Exploring Data Compression via Binary Trees[1]

Mark Daniel Ward
Purdue University

Summary

We investigate the Lempel-Ziv '77 data compression algorithm by considering an analogous algorithm for efficiently embedding strings in binary trees. This project includes a discussion of this comparison with two optional addenda on error correction and decompression, followed by exercises and solutions.

Notes for the instructor

Students in discrete mathematics often have a dual interest in computer science. This project succinctly combines these two areas. Data compression can be viewed as a discrete mathematics topic with many ramifications for computer scientists. Students who have completed one or two semesters of computer science (in particular, who are familiar with trees) may be eager to implement the algorithms discussed in C++, Java, or another object-oriented programming language.

The Lempel-Ziv '77 data compression algorithm was introduced in [1]. Analysis of the multiplicity matching parameter of suffix trees was presented in the present author's Ph.D. thesis; an abridged journal version with many more references to the literature can be found in [3]. An error correcting version of LZ'77 is outlined in [2].

Bibliography

[1] Lempel, A. and J. Ziv. "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory* 23 (1977) 337–343.

[2] Lonardi, S., W. Szpankowski, and M. D. Ward. "Error resilient LZ'77 data compression: algorithms, analysis, and experiments," *IEEE Transactions on Information Theory* 53 (2007) 1799–1813.

[3] Ward, M. D. and W. Szpankowski. "Analysis of the multiplicity matching parameter in suffix trees," *Discrete Mathematics and Theoretical Computer Science* AD (2005) 307–322, available online at `http://www.dmtcs.org/proceedings/abstracts/dmAD0128.abs.html`.

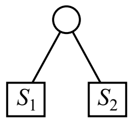
[1]The author thanks his Ph.D. advisor, Wojciech Szpankowski, for guidance and encouragement throughout graduate school at Purdue University (2001–2005).

Exploring Data Compression and Error Correction

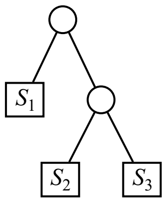
We first discuss a standard method of embedding binary strings into a tree re*trie*val structure, often abbreviated as a "trie". A trie is a rather efficient tree structure. Every node has at most two children. When a node has no children, we refer to it as a leaf. A string is inserted at the minimal depth necessary; in other words, each string is inserted at the earliest depth that allows it to be distinguished from all other strings that currently reside in the trie. A "0" in a string corresponds to a left branch in the trie; a "1" corresponds to a right branch. Consider two strings:

$$S_1 = 0010111011$$
$$S_2 = 1001010101$$

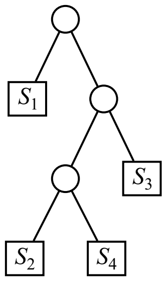
When we embed S_1 and S_2 into a trie, we can immediately distinguish these two strings just by examining their first elements, which are 0 and 1, respectively.



If we embed another string, say $S_3 = 1110001010$, we note that the first bit of S_1 already distinguishes S_1 from S_3, but two bits of S_3 are necessary to make a distinction from S_2. After S_3 is embedded into the above trie, we have



If S_4 has a long prefix in common with S_1, for instance, $S_4 = 1010111100$, then we have



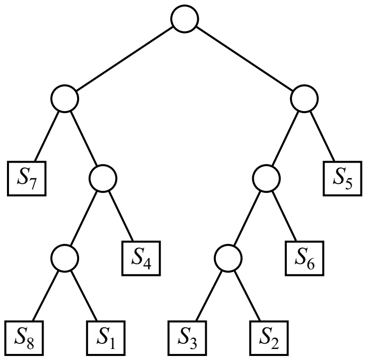
Currently four strings S_1, S_2, S_3, and S_4 have been inserted into the trie. Suppose that $S = S_i$ for some i. If S begins with 0, we know immediately that $S = S_1$. Otherwise, S begins with 1, so S is found to the right of the root of the trie. If S begins with 11, then $S = S_3$; otherwise, S begins with 10, and in this case, we proceed one level further into the trie to determine if $S = S_2$ (equivalently, S begins with 100) or $S = S_4$ (i.e., S begins with 101). This example illustrates an unambiguous way to efficiently embed binary strings into a binary tree. Many tree structures are found in the literature and have been extensively analyzed; the trie structure given above is one of the most popular and frequently analyzed.

