The Effect of Non-Greedy Parsing in Ziv-Lempel Compression Methods

R. Nigel Horspool
Dept. of Computer Science, University of Victoria
P. O. Box 3055, Victoria, B.C., Canada V8W 3P6

E-mail address: nigelh@csr.uvic.ca

Most practical compression methods in the LZ77 and LZ78 families parse their input using a greedy heuristic. However the popular gzip compression program demonstrates that modest but significant gains in compression performance are possible if non-greedy parsing is used. Practical implementations for using non-greedy parsing in LZ77 and LZ78 compression are explored and some experimental measurements are presented.

1 Introduction

All the compression methods in the LZ77 and LZ78 families are based on the principle that strings of symbols tend to recur. By identifying the repeated strings and replacing them with short codes, data compression may be achieved. Methods in the LZ77 family may be broadly characterized as maintaining a buffer, or sliding window, which contains the most recently read N bytes of data. Sequences of new data symbols are encoded as references into the buffer where indentical sequences of symbols may be found. The various LZ77 methods differ, amongst other things, on how the buffer search is implemented and on how to encode symbols that do not appear in the buffer. Methods in the LZ78 family are dictionary-based, where a subset of the symbol sequences encountered in the data so far have been selected for inclusion in the dictionary. Sequences of new data symbols are looked up in the dictionary and, if found, encoded as indexes into the dictionary. The LZ78 methods differ according to how the dictionary is dynamically updated and how to encode dictionary indexes. A survey of the major LZ77 and LZ78 variants appears in [1]. The original descriptions of the two methods are [10] for LZ77 and [11] for LZ78.

With either family of methods, the input is decomposed into a series of substrings where each substring is encoded separately. There is much freedom possible, perhaps too much freedom, in how the input should be decomposed into strings. As a major simplification, and because it is an easily implemented approach that achieves excellent results, a *greedy parsing* approach is commonly used [2]. For the LZ77 class of methods, this means that the compression program always attempts to match the longest possible sequence of symbols, starting at the first un-encoded input symbol, against the contents of the buffer. The program ignores the possibility of matching some shorter sequence so that a longer

subsequent string will be matched or that a better encoding may be achieved. Similarly for LZ78 methods, using a shorter string in the dictionary to encode the first few input symbols may open the possibility of obtaining better encodings for the following input.

On the whole, the greedy heuristic works very well. Some authors have suggested that it is the only realistic approach for practical text compression applications [4] [6]. However, it appears that this conclusion is the result of a binary choice between greedy parsing and optimal parsing. We should still consider the possibility that there may be heuristic non-greedy parsing strategies which achieve sub-optimal, but good, results and are easy to implement. Indeed, the widely used `gzip` program [3], a compression method in the LZ77 family and distributed by the Free Software Foundation, uses a simple non-greedy strategy. It was this observation that motivated the investigations reported in this paper.

First, we consider how we might modify the parsing the LZW/LZC compression method, the most popular LZ78 variant. Then we look at modifying the LZSS compression scheme, which appears to be the most popular method in the LZ77 family.

2 A Non-Greedy Parsing Version of LZW/LZC

LZW was first presented as a compression method suitable for implementation in hardware [8]. Later, a software version was widely distributed as the *compress* program. Because the compress program contains some extra features not described in the original LZW paper, compress is sometimes referred to as the LZC algorithm. More recently, a modified version of LZW has become part of the V.42bis standard for modems.

A Brief Description of LZW/LZC

LZW is an adaptive technique. As the compression algorithm runs, a changing dictionary of (some of) the strings that have appeared in the text so far is maintained. Because the dictionary is pre-loaded with the 256 different codes that may appear in a byte, it is guaranteed that the entire input source may be converted into a series of dictionary indexes. If α and β are two strings that are held in the dictionary, the character sequence $\alpha\beta$ is converted into the index of α followed by the index of β. A greedy string matching algorithm is used for scanning the input, so if the first character of β is x, then αx cannot be an element of the dictionary. The adaptive nature of the algorithm is due to that fact that αx is automatically added to the dictionary if α is matched but αx is not matched. The algorithm maintains the prefix property – namely, if αx is a string in the dictionary, then α must be held in the dictionary also. This enables an entry for an n character string in the dictionary (n>1) to be encoded as a pair `<s,x>` where s is the number of the n–1 character prefix and x is the final character. The LZC implementation uses hashing to store and to look up the `<s,x>` pairs efficiently.

