A Linear Algorithm for Data Compression

R.P. Brent[†]

We describe an efficient algorithm for data compression. The algorithm finds maximal common substrings in the input data using a simple hashing scheme, and repeated substrings are encoded using Huffman coding. Time and space requirements are proportional to the size of the input data. A modification which uses a bounded input buffer is also described. The algorithm is simpler than comparable linear-time algorithms and gives excellent compression ratios on both text and binary files.

Keywords and Phrases: **Data compaction and compression, pattern matching, hashing, Huffman coding, maximal common substrings, move-to-front strategy, Ziv-Lempel algorithm.**
CR Categories: **E.4, F.2.2.**

1. Introduction

Reversible compression of files has many applications. When combined with suitable error detection and correction schemes, it may be used to speed up transmission of data over telephone lines. It may also be used to reduce the cost and physical volume of archival storage. For example, the work described in this paper was motivated by the need to back up a 10 Mbyte hard disk using 360 Kbyte floppy disks on a personal computer. The algorithm SLH described below reduced the number of floppy disks required from about thirty to less than ten.

For compression of text files, a natural scheme is to split the input into 'words' which are then encoded in some way, eg. via static Huffman coding (Huffman, 1952; Knuth, 1968; Moffat, 1987), dynamic Huffman coding (Faller, 1973; Gallager, 1978; Knuth, 1985; Vitter, 1985), or the 'move to front' algorithm of Bentley *et al.* (1986). Such schemes perform well on English text or source programs written in high-level languges, but they perform badly on binary files (eg. compiled programs) or numerical data because of the difficulty of suitably defining 'words'. For the application mentioned above we needed an algorithm which would be effective on both text and binary files.

A popular algorithm for data compression is the Ziv-Lempel algorithm or one of its variants (Ziv and Lempel, 1977, 1978; Welch, 1984; Langdon, 1983). These use a translation table which maps strings of input bytes into non-negative integers; these are usually encoded using a fixed-length (eg. 12-bit) code, but variable-length codes may also be used (Welch, 1984). The Ziv-Lempel algorithms are simple, suitable for hardware implementation, and have some nice asymptotic optimality properties. However, they also have some disadvantages.

1. They are slow to start, as the strings in the translation table are built up one byte at a time.
2. They are not adaptive, so may perform badly if the statistical properties of the input change after the translation table has been built up.
3. When used with a finite translation table, there is a problem when the table becomes full; usually the algorithm is restarted at this point, so information acquired about statistical properties of the input is thrown away. (For other possibilities, see Tischer, 1987.)

In this paper we describe an algorithm (SLH) which avoids these disadvantages, and appears to perform better than the Ziv-Lempel algorithms, although at the expense of using more CPU time for compression and requiring two passes rather than one. The algorithm is outlined in Section 2, and details are given in Sections 3 and 6. The running time of the algorithm is considered in Section 4, and a practical modification (the use of a bounded input buffer) is discussed in Section 5. Experimental results and comparisons with some of the other algorithms mentioned above are given in Section 7.

2. Outline of the algorithm SLH

In this section we outline the main ideas of the data compression algorithm SLH. The input data to be compressed is regarded as a string $s_1s_2 \ldots s_n$ over some alphabet Σ. In principle Σ could be the set of 16-bit words, 7-bit Ascii characters, printable Ascii characters, binary digits, etc. However, we shall always assume that Σ is the set of all 8-bit bytes.

The algorithm has two passes. In the first pass repeated substrings in the input data are found, and the second and subsequent occurrences of each string are encoded as ordered pairs (δ, m), where δ is a displacement (the number of bytes between two occurrences of the string) and m is the length of the string. The second pass encodes the ordered pairs efficiently using Huffman coding. The second pass could be avoided, at the expense of some

† *Computer Sciences Laboratory, The Australian National University, Canberra, ACT 2601.* *This paper was presented at the tenth Australian Computer Science Conference at Deakin University in Geelong, Victoria in February 1987. Manuscript received March 1987.*

reduction in the compression ratio achieved, by using adaptive Huffman coding or some other on-line encoding of the ordered pairs. The first pass is the more interesting one and we defer discussion of the second pass until Section 6.