For another example, consider the following eight strings:

$$S_1 = 0101010100$$
$$S_2 = 1001001001$$
$$S_3 = 1000000111$$
$$S_4 = 0111010000$$

$$S_5 = 1100101011$$
$$S_6 = 1010010110$$
$$S_7 = 0011001001$$
$$S_8 = 0100001000$$

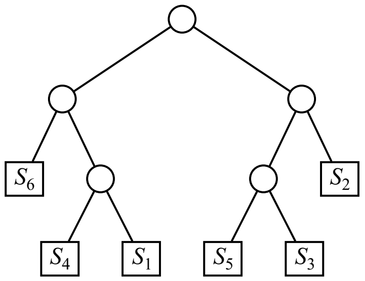
The associated trie is



Next, we discuss a particular type of trie. In the examples above, the strings S_i were not dependent on each other; the strings were generated independently. We now consider strings that are generated as suffixes of a common word. We write X for a long binary string, for instance, $X = 01101001001110011010\ldots$. Then we write S_1 to denote X itself. We let S_2 denote X with only the first character removed (so S_2 is the second suffix of X). We let S_3 denote X with only the first and second characters removed (so S_3 is the third suffix of X). So we have

$$S_1 = 01101001001110011010\ldots$$
$$S_2 = 1101001001110011010\ldots$$
$$S_3 = 101001001110011010\ldots$$
$$S_4 = 01001001110011010\ldots$$
$$S_5 = 1001001110011010\ldots$$
$$S_6 = 001001110011010\ldots$$

Computer scientists often use the term “suffix tree” to denote the trie constructed from the suffixes of a word. So the suffix tree associated with $S_1, \ldots, S_6$ is constructed by the same method that the tries above were constructed. Therefore, the suffix tree built from the first six suffixes of X is



In the 1970s, Jacob Ziv and Abraham Lempel introduced a variety of sequential data compression algorithms. In particular, they presented LZ’77 and LZ’78, two schemes that still pervade every aspect of modern data compression theory and practice. Although thirty years have passed since these algorithms were introduced, these algorithms still remain popular and widely-used. Below, we describe a slightly simplified version of LZ’77.

Keeping in mind the suffix trees constructed above, we now discuss the motivating concept for LZ'77. Our goal is to compress a binary string, performing the compression one block at a time. To see the novel idea behind LZ'77, take a second look at the suffix tree associated with

$$X = 01101001001110011010\ldots$$

in the last example above. Notice that S_4 and S_1 both have several characters at the beginning in common. For this reason, S_1 and S_4 are placed close together in the suffix tree; both of these strings begin with the prefix 01. We can make the following three observations:

1. The string S_4 is very comparable to the string S_1.
2. Both of these strings have a common prefix of length 2.
3. The next character of S_4 after this common prefix is 0.

So we have described the first three characters of S_4 by noticing that there is a block of length 2 (namely, 01) in common with S_1, and then the third character of S_4 is 0. So $S_4 = 010\ldots$. This is the inspiration behind LZ'77. (As a side note, in step 3, we encode the next character explicitly so that the algorithm never gets "stuck." This reasoning for this extra character is crucial for the LZ'77 algorithm but admittedly will seem mysterious to those who are new to the algorithm. Nonetheless, we insist that step 3 above is crucial in LZ'77 for preventing the algorithm from getting stuck.)

Next we make a general statement of the algorithm; afterwards, we consider several illustrative examples.

We compress a binary string $X = X_1X_2X_3\ldots$ in blocks, starting from the beginning. Consider the stage of the compression when the first n bits (namely, $X_1X_2\ldots X_n$) have already been compressed. Then $S_{n+1} = X_{n+1}X_{n+2}X_{n+3}\ldots$ remains uncompressed.

In order to compress the next block of S_{n+1}, we consider the various strings S_{i+1}, for $0 \le i < n$. We aim to find the value of i such that S_{n+1} and S_{i+1} have the longest prefix in common. We find this desired value of i, and we write L to denote the length of the longest common prefix. Then we are able to compress the first $L + 1$ characters of S_{n+1} by noting that:

1. The string S_{n+1} is very comparable to S_{i+1}.
2. Both of these strings have a common prefix of length L.
3. The next character of S_{n+1} after this common prefix $X_{n+1}X_{n+2}\ldots X_{n+L}$ is exactly X_{n+L+1}.

Notice the similarity in these three steps to our earlier comparison of S_1 and S_4. Despite the notation, our comparison of $S_{n+1} = X_{n+1}X_{n+2}X_{n+3}\ldots$ to $S_{i+1} = X_{i+1}X_{i+2}X_{i+3}$ is just the same idea as found in our comparison of S_1 to S_4.

Finally, we note that the ingenuity behind Lempel and Ziv's LZ'77 scheme is that, rather than storing S_{n+1}, it is often more efficient to store the following three pieces of information:

1. a pointer to X_{i+1}, i.e., to the beginning of S_{i+1},
2. the length L of the common prefix,
3. the next character after this common prefix $X_{n+1}X_{n+2}\ldots X_{n+L}$, namely, X_{n+L+1}.

(In fact, this is a provably more efficient scheme when X is generated with a Markov dependency.) At any rate, it is most helpful to consider some examples.