When the dictionary has filled up, LZW becomes non-adaptive – it compresses using an unchanging dictionary. LZC, on the other hand, monitors the compression ratio after the dictionary has filled. If the ratio starts to worsen, the dictionary is re-initialized,

thus restarting the compression algorithm. A special code (an otherwise unused string number) is output so that the decoding algorithm will re-initialize its dictionary at the same point. This enhancement is highly effective on heterogenous files (composed of sections with different characteristics) such as executable files.

In the remainder of this paper, the abbreviation LZW will be used to cover both Welch's version of LZ78 and the LZC variant.

Adding Non-Greedy Parsing to LZW

The simple example shown in Figure 1 illustrates the parsing possibilities available to LZW when it is about to read a new series of symbols from the input.

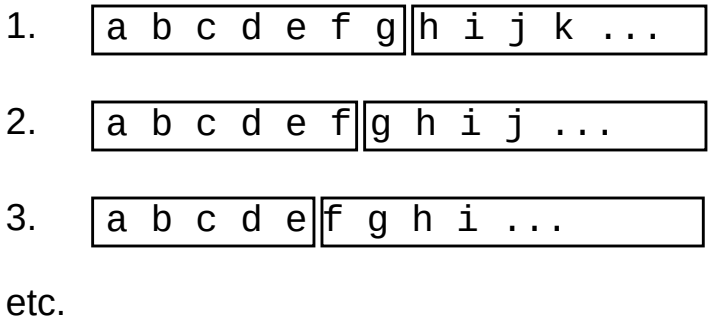

Figure 1 Example of Non-Greedy Parsing for LZW

If the longest matching string is "abcdefg", then LZW would normally output the index of that string, update the dictionary, and then proceed to find another longest matching string (shown here as a string beginning "hijk..."). However, a non-greedy version of LZW could decide to match a shorter string, as shown for example in line 2. If a following string (a string beginning "ghij..." in Figure 1) that is significantly longer than the second string in line 1 can then be matched, the overall compression performance will be improved. Even shorter initial strings may be matched, as in lines 3 and after, and could achieve even better compression performance when subsequent matching strings are considered.

An optimal parsing scheme would also have to consider the possibility of matching a short first string and then a short second string in order to match a very long third string, and so on. We will however reject such possibilities as being too expensive to implement (for minimal expected gain in compression performance). Our non-greedy version of LZW will only look ahead by one parsed string when choosing its course of action.

The adaptive nature of LZW causes a problem with non-greedy parsing. The usual method of updating the dictionary is to create a new string by appending the first symbol of the next string to the current string. When greedy parsing is used, this new string is guaranteed not to be already present in the dictionary. With non-greedy parsing, the guarantee is lost. Just examine line 2 in Figure 1: if we append 'g' to the string 'abcdef', we create the string that is matched in line 1 and is hence a member of the dictionary. So, should we add a new string to the dictionary in this case? And, if so, which string should

be added? Another consideration is that the decompression program must be able deduce which strings to add to its own dictionary and when.

At least two solutions to the dictionary update problem present themselves. Perhaps the easiest solution is to skip the dictionary update. Whenever the compression program uses a non-greedy match, no string is added to the dictionary. Unfortunately, as the experimental results will show, the easy solution does not yield acceptable compression performance and we cannot recommend it for a useful implementation. Our second solution is to update the dictionary as though the greedy match were being used. For example, even if the compression program decides to use the match of line 3 in Figure 1, it will still add the new string 'abcdefgh' to the dictionary – the same string that would have been added if the greedy matching of line 1 had been used. Using this second solution, the overall algorithm for a non-greedy version of LZW is as shown in Figure 2. In this figure, the notation α++b represents the new string formed by appending one character b to the string α, the function length(α) returns the length of the string α, the function head(α) returns the first symbol of a string α, and the function prefix(α,i) returns a substring composed of the first i symbols of the string α. The value K that appears in the code is a parameter that limits the number of non-greedy parsing possibilities considered at each step

```
initialize dictionary D with all strings of length 1;
set α = the string in D that matches the first
   symbol of the input;
set L = length(α);
while more than L symbols of input remain do
begin
   for j := 0 to max(L-1,K) do
      find β_j, the longest string in D that matches
         the input starting L-j symbols ahead;
   add the new string α++head(β_0) to D;
   set j = value of j in range 0 to max(L-1,K)
      such that L - j + length(β_j) is a maximum;
   output the index in D of the string prefix(α,j);
   advance j symbols through the input;
   set α = β_j;
   set L = length(α);
end;
output the index in D of string α;
```

