` in Encoding B

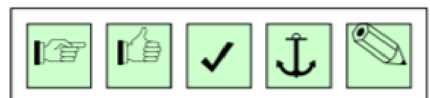
What data structures might you use to encode or decode these bits? Appropriate answers for encoding include `HashMap`s and arrays, while decoding should be done with Tries.

Shannon-Fano Coding

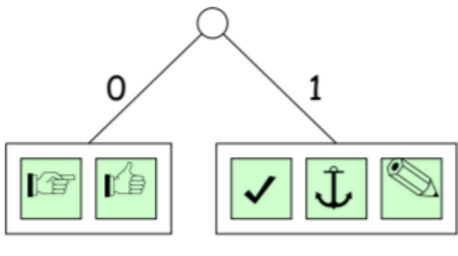
The idea is to adapt your coding to the frequency of symbols in your text, using shorter sequences as symbols for frequent symbols, and longer ones for infrequent symbols, such that for example, in the English language, a common letter such as "N" would have a short encoding, while less common letters like "X" or "Q" would have longer ones, to reduce overall size.

To do so, we would first have to count frequencies of all characters in the text to be compressed, break grouped characters into two groups of roughly equal frequency. Encode the left group with leading 0, right group with leading 1, and repeat until all groups are size 1.

Symbol	Frequency	Encoding
☞	0.35	
👍	0.17	
✓	0.17	
⚓	0.16	
✎	0.15	

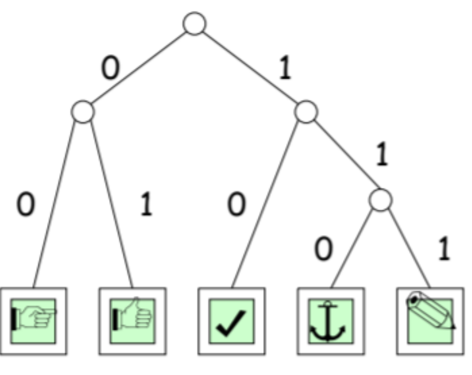


Symbol	Frequency	Encoding
☞	0.35	0...
👍	0.17	0...
✓	0.17	1...
⚓	0.16	1...
✎	0.15	1...



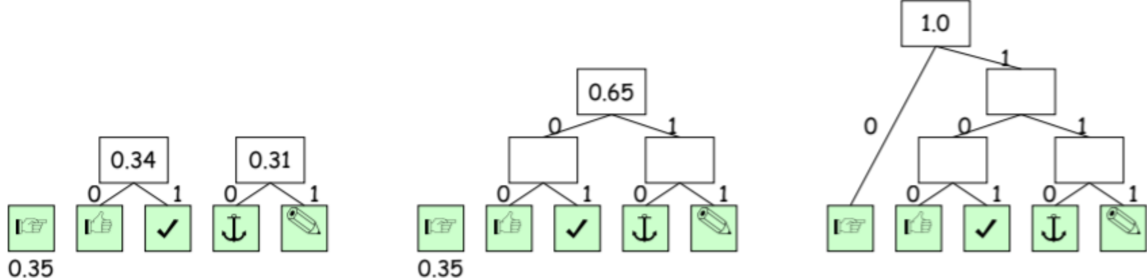
...

Symbol	Frequency	Encoding
☞	0.35	00
👍	0.17	01
✓	0.17	10
⚓	0.16	110
✎	0.15	111



Huffman Coding

We'll say an encoding of symbols to codewrods that are bitstrings is optimal for a particular text if it encodes the text if it encodes the text in the fewest bits. Shannon-Fano coding is good, but not optimal, and the optimal solution was found by an MIT graduate student, David Huffman, in a class taught by Fano. The students were given the choice of taking the final, or doing a term paper finding the encoding and proving its optimality. Fano assigned a problem he hadn't been able to solve, but Huffman found it and the result is called **Huffman coding**.



We put each symbol in a node labeled with the symbol's relative frequency. Repeat the following until there is just one node:

- Combine the two nodes with smallest frequencies as children of a new single node whose frequency is the sum of those two single nodes being combined. Let the edge to the left child be labeled '0', and the right as '1'.

The resulting tree shows the encoding for each symbol: concatenate the edge labels on the path from the root to the symnbol.

Symbol	Frequency	Shannon-Fano	Huffman
☞	0.35	00	0
👍	0.17	01	100
✓	0.17	10	101
⚓	0.16	110	110
✎	0.15	111	111

For this case, Shannon-Fano coding takes a weighted average of 2.31 bits per symbol, while Huffman coding take 2.3.

