1 Ziv-Lempel data compression

Large text or graphics files are often compressed in order to save storage space or to speed up transmission when the file is shipped. Most operating systems have compression utilities, and some file transfer programs automatically compress, ship, and uncompress the file, without user intervention. The field of text compression is itself the subject of several books (for example see [8]), and will not be handled in depth here. However, a popular compression method due to Ziv-Lempel [9, 10] has an efficient implementation using suffix trees [6], providing another illustration of their utility.

The Ziv-Lempel compression method is widely used (it is the basis for the Unix utility *compress*), although there are actually several variants of the method that go by the same name (see [9, 10]). In this section, we present a basic variant of the method and an efficient implementation of it using suffix trees.

Definition For any position i in a string S of length m, define the substring $Prior_i$ to be the longest prefix of $S[i..m]$ that also occurs as a substring of $S[1..i-1]$.

For example, if $S = abaxcabaxabz$ then $Prior_7$ is bax.

Definition For any position i in S, define l_i as the length of $Prior_i$. For $l_i > 0$, define s_i as the starting position of the leftmost copy of $Prior_i$.

In the above example, $l_7 = 3$ and $s_7 = 2$.

Note that when $l_i > 0$, the copy of $Prior_i$ starting at s_i is totally contained in $S[1..i-1]$.

The Ziv-Lempel method uses some of the l_i and s_i values to construct a compressed representation of string S. The basic insight is that if the text $S[1..i-1]$ has been represented (perhaps in compressed form) and l_i is greater than zero, then the next l_i characters of S (substring $Prior_i$) need not be explicitly described. Rather, that substring can be described by the pair (s_i, l_i), pointing to an earlier occurrence of the substring. Following this insight, a compression method could process S left to right, outputting the pair (s_i, l_i) in place of the explicit substring $S[i..i+l_i-1]$ when possible, and outputting the character $S(i)$ when needed. Full details are given in the algorithm below.

Compression Algorithm One
begin
$i := 1$
Repeat

    compute l_i and s_i
    if $l_i > 0$ then
    begin

```
        output (s_i, l_i)
        i := i + l_i
    end
    else
    begin
        output S(i)
        i := i + 1
    end

Until i > n

end.
```

For example, $S = abacabaxabz$ can be described as $ab(1,1)c(1,3)x(1,2)z$. Of course, in this example the number of symbols used to represent S did not decrease, but rather increased! That's typical of small examples. But as the string length increases, providing more opportunity for repeating substrings, the compression improves. Moreover, the algorithm could choose to output character $S(i)$ explicitly whenever l_i is "small" (the actual rule depends on bit level considerations determined by the size of the alphabet, etc.) For a small example where positive compression is observed, consider the contrived string $S = abababababababababababababababab$, represented as $ab(1,2)(1,4)(1,8)(1,16)$. That representation uses 24 symbols in place of the original 32 symbols. If we extend this example to contain k repeated copies of ab, then the compressed representation contains approximately $5 \log_2 k$ symbols, a dramatic reduction in space.

To decompress a compressed string, process the compressed string left to right, so that any pair (s_i, l_i) in the representation points to a substring that has already been fully decompressed. That is, assume inductively that the first j terms (single characters or s, l pairs) of the compressed string have been processed, yielding characters 1 through $i-1$ of the original string S. The next term in the compressed string is either character $S(i+1)$, or it is a pair (s_i, l_i) pointing to a substring of S strictly before i. In either case, the algorithm has the information needed to decompress the j'th term, and since the first term in the compressed string is the first character of S, we conclude by induction that the decompression algorithm can obtain the original string S.

1.1 Implementation using suffix trees

The key implementation question is how to compute l_i and s_i each time the algorithm requests those values for a position i. The algorithm compresses S left to right and does not request (s_i, l_i) for any position i already in the compressed part of S. The compressed substrings are therefore non-overlapping, and if each requested pair (s_i, l_i)

can be found in $O(l_i)$ time, then the entire algorithm would run in $O(m)$ time. Using a suffix tree for S, the $O(l_i)$ time bound is easily achieved for any request.

