An Approximation Algorithm for Space-optimal Encoding of a Text

J. KATAJAINEN AND T. RAITA*
Department of Computer Science, University of Turku, SF-20520 Turku, Finland

In many situations text compression is carried out with a previously formed fixed dictionary (code book) expressing those often-occurring substrings of a text which are to be replaced by code words. The problem of encoding a text in a space-optimal manner is equivalent to the problem of finding a shortest path between a given pair of vertices in an acyclic and bandwidth-limited network. By combining an algorithm for finding shortest paths with the string matching algorithm of Aho and Corasick,[1] a time-efficient approximation algorithm for the space-optimal encoding is obtained. The performance of the approximation algorithm depends on the amount of storage space available in the fast memory of a computer. With an unrestricted, though at most linear working storage on the length of the input text, a space-optimal encoding is obtained. However, even a fixed internal memory of moderate size guarantees almost optimal compression, and in spite of this the running time of the algorithm is comparable to that of the longest match heuristic.

Received April 1987

1. INTRODUCTION

The aim of text compression is to reduce the size of a text file in order to save storage space in a secondary storage device or to have a higher throughput in a communication channel: Text compression may also have some cryptographic value, as has been pointed out, for instance, by Rubin.[14]

Most text compression schemes described in the literature are based on textual substitution:[17] frequently occurring substrings of the text are to be replaced by some code words which uniquely identify the replaced data. The code words may be references to a separate dictionary (code book), to the original text, or even to the compressed representation of the text.

A typical text compression system contains the following three parts:

(*a*) an analyser, which constructs a dictionary consisting of strings and the corresponding code words;

(*b*) an encoder, which performs the actual compression by using the dictionary, and

(*c*) a decoder, which regenerates the original text from the encoded representation by substituting dictionary strings for the code words.

Depending on the coding algorithm, the first two parts may either be separate or intertwined with each other.

In this paper we shall study the encoding of a text, assuming that the dictionary is given as an input to the encoder. Alternatively, each text could have a dictionary of its own. This presupposes that there has been a preprocessing phase during which the source text is sampled and the dictionary generated. It is not necessary to deal here with the way in which the dictionary strings are chosen. However, it should be noted that the selection of the best possible dictionary is a difficult task; as shown by Storer[16] and Fraenkel *et al.*,[4] the problem is NP-complete. The inherent difficulty lies in the fact that the substrings of the optimal dictionary may overlap. However, several efficient heuristic methods for constructing dictionaries have been proposed; a survey of these methods has been published in Ref. (3).

We assume that the compression scheme is reversible, that is to say that the source text can be fully recovered from the compressed form. The decoding process itself is usually very simple, basically just a single left-to-right scan of the compressed form, if the code words are properly selected. In what follows, both the variable and the fixed-length code words are accepted.

Different encoding algorithms have been examined by various authors (e.g. Refs. 3, 5, 13, 15, 19). Given a dictionary, space-optimal encoding of a text can be accomplished in polynomial time. Wagner has given a nonlinear integer programming formulation to the problem[19] and a dynamic programming algorithm for finding a shortest compressed form of a given text string.[20]

Schuegraf and Heaps[15] showed that the optimal encoding problem is equivalent to the problem of finding a shortest path between a given pair of vertices in a directed network, and they used a general network algorithm for finding the shortest path. Rubin[13] presented yet another optimal encoding algorithm, which simultaneously maintains several coding possibilities, discarding any alternative as soon as it is found to be non-optimal.

In general, the optimal encoding algorithms have been considered impractical because of their computational cost. Therefore, numerous heuristic algorithms have been developed, e.g. the longest match heuristic, the longest fragment first heuristic,[15] and the greedy heuristic.[5] The longest match algorithm (LM) processes the input text from left to right, choosing at each position the longest dictionary substring which matches the original text, and replaces the substring by a code word. This heuristic has been widely used because it is simple, gives almost the same gain in compression as an optimal algorithm, and also performs faster.[3,15]

In this paper we give an optimal algorithm (OPT), which can be regarded as a refinement of those introduced by Wagner[20] and Rubin.[13] Although the new algorithm gives an optimal coding result, a working storage of linear size on the input text length is needed in extreme cases,

* To whom correspondence should be addressed.