Figure 2 The Non-Greedy LZW Compression Algorithm

The decompression program for LZW has a minor anomaly, named the KωKωK problem. This anomaly occurs if the decompression program reads a dictionary index for a string that has not yet been added to the dictionary. However, Welch argues that there is only one circumstance when this can arise, namely when the compression program has

used a new string immediately after adding that string to the dictionary. If the string previously matched has the form Kω where K represents any single symbol and ω represents a series of zero or more symbols, then the new string must have the form KωK. The decompression program has to check for this anomaly occurring and, when it does, construct and output the KωK string. A similar anomaly is possible for our non-greedy version of LZW. The decompression algorithm shown below in Figure 3 takes it into account. In the figure, the function `tail(α)` returns the string formed by dropping the first symbol of α. The nested **while** loop constructs the new string that the normal (greedy) version of LZW would add to its dictionary at this step. We leave it as an exercise to the reader to verify that (1) the correct string is constructed, and (2) that the **while** loop must always terminate before the `head` function is applied to an empty string.

```
initialize dictionary D with all strings of length 1;
set A = dictionary index read from input;
forever do
begin
   set α = string in D with index A;
   output α;
   if no more input remains then exit;
   set B = next dictionary index read from input;
   if B is a valid index for D then
      set β = string in D with index B;
   else        /* handle the KωKωK anomaly */
      set β = string in D with index A;
   while α++head(β) is a string in D do
   begin
      set α = α++head(β);
      set β = tail(β);
   end;
   add the new string α++head(β) to D;
   set A = B;
end
```

Figure 3 The Non-Greedy LZW De-Compression Algorithm

Experimental Results with Non-Greedy LZW

The algorithms outlined in Figures 2 and 3 were implemented as C programs. Two of the larger Unix command descriptions in the on-line manual were used as samples of typical text input. A data file containing digitized ECG readings and converted to ASCII characters was used as a third test input. This data file is highly compressible because of the small set of ASCII characters used in the file.

The compression results are displayed in Table 1 below. All compression figures are reported as the ratio between the sizes of the compressed file and the original file. The parameter K represents the number of non-greedy parsing choices being considered at each compression step (in addition to the normal LZW greedy choice). As may be expected, the degree of compression improves as K is increased. When K=0, the conventional LZW compression method is being used. Almost all the compression improvement has been achieved when K has reached 3. It is interesting to observe how much benefit can be attributed to each choice in the non-greedy parse. Table 2, below, shows the average benefits that are observed for the compression run using the 'csh' input file with K=8. The entry for the row with j=3, for example, tells us that the compression program chose to match a string three bytes shorter than the maximum possible on 305 occasions. (That is, there were 305 occasions when the assignment statement "`set j = ...`" in Figure 2 assigned three to `j`.) When averaged over these 305 occasions, the algorithm would have been able to match 6.25 more bytes for the next string match (if greedy parsing were actually used for that next match) than for the subsequent string match in the normal greedy parsing case. Reducing the 6.25 by three because of the shorter first match yields the net gain of 3.25 shown in the table.

Table 1: Compression Performance of Non-Greedy LZW

K parameter	0	1	2	3	4	5	6	7	8
'csh' description	41.0%	39.0%	38.1%	37.7%	37.6%	37.5%	37.5%	37.4%	37.4%
'make' description	42.7%	40.5%	39.5%	39.2%	39.1%	39.1%	39.0%	39.0%	39.0%
ECG data	9.7%	8.8%	8.3%	7.8%	7.7%	7.6%	7.5%	7.4%	7.4%

Table 2: Benefits from Non-Greedy Parsing with LZW

Backup value `j`	0	1	2	3	4	5	6	7	8
No. of occurrences	13392	2478	855	305	115	65	31	18	6
Average gain (bytes)	—	3.02	3.12	3.25	3.32	4.65	4.68	5.67	4.00

As mentioned earlier, there is an alternative and simpler strategy for updating the dictionary. Instead of performing the additional work of determining which string would have been added if the usual greedy implementation of LZW were used and adding that string, we could suppress the addition whenever the match is not a greedy match. However, that strategy turns out to produce significantly worse compression performance. Here are a few examples to be compared against the figures in Table 1. The compression for K=3 with the 'csh' example turns out to be 40.9%; for the 'make' example it is 42.0%; for the ECG data, it is 17.4%. Clearly, the dictionary does not grow sufficiently fast to achieve good compression. Perhaps a strategy of accelerating the loading of new strings into the dictionary, such as the 'All-Prefixes' heuristic [7] or similar [5], would help.

