Lecture 5: Adaptive Prefix-Free Coding

Lecturer: Travis

January 27th, 2015

In the first week we discussed so-called *static* prefix-free compression algorithms (such as Morse coding) that don't adapt to their input, and *semi-static* algorithms (such as Huffman coding) that use two passes over their input (one to compute the characters' frequencies and another to encode).

Today we're going to talk about *adaptive* prefix-free compression algorithms — also known as *dynamic* or *one-pass* — which use only one pass over their input and adjust the code they use as they go along, so that it *adapts* to the frequencies of the characters (so far).

(By the way, prefix-free codes are sometimes called *instantaneous* because we can encode each character immediately after reading it and decode it immediately after reading its encoding.)

Recall the situation of Alice sending a message to Bob by either using an agreed-upon code or first sending him a code, and sending the encoding of each character? How does that work if Alice starts changing the code?

If Alice uses an agreed-upon code to send the first character, then updates her copy of the code in a deterministic way that depends only on that character, then Bob — who now knows the first character — can update his copy of the code in the same way.

In general, after sending the first i characters of the message, Alice can update the code in a deterministic way that depends only on the current state of the code and those i characters.

One of the simplest adaptive prefix-free coding algorithms is Move-To-Front (MTF). Alice and Bob start with the same list of characters in the alphabet. Alice reads each character of the message, sends its current position in the list to Bob, then moves it to the front of the list. When Bob receives each position, he learns the character, which he also moves to the front of the list.

Of course, Alice has to send each position in a prefix-free manner. To do this, she can use one of the integer codes you learned about last week, such as Elias' gamma or delta codes.

Elias himself published a paper in 1986 proposing MTF, but he was beaten to it by Ryabko and Bentley, Sleator, Tarjan and Wei. Most people cite Bentley et al. because they described data structures that allow us to run MTF in time proportional to the length of the encoding.[1]

[1] It's pretty easy to do this with modern data structures, but that's partly because of the techniques these guys invented around this time (e.g., splay trees — which use a kind of MTF — and amortized analysis).

ABRACADABRA

ABRACADABRA

A, B, C, D, R

ABRACADABRA

A, B, C, D, R $\qquad \gamma(1)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$
D, A, C, R, B	$\gamma(2)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$
D, A, C, R, B	$\gamma(2)$
A, D, C, R, B	$\gamma(5)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$
D, A, C, R, B	$\gamma(2)$
A, D, C, R, B	$\gamma(5)$
B, A, D, C, R	$\gamma(5)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$
D, A, C, R, B	$\gamma(2)$
A, D, C, R, B	$\gamma(5)$
B, A, D, C, R	$\gamma(5)$
R, B, A, D, C	$\gamma(3)$

ABRACADABRA

A, B, C, D, R	$\gamma(1)$
A, B, C, D, R	$\gamma(2)$
B, A, C, D, R	$\gamma(5)$
R, B, A, C, D	$\gamma(3)$
A, R, B, C, D	$\gamma(4)$
C, A, R, B, D	$\gamma(2)$
A, C, R, B, D	$\gamma(5)$
D, A, C, R, B	$\gamma(2)$
A, D, C, R, B	$\gamma(5)$
B, A, D, C, R	$\gamma(5)$
R, B, A, D, C	$\gamma(3)$
A, R, B, A, D	$\gamma(6)$

(here $\gamma(6)$ means "end of string")

For $i \geq 1$, the codeword $\gamma(i)$ consists of $\lceil \log(i+1) \rceil - 1$ copies of 0 followed by the $\lceil \log(i+1) \rceil$-bit binary representation of i, so $|\gamma(i)| \approx 2 \lg i$.

$$\begin{array}{rcl} \gamma(1) & = & 1\,, \\ \gamma(2) & = & 010\,, \\ \gamma(3) & = & 011\,, \\ \gamma(4) & = & 00100\,, \\ \gamma(5) & = & 00101\,, \\ & \vdots & \end{array}$$

For $i \geq 1$, the codeword $\delta(i)$ consists of $\gamma(\lceil \log(i+1) \rceil)$ followed by the $\lceil \log(i+1) \rceil$-bit binary representation of i with the leading 1 removed, so $|\delta(i)| \approx \lg i + 2 \lg \lg(i+1)$.

$$\begin{array}{rcl}
\delta(1) & = & 1\,, \\
\delta(2) & = & 0100\,, \\
\delta(3) & = & 0101\,, \\
\delta(4) & = & 01100\,, \\
\delta(5) & = & 01101\,, \\
& \vdots &
\end{array}$$

So how long is the final encoding? Well, at any point the position of any character in the list is at most the time since it last occurred, or σ if it has never occurred. Therefore, if a character x occurs $\mathrm{occ}(x, S)$ times in the message $S[1..n]$, then the sum of the base-2 logarithms of the positions we send for those occurrences is bounded from above by

$$\lg \sigma + (\mathrm{occ}(x, S) - 1) \lg \frac{n - 1}{\mathrm{occ}(x, S) - 1} \, .$$

Summing over all the characters, we find that the sum of the logs of the positions is bounded by $nH_0(S) + \sigma \lg \sigma$. (Recall that $H_0(S)$ is the 0th-order empirical entropy of S, which is the entropy of the distribution of characters in S.) All good so far, but what about the cost of using a prefix-free code to send the positions?

With the overhead of the delta code included, by Jensen's Inequality the bound is something like $nH_0(S) + 2n \lg H_0(S) + \sigma \lg \sigma$. That's pretty good, and MTF is still used all over the place, including in lots of BWT-based compression algorithms.