Example 1. Suppose $X = 101101000110\ 00010100\ldots$ and $n = 12$ (the small space in X is just written for the clarity of the reader in this example; of course X does not contain any actual gaps or spaces). So we are considering the stage where the first twelve bits of X, namely $X_1\ldots X_{12} = 101101000110$, have already been compressed. The rest of X, namely $S_{13} = X_{13}X_{14}\ldots = 00010100\ldots$, is still uncompressed.

In the notation given above, we have

$$S_{n+1} = X_{n+1}X_{n+2}\dots = 00010100\dots$$

We should compare this string to $S_{i+1} = X_{i+1}X_{i+2}\dots$ for $i = 6$, because $S_7 = X_7X_8\dots$ has a prefix of length $L = 4$ (namely, 0001) in common with S_{13}. Thus, the first five characters of S_{13} can be efficiently compressed by:

1. storing a pointer to X_7,
2. recording the length $L = 4$ of the common prefix between $S_7 = X_7X_8\dots$ and $S_{13} = X_{13}X_{14}\dots$,
3. noting that the next character after this common prefix $X_{13}X_{14}X_{15}X_{16}$ is $X_{17} = 0$.

By this method, $X_{13}X_{14}X_{15}X_{16}X_{17}$ efficiently gets compressed. The algorithm proceeds to the next step, with $X_1\dots X_{17}$ already compressed; the remainder $S_{18} = X_{18}X_{19}\dots$ still needs to be compressed.

Incidentally, when constructing the suffix tree built on $S_1, S_2, \dots, S_{13}$, one notices that S_7 and S_{13} are siblings. This is not a coincidence; the general situation will be described at the end of the next example.

Example 2. Suppose $X = 0110110\,01110100000100010\dots$ and $n = 7$. So the first seven bits of X, namely $X_1\dots X_7 = 0110110$, have already been compressed. The rest of X, namely $S_8 = X_8X_9\dots = 01110100000100010\dots$, has not yet been compressed.

We should compare S_8 to $S_{i+1} = X_{i+1}X_{i+2}\dots$ for $i = 0$ or $i = 3$, because $S_1 = X_1X_2\dots$ and $S_4 = X_4X_5\dots$ each have a prefix of length $L = 3$ (namely, 011) in common with S_8. So, the first four characters of $S_{n+1} = X_{n+1}X_{n+2}\dots$ can be efficiently compressed with the usual three steps:

1. storing a pointer to X_1 (or a pointer to X_4; either pointer is OK to use),
2. recording the length $L = 3$ of the common prefix between S_1 and S_8,
3. noting that the next character after this common prefix $X_8X_9X_{10}$ is $X_{11} = 1$.

By this method, $X_8X_9X_{10}X_{11}$ is efficiently compressed. The algorithm proceeds to the next step, with $X_1\dots X_{11}$ already compressed, and $S_{12} = X_{12}X_{13}\dots$ is still uncompressed.

Consider the suffix tree built on $S_1, S_2, \dots, S_8$. We note that S_8's parent has S_1 and S_4 as descendants. This is true in general: the parent of the uncompressed string (in this case, the parent of S_8) will be an ancestor of the appropriate strings S_i used to perform the compression (in this case, S_1 and S_4). In fact, all descendants of the parent of the uncompressed string are appropriate candidates for performing the compression. Drawing the relevant suffix trees is extremely helpful for illustrating the situation.

Addendum about Error Correction

For those readers wishing to explore research in this vein: Whenever two or more values of i are valid (i.e., two or more strings S_i are available for use in the compression), then some error correction can be performed at this stage in the compression. In fact, this author dedicated his entire Ph.D. thesis to a precise study of how much error correction can be performed in such a situation. The number of S_i's available is called the multiplicity matching parameter; see the References section for more details.

At each stage of the compression, whenever there are multiple pointers to choose from, let M_n denote the number of valid values of i when compressing $X_{n+1}X_{n+2}\ldots$; in the example just given, $M_7 = 2$ because $i = 0$ or $i = 3$ could be used. Then we can embed $\lfloor \log_2 M_n \rfloor$ extra bits at no extra cost whatsoever. (Of course, for many n, we have $M_n = 1$, and no parity check can be performed.)

As an example, when $M_n = 2$, we can perform a parity check, choosing the first pointer if a "1" is required in our parity check, or selecting the second pointer if a "0" is required in our parity check. For example, the algorithm can check the parity of the bits compressed up until the present stage. If an error is detected, a warning message about the inconsistency could be given, or some error recovery could be performed. Such methods allow students a great degree of freedom and creativity, because there are many possible ways of implementing the algorithm in an object-oriented computer language.