3 A Non-Greedy Parsing Version of LZSS

The LZ77 methods encodes an input string by finding an occurrence of that string in its history buffer and outputs a position/length pair that identifies the occurrence. In the original formulation of LZ77, Ziv and Lempel proposed that the position/length pair should be immediately followed in the output by a copy of the next input symbol. This guarantees that input symbols can be encoded even when there are no occurrences of the same symbol in the buffer. The compression performance can be significantly improved by allowing the output stream to consist of a free mixture of position/length pairs and unencoded (literal) symbols. It is conventional for each item in the output stream to be prefixed by a single bit that identifies which kind of item follows. This variation of LZ77 has been named LZSS [1]. It is the basis of much compression software in current use, including stacker and gzip.

If the tagged items in the compressed (output) stream are all encoded using identical numbers of bits, then greedy parsing is provably optimal. For any optimal parse of the input that is non-greedy, we can construct an equivalent greedy parse that generates the same number of output items. However, literal items are normally encoded using fewer bits than position/length pairs. The number of bits used for a pair may also be variable because the position and length components may be coded with variable-length schemes. (Variable-length coding is effective because positions near the end of the buffer are more likely to be referenced than positions further back, and short lengths occur more frequently than longer lengths.)

The gzip program uses non-greedy parsing. Quoting from the program documentation [3]:

> "zip also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, zip searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the longer match is emitted afterwards. Otherwise, the original match is kept, and the next match search is attempted only N steps later."

The gzip approach achieves better compression because a single literal byte has a short encoding. Indeed, any non-greedy version of the LZSS must necessarily take advantage of the different coding lengths of items. Our non-greedy version of LZSS may be seen as a generalization of the gzip approach. It is easy to implement in that only a handful of possibilities need to be carried along at any step. The algorithm has the structure shown in Figure 2, below. All the details of how to find a longest match between the input and the buffer and the transferral of input bytes into the buffer have been omitted from the algorithm. The algorithm contains a parameter, `MAXDELTA`, which limits the number of non-greedy possibilities that are tested at each step.

The algorithm generalizes the gzip approach by testing to see how many symbols would be matched if we skipped forward by 2 positions, by 3 positions, and so on. If we

discover that skipping forward by, say, 3 symbols would yield a benefit then the initial match can be shortened to a match of just three symbols. MAXDELTA limits how far ahead the algorithm considers skipping ahead with a short match.

If it is sometimes profitable to skip ahead by one byte, generating a single literal byte, then it is likely that skipping ahead two bytes may also be profitable if those two bytes can be represented compactly. The encoding scheme used in our experiments uses 9 bits for a literal byte and usually uses 10 bits to encode a pair where the length is 2, 3 or 4. (It sometimes uses 14 bits, depending on the position in the buffer.) The cost of a single extra bit definitely seems worthwhile if a match that extends even one byte further into the input can thereby be used.

The decoding algorithm for this compression approach is the same as for the original (greedy) version of LZSS. It is therefore not shown here.

```
while input remains to be compressed do begin
   set len[0],pos[0] = length and position of longest
      match between input stream and buffer contents;
   if len[0] < 2 then
      set delta = 1;
   else begin
      set delta = 0;
      for j := 1 to max(len[0]-1, MAXDELTA) do begin
         set len[j],pos[j] = length and position of
            longest match between input starting j
            bytes ahead and contents of buffer;
         if len[j] > len[delta]+j then
            set delta = j;
      end
   end
   if delta = 0 then begin
      output length/position pair <len[0],pos[0]>;
      advance len[0] symbols through input;
   end else if delta = 1 then begin
      output next input byte as a literal;
      advance one symbol through input;
   end else begin
      output length/position pair <delta,pos[0]>;
      advance delta symbols through input;
   end
end
```

Figure 4 The Non-Greedy LZ77 Compression Algorithm

Experimental Results with Non-Greedy LZSS

The compression results for the algorithm shown in Figure 2 are summarized in Table 3. The notation ‘Δ=n’ shows the setting used for `MAXDELTA` in each test. For comparison purposes, the compression achieved with the basic LZSS scheme and LZSS with the ‘lazy evaluation’ heuristic are also included in the table. The experimental algorithm was implemented using the same published table of encodings for literal bytes and pairs [9] as is used in the Stacker product. The length characteristics of the coding scheme are shown in Table 4 (which assumes that positions are relative to the end of the buffer).