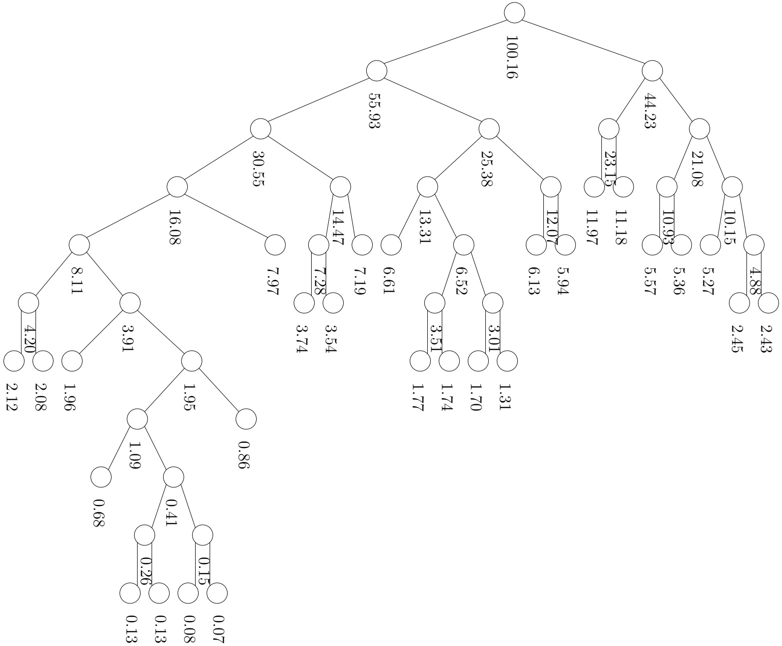
Still, can we do better? For example, what if every time Alice sends a character, she builds a new Huffman code — or updates the one she has — for the prefix of S she's encoded so far, with an extra escape character she can use to send characters that haven't appeared yet?

In practice, this is a bit masochistic. If we really think the characters of S are drawn i.i.d. — which is when Huffman coding the characters is actually a good choice — then we can just wait for a while, build a Huffman code, wait a while longer, build another Huffman code, wait much longer, build a third Huffman code, wait even longer. . . and probably not have to build another code, since the frequencies will almost certainly not change much.

Nevertheless, people — Faller in 1973, Gallagher in 1978 and Knuth in 1985 — figured out how to update a Huffman code quickly when one of the characters' frequencies is incremented. The algorithm described by these three researchers is known as FGK, for obvious reasons. They *didn't* analyse the length of the encoding, though.

Lemma (Sibling Property)

Suppose that a strict binary tree T has non-negative weights assigned to its leaves and each internal node is assigned the sum of the weights of its children. Then T is a Huffman tree for its leaf weights if all nodes can be placed in order of non-decreasing weight such that siblings are adjacent and children precede parents.


100.16
55.93
44.23
30.55
25.38
23.15
21.08
16.08
14.47
13.31
12.07
11.97
11.18
10.93
10.15
8.11
7.97
7.28
7.19
6.61
6.52
6.13
5.94
5.57
5.36
5.27
4.88
4.20
3.91
3.74
3.54
3.51
3.01
2.45
2.43
2.12
2.08
1.96
1.95
1.77
1.74
1.70
1.31
1.09
0.86
0.68
0.41
0.26
0.15
0.13
0.13
0.08
0.07

Vitter gave an improved algorithm in 1986, showed his algorithm uses at most 1 more bit per character than semi-static Huffman coding, showed that this is optimal for adaptive Huffman coding, and showed FGK uses at most twice as many bits as semi-static Huffman. Milidiu, Laber and Pessoa showed in 2000 that FGK uses at most 2 more bits per character than semi-static Huffman.

Theorem

Vitter's algorithm for adaptive Huffman coding stores $S[1..n]$ using at most $nH_0(S) + 2n + o(n)$ bits and $\mathcal{O}(nH_0(S) + n)$ time for encoding and decoding.[2].

[2]Assuming $\sigma \ll n$ for simplicity.

(We can also use an adaptive Huffman code based only on the characters' frequencies is a sliding window that contains the part of S we have just encoded. This works fairly well in practice when we choose the window length well, but messes up the theory a bit.)

FGK and Vitter's algorithm maintain a code-tree and encode and decode via root-to-leaf descents. As we discussed in the first week, it's much faster to use a canonical code. There were a series of papers in the '90s by an Australian called Alistair Moffat — Simon's PhD supervisor's PhD supervisor, I think, and Simon's first post-doc supervisor — on adaptive canonical Huffman coding. Moffat cares more about practice, though, so he didn't bother proving theorems.

People worked on adaptive Huffman coding rather than adaptive Shannon coding because semi-static Huffman beats semi-static Shannon (and because adaptive Huffman works better in practice). About tens years ago, though, some people started trying to prove better bounds for adaptive Shannon. It turned out to be really easy to maintain an adaptive canonical Shannon code.

Theorem

The best algorithm for adaptive Shannon coding stores $S[1..n]$ *using at most* $nH_0(S) + n + o(n)$ *bits and* $\mathcal{O}(1)$ *worst-case time for encoding and decoding each character.*[3]

(Notice these are the same bounds as for semi-static Shannon coding.)

[3] Again assuming $\sigma \ll n$ for simplicity.

You should understand and remember MTF. You don't need to remember anything about adaptive Huffman or Shannon coding once you've done next week's exercise.