Before beginning the compression, the algorithm first builds a suffix tree $\mathcal{T}$ for S and then numbers each node v with the number c_v. This number is the smallest suffix (position) number of any leaf in v's subtree, and it gives the leftmost starting position in S of any copy of the substring that labels the path from r to v. The tree can be built in $O(m)$ time, and all the node numbers can be obtained in $O(m)$ time by any standard tree traversal method (or bottom up propagation).

When the algorithm needs to compute (s_i, l_i) for some position i, it traverses the unique path in $\mathcal{T}$ that matches a prefix of $S[i..m]$. The traversal ends at point p (not necessarily a node) either when no further matches are possible, or when i equals the string-depth of point p plus the number c_v, where v is the first node at or below p. In either case, the path from the root to p describes the longest prefix of $S[i..m]$ that also occurs in $S[1..i]$. So, s_i equals c_v and l_i equals the string-depth of p. Exploiting the fact that the alphabet is fixed, the time to find (s_i, l_i) is $O(l_i)$. So the entire compression algorithm runs in $O(m)$ time.

1.2 A one-pass version

The implementation above assumes that S is known ahead of time and that a suffix tree for S can be built before compression begins. That is fine in many contexts. But the method can also be modified to operate on-line as S is being input, one character at a time. Essentially, the algorithm is implemented so that the compaction of S is interwoven with the construction of $\mathcal{T}$. The easiest way to see how to do this is with Ukkonen's linear-time suffix tree algorithm.

Ukkonen's algorithm builds *implicit* suffix trees on-line as characters are added to the right end of the growing string. Assume that the compaction has been done for $S[1..i-1]$, and that implicit suffix tree $\mathcal{I}_{i-1}$ for string $S[1..i-1]$ has been constructed. At that point, the compaction algorithm needs to know (s_i, l_i). It can obtain that pair in exactly the same way that is done in the above implementation *if* the c_v values have been written at each node v in $\mathcal{I}_{i-1}$. But unlike the above implementation which establishes those c_v values in a linear time traversal of $\mathcal{T}$, the algorithm cannot traverse each of the implicit suffix trees, since that would take more than linear time overall. Instead, whenever a new internal node v is created in Ukkonen's algorithm by splitting an edge (u, w), c_v is set to c_w, and whenever a new leaf v is created, c_v is just the suffix number associated with leaf v. In this way, only constant time is needed to update the c_v values when a new node is added to the tree. In summary,

Theorem 1.1 *Compression algorithm one can be implemented to run in linear time as a one-pass, on-line algorithm to compress any input string S.*

1.3 The real Ziv-Lempel

The compression scheme given in *Compression algorithm one* is not the actual Ziv-Lempel method, but is close to it and does capture its spirit. The real Ziv-Lempel method is a one-pass algorithm whose output differs from the output of *Compression algorithm one* in that, whenever it outputs a pair (s_i, l_i), it then explicitly outputs $S(i+l_i)$, the character following the substring. For example, $S = abababababababababababababababab$ would be compressed to $ab(1,2)a(2,4)b(1,10)a(2,12)b$, rather than as $ab(1,2)(1,4)(1,8)(1,16)$. The one-pass version of *Compression algorithm one* can trivially be converted to implement Ziv-Lempel in linear time.

It is not completely clear why the Ziv-Lempel algorithm outputs the extra character. Certainly for compaction purposes, this character is not needed and seems extraneous. One suggested reason for outputing an explicit character after each (s, l) pair is that $(s_i, l_i)S(i + l_i)$ defines the *shortest* substring starting at position i which *does not* appear anywhere earlier in the string, while (s_i, l_i) defines the longest substring starting at i which does appear earlier. Historically, it may have been easier to reason about shortest substrings that do not appear earlier in the string, than to reason about longest substrings which do appear earlier.