Suppose that $s_1 \ldots s_j$ has already been encoded. The idea is to find a maximal substring of $s_1 \ldots s_j$ matching $s_{j+1}s_{j+2} \ldots$, ie. find k, $1 \leqq k \leqq j$, such that $s_k \ldots s_{k+m-1} = s_{j+1} \ldots s_{j+m}$ and m is maximal (subject to the constraint that $j + m \leqq n$). If $m \geqq 3$ then $s_{j+1} \ldots s_{j+m}$ is encoded as the ordered pair (δ, m), where $\delta\ j - k$. We may also choose such an encoding if $m = 2$ (this is a borderline case). If $m = 1$ the first pass simply outputs s_{j+1} and increments j. Thus the output of the first pass is a string of bytes interspersed with ordered pairs. Note that overlapping strings are permitted, ie. $k + m - 1 > j$ is possible. However, we do insist that $\delta \geqq 0$. For more exotic possibilities, see the survey by Storer and Szymanski (1982). Expansion of the output of the first pass to retrieve the original input string is extremely simple and fast.

As an example, the first pass would encode the input string *aaaabaaacaaba* as *a*(0,3)*b*(3,3)*c*(6,4).

In practice we have to operate in bounded space, so we keep a circular input buffer of length *B* say and restrict $\delta \leqq B$. It is also convenient to practice to impose a moderate upper bound on *m*. Details of these modifications are given in Section 5.

The maximal matching substring problem is well known and can be solved by constructing a 'position tree' or 'prefix tree' corresponding to the input string (Aho, Hopcroft and Ullman, 1974). Unfortunately, the position tree can have $\Omega(n^2)$ vertices, so any algorithm which depends on its construction can not run in time $O(n)$, at least in the worst case. Weiner (1973) showed that the position tree could be represented in space $O(n)$ and that the maximal matching substring problem could be solved in linear time. However, Weiner's algorithm is rather complicated and, like other theoretically good algorithms (McCreight, 1976; Rodeh *et al.*, 1981), does not appear to have been used much in practice. Our algorithm SLH avoids explicit construction of the position tree and is quite easy to implement, so we hope that it will be useful in practice.

Recently Bell (1986) has proposed the use of a binary search tree to solve the maximal matching problem. If implemented with a balanced binary tree, Bell's algorithm would have running time $O(n \log n)$. In Section 4 we show that alogorithm SLH has running time $O(n)$.

```
m ← 2; m' ← 0;
repeat
  look up $s_{j+1} \cdots s_{j+m}$ in H,
  if found then match ← true
    else match ← false;
  if match then
    begin
    represent the matching entry
          in H by (k, m);
    { Save the best match found so far }
    k' ← k; m' ← m;
    replace the representation of the
          matching entry in H by (j+1, m);
    if k + m ≤ n then
      enter $s_k \cdots s_{k+m}$ in H
        (overwriting any matching entry);
    m ← m + 1
    end
  else
    enter $s_{j+1} \cdots s_{j+m}$ in H
until (not match) or (j + m > n)
          {i.e. all input processed }
if m' ≥ 2 then
  begin
  output (j − k', m');
  if j + m' < n then
    { enter substrings of maximal match in H }
    for j' ← j + m' downto j + 2 do
      begin
      look up $s_{j'} \cdots s_{j+m'+1}$ in H
      if found then
        { representation is (k", m") say }
        replace the representation by (j', m")
      else
        enter $s_{j'} \cdots s_{j+m'+1}$ in H
      end;
  j ← j + m'
  end
else
  begin
  output $s_{j+1}$;
  j ← j + 1
end.
```

Figure 1.

3. A linear algorithm for maximal matching

In this section we describe the first pass of algorithm SLH. We assume that the input string $s_1 \ldots s_n$ is stored in a buffer and that $s_1 \ldots s_j$ has already been encoded. We use a hash table *H* with keys of the form $s_k \ldots s_{k+m-1}$ which are represented in constant space by ordered pairs (k, m). Initially *H* is empty and $j = 0$.

To find a maximal substring matching $s_{j+1}s_{j+2} \ldots$ we use the algorithm in Figure 1.

For the first pass, the algorithm is repeated until $j \geqq n$. The reader may find it instructive to verify the example given in Section 2.