LZW Coding

To get even better compression, we must encode multiple symbols per code word, allowing us to code string such as

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
abababababababababababababababababababababa
abcdabcdeabcdefabcdefgabcdefghabcdefghiabcd
```

in a space that can be less than 43 times the weighted average symbol length.

In LZW coding, we create new codewords as we go along, each corresponding to substrings of the text. We start with a trivial mapping of codewords to single symbols, and after outputting a codeword that matches the longest possible prefix, X, of the remaining input, add a new codeword Y that maps to the substring X followed by the next input symbol.

The string `B = aababcabcdabcdeabcdefabcdefgabcdefgh` is encoded as `C(B) = 0x616162816383648565876689678b68`, from 200 bits to 120 bits, with the following table:

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba
0x83	abc
0x84	ca
0x85	abcd
0x86	da

Code	String
0x87	abcde
0x88	ea
0x89	abcdef
0x8a	fa
0x8b	abcdefg
0x8c	ga
0x8d	abcdefgh

Here's a question to think about: how might we represent this table to allow easily finding the longest prefix at each step?

Decompression involves checking against the known list of codes, then manually deducing what a particular sequence might represent:

C(B) = 0x61616281638364

- S(0x81) is 'ab' in the table, so add it to B.
- We have two codewords—S(0x62)='b' and S(0x81)='ab'—so add 'ba' to the table as a new codeword.

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	aa
0x81	ab
0x82	ba

B = aabab

This table will almost always contain the mapping we need, but there are cases where it doesn't. Consider the string `B = cdcdcdc`, and after encoding it, we end up with `C(B) = 0x63648082`. Following the above procedure results in this table:

Code	String
0x61	a
0x62	b
0x63	c
...	...
0x7e	~
0x7f	<DEL>
0x80	cd
0x81	dc
0x82	???

B = cdcd???

The problem is that we could look ahead while coding, but can only look behind while decoding. So we must try to figure out what `0x82` is going to be by looking back.

Let's say `0x82` is some sequence `Z` to be figured out. We previously decoded `0x80 = cd`, and now we have `0x82 = Z`, so we will add `cdZ_0` to the table as `0x82`.

So `Z` starts with `0x80` and therefore `Z_0` must be `c`! This means `0x82` is `cdc`.

The LZW algorithm is named for its inventors: Lempel, Ziv and Welch. It was once widely used, but patent issues made it rather unpopular. Those patents expired in 2003 and 2004, and it is now found in `.gif` files, some PDF files, the BSD Unix `compress` utility, and elsewhere. There are numerous other, better algorithms, such as those uses in `gzip` and `bzip2`.

The presentation here is considerably simplified:

- We used fixed-length (8-bit) codewords, but the full algorithm produces variable-length codewords using (!) Huffman coding (compressing the compression).
- The full algorithm clears the table from time to time to get rid of little-used codewords.

Conclusion

Compressing a compressed text doesn't result in much compression. Why must it be impossible to keep compressing a text? Otherwise you'd be able to compress any number of different messages to 1 bit!

A program that takes no input and produces an output can be thought of as an encoding of that output, which leads to the following question: Given a bitstream, what is the length of the shortest program that can produce it? For any specific bitstream, there is a specific answer! This is a deep concept, known as Kolmogorov Complexity.

It's actually weird that one can compress much at all. In a 1000-character ASCII text (8000 bits), and suppose we compress it by 50%. There are 2^8000 distinct messages in 8000 bits, but only 2^4000 possible messages in 4000 bits.

That is, no matter what one's scheme, we can only encode 1 in 2^4000 of the possible 8000-bit messages by 50%! Yet, we do this all the time. The reason is that our texts have a great deal of redundancy and repetition, called low information entropy. Texts with high entropy, such as random bits, previously compressed texts, or encrypted texts, are nearly imcompressible.

Git

Git actually uses a different scheme from LZW for compression: a combination of LZ77 and Huffman coding. LZ77 is kind of like delta compression, but within the same text. For example:

`One Mississippi, two Mississippi` is compressed into something like

`One Mississippi, two <11,7>`, where the `<11,7>` is intended to mean "the next 11 characters come from the text that ends 7 characters before this point."

We add new symbols to the alphabet to represent these (length, distance) inclusions. When done, Huffman encode the result.

Lossy Compression

For some applications, like compressing video and audio streams, it really isn't necessary to be able to reproduce the exact stream. • We can therefore get more compression by throwing away some information. The reason is that there is a limit to what human senses respond to. For example, we don't hear high frequencies, or see tiny color variations, and thus, formats like JPEG, MP3, or MP4 use lossy compression and reconstruct output that is (hopefully) imperceptibly different from the original at large savings in size and bandwidth. You can see more of this in EE 120 and other courses.

Summary

Lossless compression saves space (and bandwidth) by exploiting redundancy in data.

- Huffman and Shannon-Fano coding represent individual symbols of the input with shorter codewords.
- LZW and similar codes represents multiple symbols with shorter codewords.
- Both adapt their codewords to the text being compressed.

Lossy compression both uses redundancy and exploits the fact that certain consumers of compressed data (like humans) can't really use all the information that could be encoded.