2 APL17: Minimum length encoding of DNA

Recently, several molecular biology and computer science research groups have used the Ziv-Lempel method to compress DNA strings, not for the purpose of efficient storage, but to compute a measure of the "complexity" or "information content" of the strings [7, 1, 5, 4]. Without fully defining the central technical terms "complexity", "information", "entropy", etc., the basic idea is that substrings of greatest biological significance should be more compressable than substrings that are essentially random. One expects that random strings will have too little structure to allow high compression, since high compression is based on finding repetitive segments in the string. Therefore, by searching for substrings that are more compressable than random strings, one may be able to find strings that have a definite biological function.

Compression has also been used to study the "relatedness"[1] of two strings S_1 and S_2 of DNA [1, 3]. Essentially, the idea is to build a suffix tree for S_1 and then compress string S_2 using only the suffix tree for S_1. That compression of S_2 takes advantage of substrings in S_2 that appear in S_1, but does not take advantage of repeated substrings in S_2 alone. Similarly, S_1 can be compressed using only a suffix tree for S_2. These compressions reflect and estimate the "relatedness" of S_1 and S_2. If the two strings are highly related, then both computations should significantly compress the string at hand.

[1] Other, more common ways to study the relatedness or similarity of strings of two strings are extensively discussed in Part III.

Another biological use for ZL-like algorithms is to estimate the "entropy" of short strings in order to discriminate between exons and introns in eukaryotic DNA [2]. They report that the average compression of introns does not differ significantly from the average compression of exons, and hence compression by itself does not distinguish exons from introns. However, they also report that the following extension of that approach is effective in distinguishing exons from introns.

Definition For any position i in string S, let $ZL(i)$ denote the length of the longest substring beginning at i that appears somewhere in the string $S[1..i]$.

Definition Given a DNA string S partitioned into exons and introns, the *exon-average ZL value* is the average $ZL(i)$ taken over every position i in the exons of S. Similarly, the *intron-average ZL* is the average $ZL(i)$ taken over positions in introns of S.

It should intuitive at this point that the exon-average ZL value and the intron-average ZL value can be computed in $O(n)$ time, by using suffix trees to compute all the $ZL(i)$ values. The technique is similar to the way matching statistics are computed, but more involved since the substring starting at i must also appear to the left of position i.

The main empirical result of [2] is that the exon-average ZL value is lower than the intron-average ZL value by an amount that is statistically significant. That result is contrary to the expectation stated above that biologically significant substrings (exons in this case) should be more compressable than more random substrings (which introns are believed to be). Hence, the full biological significance of string compresability is still an open question.

References

[1] L. Allison and C.N. Yee. Minimum message length encoding and the comparison of macro-molecules. *Bull. of Math. Biology*, 52:431–453, 1990.

[2] M. Farach, M. Noordewier, S. Savari, L. Shepp, A. Wyner, and J. Ziv. On the entropy of dna: Algorithms and measurements based on memory and rapid convergence. *Proc. 6'th ACM-SIAM Symp. on Discrete Algs.*, pages 48–57, 1995.

[3] A. Milosavljevic. Discovering dependencies via algorithmic mutual information: A case study in dna sequence comparisons. *Maching Learning*, 21:35–50, 1995.

[4] A. Milosavljevic and J. Jurka. Discovering simple dna sequences by the algorithmic significance method. *Maching Learning*, 9:407–411, 1993.

[5] A. Milosavljevic and J. Jurka. Discovery by minimal length encoding: A case study in molecular evolution. *Maching Learning*, 12:69–87, 1993.

[6] M. Rodeh, V.R. Pratt, and S. Even. A linear algorithm for data compression via string matching. *J. ACM*, 28:16–24, 1981.

[7] P. Salamon and A. Konopka. A maximum entropy principle for distribution of local complexity in naturally occurring nucleotide sequences. *Computers and Chemistry*, 16:117–124, 1992.

[8] J. A. Storer. *Data Compression: Methods and theory*. Computer Science Press, 1988.

[9] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Info. Theory*, 23:337–343, 1977.

[10] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. on Info. Theory*, 24:530–536, 1978.