We shall not give a formal correctness proof of the fact that the algorithm above finds a maximal matching substring $s_k \ldots s_{k+m-1} = s_{j+1} \ldots s_{j+m}$ (subject to $k \leqq j, j + m \leqq n$). The idea of such a proof is to show, by induction on k and m, that $s_k \ldots s_{k+m'-1}$ is in *H* for some $m' \leqq m$. Note that some of the steps in the algorithm are unnecessary, but are included to minimise $\delta = j - k'$ and to facilitate the use of a bounded buffer (for which see Section 5).

4. Linearity of the maximal matching algorithm

In this section we show how the maximal matching algorithm of Section 3 can be implemented so that the time required by the first pass of algorithm SLH is $O(n)$, ie. the algorithm runs in time linear in the length of the input. We assume that the hash table H is implemented so that the time required to look up or insert a key K, given its hash function $h(K)$, is constant. This is not strictly correct in the worst case, but for practical purposes the assumption is justified.

Since the keys K occurring in the algorithm are strings $s_{j+1} \ldots s_{j+m}$, it is not immediately obvious that the hash function $h(K)$ can be computed in constant time. Inspection of the first half of the algorithm shows that we need to compute $h(s_{j+1}s_{j+2}), h(s_{j+1}s_{j+2}s_{j+3}), \ldots, h(s_{j+1} \ldots s_{j+m})$. This can be done in time $O(m)$ provided that h has the form

$$h(s_{j+1} \ldots s_{j+m}) = \sum_{i=1}^{m} \beta^{m-i} \sigma(s_{j+i}) \bmod p \qquad (4.1)$$

for some fixed β and p, where $\sigma(s)$ is just the ordinal value of s (or some other easily computed function of s). For example, we may take $\beta = 256$ and p the hash table size (preferably odd). Once $h(K)$ is known, we can compute $h(Ks)$ from

$$h(Ks) = \beta h(K) + \sigma(s) \bmod p \qquad (4.2)$$

In the second half of the algorithm ('**if** $m' \geqq 2$ **then** . . .') we need to compute $h(s_{j+m'}s_{j+m'+1}), \ldots, h(s_{j+2} \ldots s_{j+m'+1})$. This can be done in time $O(m')$ using the relations

$$h(sK) = \beta^{|K|}\sigma(s) + h(K) \bmod p \qquad (4.3)$$

and

$$\beta^{|sK|} \bmod p = \beta(\beta^{|K|} \bmod p) \bmod p \qquad (4.4)$$

provided that h has the form (4.1). Thus there is no difficulty in computing the required hash functions in constant time.

It is easy to prove by induction on j that when $s_1 \ldots s_j$ has been encoded H contains at most $2j$ entries, and if $s_{j+1} \ldots s_{j+m}$ is the next string encoded then the time required for this is $O(m)$. Thus, the total time (and space) required to encode $s_1 \ldots s_n$ is $O(n)$.

In the analysis above we implicitly assumed that two keys K and K' could be compared in constant time. With care we can ensure that this is true because it is often known that only the first or last bytes of K and K' can disagree. In practice it is sufficient just to check that $h(K) = h(K')$, $|K| = |K'|$, and that the first few bytes of K and K' agree; the probability of a 'false match' is small and we can check for it and backtrack if necessary before encoding $(j - k', m')$.

5. Use of a bounded buffer

The algorithm described in Section 3 is impractical as the whole input string $s_1 \ldots s_n$ must be stored in random-access memory. However, we may use a circular buffer of fixed size B say, and only store the last B bytes which have been processed. In practice it is also convenient to impose an upper limit M on the length m of a matching substring. Thus, when encoding $s_{j+1} \ldots$ we need only store $s_{\max(1,j-B)} \ldots s_j s_{j+1} \ldots s_{j+M}$ in random-access memory, so the space required is $O(B + M)$. The algorithm described in Section 3 can easily be modified to take these constraints into account. The only difficulty is how to delete hash table entries which are no longer relevant because they no longer correspond to strings contained in the buffer. A simple and efficient solution is not to delete entries (k, m) from H as soon as $k < j - B$, but to periodically perform 'garbage collections' to remove such entries. Of course, when looking up a key in H, we must not return a match (k, m) if $k < j - B$. In practice garbage collections are seldom necessary if garbage is removed whenever it is encountered during hash table operations.