[1] A preliminary version of this article was presented at the Third Finnish Symposium on Theoretical Computer Science, Merkrijärvi, 5–8 Jan. 1987.

Moreover, if the input is extensive, the working storage overflows, resulting in data transfers between the fast and the slow memories. However, the OPT algorithm is easily modified to an approximation algorithm (OPT_{buf}), which uses a bounded memory (buffer) for handling the input text piece by piece. Theoretically, OPT_{buf} guarantees almost optimal compression. In the experiments we have made, it always yielded an optimal coding result and its running time was comparable to that of the LM heuristic.

We use the network formulation for the encoding problem. Because of the special characteristics of the input networks, the shortest-path problem is theoretically interesting. First, the arcs of a network are generated simultaneously with the shortest path computation; they are not given explicitly as an input. Secondly, the vertices of a network can be arranged linearly one after another and each arc can emanate from a vertex only to one of its immediate neighbours on the right. Thirdly, the arc weights are small integers. It is easy to see that the networks are acyclic, which means that the single-source shortest path problem can be solved in linear time, if the model of computation is a random access machine (see e.g. Ref. 18, section 7.2). In our case, however, the problem networks are typically far too big to fit into the fast memory of a computer. Hence we assume a two-level machine model in complexity issues. In other words, the complexity of an algorithm is determined by the number of memory blocks transferred between the internal and the external memories.

The paper is organised as follows. Section 2 recalls the equivalence of the encoding problem and the shortest-path problem. Section 3 presents the space-optimal encoding algorithm, based on a general algorithm for finding a shortest path in acyclic networks. Also the time and space complexities of the algorithm are analysed. In Section 4 the compression gains achieved with the OPT_{buf} algorithm, using different types of dictionaries, are compared with those obtained by the optimal algorithm. The experimental results are summarized in Section 5 and there are some concluding remarks in Section 6.

2. THE PROBLEM

In this section we define the encoding problem and introduce the corresponding network formulation for the problem.

Let us assume that a text string over some finite (input) alphabet and a coding dictionary are given. The dictionary consists of a collection of (substring, code word) – *pairs*. The dictionary is assumed to be complete in the sense that there is at least one way to represent every text string as a concatenation of the code words. Therefore we assume that the dictionary contains all the members of the input alphabet as substrings. The code words are bit strings, not necessarily of equal length. It is assumed that the coding scheme is instantaneous, i.e. that the mapping between a code word and a substring is uniquely defined and independent of the context in which the code word is used. The problem is to determine how the input text should use the dictionary strings in order to minimise the storage requirements.

Example 1. Assume that the characters of the input text are represented in a machine which uses 8-bit bytes. In this situation it is often convenient to use all the available 256 bit strings as fixed-length code words. Each character of the input text is chosen as an element in the dictionary, and the remaining bit combinations are assigned to some frequently occurring substrings. □

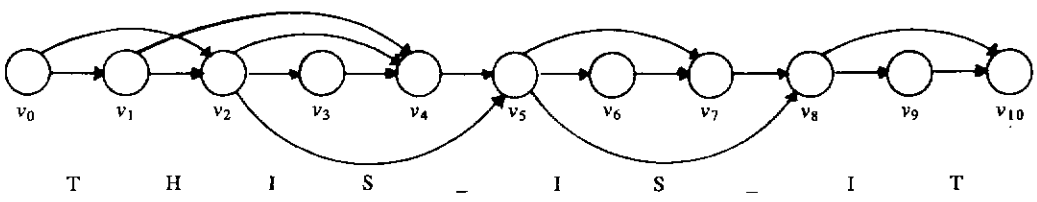
Let n be the length of the original text in characters. The space-optimal encoding can be understood as the finding of the shortest path between the start and the end vertices in a directed and weighted network $G = (V, E)$. V is the set of vertices, $V = \{v_0, v_1, \ldots, v_n\}$, and E the set of arcs containing m elements, each of which corresponds to a possible substitution of a substring by a code word. Hence E contains the directed arc (v_i, v_{i+d}), if there exists a dictionary string of length d ($d > 0$) which matches the original text at positions $i+1, \ldots, i+d$. The weight (or cost) of an arc is the number of bits in the corresponding code word.