Addendum about Decompression

Students who are quite eager to implement the LZ'77 algorithm with these extra features will certainly want to know how to write decompression algorithms too. Such files are uncompressed in an analogous way to the method by which they are compressed. The decompression is performed block-by-block, starting at the beginning of the file. In order to decompress the next block, we consider the stage at which n bits have already been decompressed, say $X_1X_2\ldots X_n$. In the compressed version of the file, we encounter a triple that describes a pointer, a length, and a next bit. The pointer to some X_{i+1} and the length L tell us to copy $X_{i+1}X_{i+2}\ldots X_{i+L}$ into the next L positions, namely $X_{n+1}X_{n+2}\ldots X_{n+L}$. The next bit is exactly X_{n+L+1}. By repeatedly decoding triples, we reproduce the original (uncompressed) file.

We note that students who are interested in both discrete mathematics and computer science will perhaps enjoy implementing the LZ'77 program in C++, Java, or another object-oriented programming language. Afterwards, they are able to measure the redundancy present in this aspect of LZ'77 by computing and tabulating M_n at each step during the execution of the program. They are also able to experiment with a variety of uses for the redundant bits that are available at most stages of the compression. Students can devise various error correction schemes or image embedding techniques. See the references for a published implementation of the LZ'77 algorithm with error correcting extensions.

Questions on Exploring Data Compression via Binary Trees

1. Construct the trie associated with the following eight strings:

$$\begin{aligned}S_1 &= 0001000011\\ S_2 &= 0011011111\\ S_3 &= 0101010110\\ S_4 &= 0010000101\\ S_5 &= 1110000001\\ S_6 &= 1001010001\\ S_7 &= 0000010011\\ S_8 &= 1100001001\end{aligned}$$

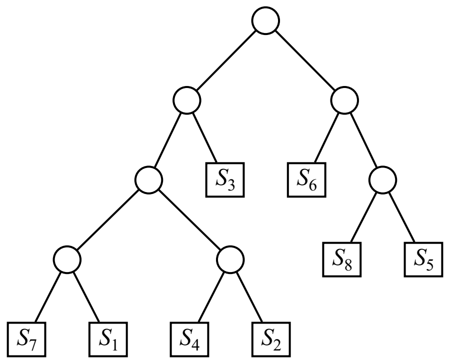
2. Consider the string $X = 01011001111101001110\ldots$. Let $S_1, S_2, \ldots, S_8$ denote the first eight suffixes of X. Construct the suffix tree associated with these eight strings.

3. Consider the string $X = 110110001010\ 10010011\ldots$; suppose that the first twelve characters (namely, $X_1 \ldots X_{12} =$ 110110001010) have been compressed and the remainder (i.e., $S_{13} = 10010011\ldots$) is uncompressed. What is the next block to be compressed in the LZ'77 algorithm?

4. Consider the string $X = 1111001101001101110110101\ 001101101000011\ldots$; suppose that the first twenty-five characters (namely, $X_1 \ldots X_{25} = 1111001101001101110110101$) have been compressed and the remainder (i.e., $S_{26} = 001101101000011\ldots$) is uncompressed. What is the next block to be compressed in the LZ'77 algorithm?

Solutions

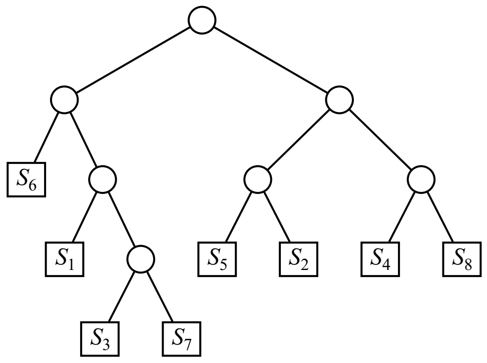
1. The desired trie is



2. The eight strings are

$$\begin{aligned}
S_1 &= 01011001111101001110\ldots\\
S_2 &= 1011001111101001110\ldots\\
S_3 &= 011001111101001110\ldots\\
S_4 &= 11001111101001110\ldots\\
S_5 &= 1001111101001110\ldots\\
S_6 &= 001111101001110\ldots\\
S_7 &= 01111101001110\ldots\\
S_8 &= 1111101001110\ldots
\end{aligned}$$

and the suffix tree associated with these eight strings is



3. We see that $S_{13} = 10010011\ldots$ and $S_5 = 1000101010010011\ldots$ have a prefix of $L = 3$ characters in common (namely, $X_{13}X_{14}X_{15} = X_5X_6X_7 = 100$). So the next block has four characters, namely $X_{13}X_{14}X_{15} = 100$, followed by the next character, which is $X_{16} = 1$.

4. We see that $S_{26} = 001101101000011\ldots$ and $S_{11} = 0011011101101010100110110100011\ldots$ have a prefix of $L = 7$ characters in common (namely, $X_{26}\ldots X_{32} = X_{11}\ldots X_{17} = 0011011$). So the next block has eight characters: namely, $X_{26}\ldots X_{32} = 0011011$, followed by the next character, which is $X_{33} = 0$.