As may be expected, the compression steadily improves with increasing values for `MAXDELTA`. However, at 4, almost all the benefit has been achieved.

Table 3: Benefits from Non-Greedy Parsing with LZSS

	Basic LZSS	LZSS/ gzip	Non-Greedy Algorithm					
			Δ=2	Δ=3	Δ=4	Δ=5	Δ=6	Δ=7
‘csh’ description	35.0%	34.0%	33.9%	33.6%	33.4%	33.3%	33.3%	33.3%
‘make’ description	33.8%	33.0%	32.8%	32.4%	32.3%	32.2%	32.1%	32.1%
ECG data	14.2%	14.1%	13.9%	12.9%	12.5%	12.4%	12.3%	12.3%

Table 4: LZSS (Stacker) Coding Characteristics

Literal Item:	1 bit for tag + 8 bits for symbol
Pair:	1 bit for tag + Position + Length
Position component:	7 bits if 1 <=position < 128 11 bits if 128 <=position
Length component:	2 bits if 2 <=length < 4 4 bits if 4 <=length < 8 8 bits if 8 <=length < 24 12 bits if 24 <=length < 40 ... etc.

4 Summary and Conclusions

The non-greedy variations on LZW and LZSS that are proposed in this paper offer a simple way to trade CPU time for better compression. In the case of LZSS, the de-compression program does not require any modifications at all. The de-compression program for LZW, however, does need to be extended in a minor and inexpensive way. (It only consumes extra CPU time at each use of a non-greedy match.) This is an upward compatible extension in the sense that the same program could still be used to de-compress the output of the standard (greedy) LZW algorithm. The two non-greedy methods are well-suited for

situations where maximal compression is desirable and where compression speed is less important.

It is interesting to observe that these algorithmic extensions to LZW and LZSS do not require us to modify the coding of items in the compressed file. This is a symptom of the redundancy that remains in a file compressed with LZW or LZSS. In the case of LZW, many combinations of adjacent dictionary indexes cannot occur in the output (those where the string represented by the first index and extended by the first symbol of the second string are already contained in the dictionary). Similarly, in the case of LZSS, many combinations of adjacent length/position pairs or literal bytes cannot normally occur in the compressed file. If some other method could be used to remove the redundancy, the advantage of non-greedy parsing would be largely lost.

The numbers tend to support the view that the greedy methods give results fairly close to those achievable with the more general non-greedy parsing. However, if modest improvements in compression are available for modest effort and with little extra complexity in the algorithm, why not go after them?

References

[1] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ (1990).

[2] Bell, T. C., and Witten, I. H. The Relationship between Greedy Parsing and Symbolwise Text Compression. Journal of ACM 41, 4 (July 1994), pp. 708-724.

[3] Gailly, J.-L. Documentation for `gzip` program, version 1.2.4, (Aug. 1993).

[4] Gonzalez-Smith, M. E., and Storer, J. A. Parallel Algorithms for Data Compression. Journal of ACM 32, 2 (April 1985), pp. 344-373.

[5] Horspool, R.N Improving LZW. Proceedings of Data Compression Conference, DCC'91, IEEE Computer Society Press, (April 1991), pp. 332-341.

[6] Schuegraf, E. J., and Heaps, H. S. A Comparison of Algorithms for Database Compression by use of Fragments as Language Elements. Inf. Stor. Ret. 10 (1974), pp. 309-319.

[7] Storer, J.A. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, MD (1988).

[8] Welch, T. A. A Technique for High-Performance Data Compression. IEEE Computer 17,6 (June 1984), pp. 8-19.

[9] Whiting, D. L., and George, G. A. Data Compression Apparatus and Method. U.S. Patent 5,016,009 (May 1991).

[10] Ziv, J, and Lempel, A. A Universal Algorithm for Sequential Data Compression. IEEE Trans. on Inf. Theory IT-23,3 (May 1977), pp. 337-343.

[11] Ziv, J, and Lempel, A. Compression of Individual Sequences via Variable-Rate Coding. IEEE Trans. on Inf. Theory IT-24,5 (Sept. 1978), pp. 530-536.