When expanding (i.e. reversing the compression process) a circular buffer of size B is sufficient if compression was performed with a buffer of this size, because all ordered pairs (δ, m) encountered must have $\delta \leqq B$.

6. Encoding ordered pairs

The second pass of algorithm SLH has to encode a string of bytes s and ordered pairs (δ, m) produced by the first pass. There are many ways to do this. Our implementation assumes that $M < 256$ and (because of memory constraints) $B < 2^{18}$, where M and B are upper bounds on m and δ (see Section 5). We map 8-bit bytes s into 9-bit numbers by the identity mapping, and ordered pairs (δ, m) into triples of 9-bit numbers $(256 + m, \delta_1, \delta_2)$. (The ordering is important so that single bytes and triples can easily be distinguished when the process is reversed.) The 9-bit numbers are then encoded using Huffman's algorithm (Knuth, 1968). To avoid an extra pass, the frequencies required for Huffman coding are accumulated during the first pass of algorithm SLH. Before the second pass these frequencies are normalized to the range 0 . . . 255 (with zero used only for symbols with genuine zero frequency). This allows us to encode the Huffman tree indirectly, via the table of normalized frequencies, in at most 512 8-bit bytes. If the two-pass nature of the algorithm were considered undesirable, then dynamic Huffman coding or arithmetic coding schemes (Cleary and Witten, 1984; Pasco, 1976) could be used in place of static Huffman coding.

Various refinements are possible, eg. pairs (δ, m) with $m = 2$ or 3 and small δ may be encoded specially, and the mapping $\delta \rightarrow (\delta_1, \delta_2)$ may be chosen to optimize performance of the subsequent Huffman coding, but space limitations prevent us from going into details here.

7. Experimental results

The data compression algorithm SLH described above was implemented on an IBM PC-compatible microcomputer (actually a COMPAQ using an 8 MHz Intel 80286 CPU chip) in Turbo Pascal. Results below are given for a circular buffer size B as large as possible, given memory constraints (in fact $B \doteq 17000$). The simple example of a repeated cycle of length B shows that a buffer of size B may be much better than one of size less than B. However, in practice it was found that the compression ratio for a buffer of size 4000 was usually within a few per cent of the compression ratio for larger buffer sizes.

For purposes of comparison we implemented the Ziv-Lempel algorithm (Welch, 1984) with output symbol size 9, 10, . . . , 15 bits. This is as in the 'compress' utility available on some Unix systems (though without the 'block compression' feature). We also implemented the 'move to front' (MTF) algorithm of Bentley *et al.* (1986) with a list size of 128 words and Huffman coding of the output. To implement MTF we used a 'splay tree' (Sleator and Tarjan, 1985). Our definition of 'word' was slightly different to that of Bentley *et al.* (1986) so we did not alternate between alpha-numeric and non-alphanumeric words. We also implemented straightforward Huffman coding (HUF) of the input string, regarded as a string of bytes.

In Table 1 we give the results obtained when the different algorithms were applied to a Pascal program of size 20873 bytes, with trailing blanks already removed. In the table, 'compression ratio' is the ratio of the input data size to the output size (including encoding of the Huffman tree when applicable). The time estimates should only be taken as a rough guide, since no particular effort was made to optimize the running times of the programs.

Table 1. Comparison of data compression algorithms.

Algorithm	Compression time (seconds)	Expansion time (seconds)	Compression ratio
HUF	16	12	1.75
MTF	44	42	2.29
Ziv-Lempel	26	20	2.48
SLH	44	12	3.18

The results given in Table 1 are typical of those obtained for text files, including some much larger than the buffer size. On binary files the compression ratios attained were more variable than for text files and generally smaller. For example, on one executable file of size 36800 bytes, the algorithms HUF, MTF, Ziv-Lempel and SLH gave compression ratios of 1.17, 0.99, 1.21 and 1.56 respectively.

8. Conclusion

The compression ratios achieved by algorithm SLH compare well with those achieved by the Huffman, MTF and Ziv-Lempel algorithms. The CPU time required for compression by algorithm SLH is comparable to that for MTF, somewhat more than that for Ziv-Lempel, but within a factor of two. Thus SLH should be preferred unless the simplicity and speed of Ziv-Lempel is considered more important than its poorer performance in compressing the input. Although our implementation of MTF might be improved, MTF does not appear to be a serious competitor.

To conclude on a practical point: the operating system (MS-DOS) under which our programs were developed has a minimum file size of 4096 bytes on a 10 Mbyte disk (presumably so that the starting location of a file can be identified with 12 bits). Thus the compression ratios attainable by simply concatenating small files (with appropriate headers containing their names, etc.) can easily exceed those attainable by more sophisticated data compression algorithms. Our implementation of algorithm SLH includes a facility to concatenate files before they are compressed.

9. Acknowledgement

We thank Tektronix Australia for providing the COMPAQ Portable and Alistair Moffat for his comments on a first draft of this paper.

10. References

AHO, A.V., HOPCROFT, J.E. and ULLMAN, J.D. (1974): *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

BELL, T.C. (1986): Better OPM/L text compression, to appear in *IEEE Transactions on Communications*.

BENTLEY, J.L., SLEATOR, D.D., TARJAN, R.E. and WEI, V.K. (1986): A locally adaptive data compression scheme, *Communications of the ACM*, 29, 4, pp. 320-330.

CLEARY, J.G. and WITTEN, I.H. (1984): Data compression using adaptive coding and partial string matching, *IEEE Transactions on Communications*, COM-32, 4, pp. 396-402.

FALLER, N. (1973): An adaptive system for data compression, *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, pp. 593-597.

GALLAGER, R.G. (1978): Variations on a theme by Huffman, *IEEE Transactions on Information Theory*, IT-24, 5, pp. 668-674.

HUFFMAN, D.A. (1952): A method for the construction of minimum redundancy codes, *Proceedings of the IRE*, 40, pp. 1098-1101.

KNUTH, D.E. (1968): *The Art of Computer Programming, Volume 1*, Addison-Wesley.

KNUTH, D.E. (1985): Dynamic Huffman coding, *J. Algorithms*, 6, 2, pp. 163-180.

LANGDON, G.G. (1983): A note on the Ziv-Lempel model for compressing individual sequences, *IEEE Transactions on Information Theory*, IT-29, 2, pp. 284-287.

McCREIGHT, E.M. (1976): A space-economical suffix tree construction algorithm, *Journal of the ACM*, 23, 2, pp. 262-272.

MOFFAT, A. (1987): Predictive text compression based on the future rather than the past, *Australian Computer Science Communications*, 9, 1, pp. 254-261.

PASCO, R. (1976): *Source Coding Algorithms for Fast Data Compression*, PhD dissertation, Stanford University, Stanford, California.

RODEH, M., PRATT, V.R. and EVEN, S. (1981): Linear algorithms for data compression via string matching, *Journal of the ACM*, 28, 1, pp. 16-24.

SLEATOR, D.D. and TARJAN, R.E. (1985): Self-adjusting binary search trees, *Journal of the ACM*, 32, 3, pp. 652-686.

STORER, J.A. and SZYMANSKI, T.G. (1982): Data compression via textual substitution, *Journal of the ACM*, 29, 4, pp. 928-951.

TISCHER, P. (1987): A modified Lempel-Ziv-Welch data compression scheme, *Australian Computer Science Communications*, 9, 1, pp. 262-272.

VITTER, J.S. (1985): Design and analysis of dynamic Huffman coding, *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, New York, pp. 293-302.

WEINER, P. (1973): Linear pattern matching algorithms, *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, pp. 1-11.

WELCH, T.A. (1984): A technique for high-performance data compression, *IEEE Computer*, 17, 6, pp. 8-19.

ZIV, J. and LEMPEL, A. (1977): A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, IT-23, 3, pp. 337-343.

ZIV, J. and LEMPEL, A. (1978): Compression of individual sequences via variable-rate coding, *IEEE Transactions on Information Theory*, IT-24, 5, pp. 530-536.

Biographical Note

Richard Brent was born in Melbourne in 1946. He obtained a PhD in Computer Science from Stanford University in 1971 and a DSc from Monash University in 1981. He is a Fellow of the Australian Academy of Science and was an invited speaker at the 1980 and 1983 IFIP Congresses. In 1978 he was appointed Foundation Professor of Computer Science at the Australian National University, where he is now leader of the Computer Sciences Laboratory in the Research School of Physical Sciences. His research interests include analysis of algorithms and parallel computation.