Each path from the source v_0 to the sink v_n corresponds to a compressed representation of the source text. From the completeness of the dictionary it follows that at least one such path exists. Furthermore, because every character of the input alphabet is an element in the dictionary, the network contains a base path, i.e. (v_i, v_{i+1}) is an arc of the network for $i = 0, 1, 2, \ldots, n-1$. It is evident that the shortest path (the path of minimal weight) from the source to the sink corresponds to the minimum compressed form of the text string.

Example 2. Consider the string THIS_IS_IT and the dictionary

Substring	T	H	I	S	_	TH	HIS	IS	IS_	IT
Code word	t	h	i	s	-	a	b	c	d	e

The corresponding network is



The shortest path from v_0 to v_{10} is $v_0v_2v_5v_8v_{10}$, if the code words are of equal length. The compressed form of the string is added. □

It is clear that the encoding networks are acyclic and their vertices are initially in topological order, i.e. if (v_i, v_j) is an arc of G, then $i < j$. The difference $j-i$, which is bounded above by the length of the longest substring of the dictionary, is usually small. A weighted network with these properties is called an acyclic bandwidth-limited network[9] because the band of non-zero values around the diagonal of the conventional distance matrix defines the network totally.

A vertex v is a cut vertex, if $G-v$ is not a connected network. Also the sink and the source are cut vertices. In our application this definition may also be stated as follows: a vertex v_i is a cut vertex, if there does not exist an arc (v_j, v_k), such that $j < i < k$ and there is a path from the source to v_j and v_i. This means that all paths from the source to the sink must go through all cut vertices. For example, the vertices v_0, v_5, v_8 and v_{10} are cut vertices in the network of Example 2.

The existence of the cut vertices was perceived by Rubin[13] (although he did not use the network terminology). The text string may be cut into separate pieces at cut vertices without losing optimality. Therefore the original problem recurs as a set of smaller subproblems which can be solved independently. The number of the cut vertices will depend greatly on the characteristics of the dictionary substrings and on the input text.

The following notation is used throughout the paper:

a = the size of the input alphabet,
$Bt = \lceil \log_2 a \rceil$, the minimal length of an input character in bits (byte length),
$S = s_1 s_2 \ldots s_n$, the text string to be encoded,
$|S|$ = the number of characters in S,
D = the dictionary used for the encoding,
l_i = the ith substring of D, $i = 1, \ldots, k$,
$|l_i|$ = the length of l_i in multiples of Bt,
$lmax = \max \{|l_i| \mid i = 1, \ldots, k\}$,
$L = |l_1| + |l_2| + \ldots + |l_k|$
c_i = the code word corresponding to the substring l_i
$||c_i||$ = the length of c_i in bits,
$cmin = \min \{||c_i|| \mid i = 1, \ldots, k\}$,
$cmax = \max \{||c_i|| \mid i = 1, \ldots, k\}$,
$A(D, S)$ = the compressed form of the string S, achieved with the dictionary D and the encoding algorithm A
$||A(D, S)||$ = the length of $A(D, S)$ in bits,
$r = ||A(D, S)||/(|S| * Bt)$, the compression ratio,
distance (v_j) = the minimum distance from the source to the vertex v_j, and
parent (v_j) = the predecessor of the vertex v_j on the shortest path

3. A SPACE-OPTIMAL ENCODING ALGORITHM AND ITS COMPLEXITY

In this section we present a new optimal encoding algorithm, based on the network formulation. Although the shortest path from the source to the sink is all we need, the algorithm will also find the shortest paths from the source to all the other vertices that can be reached from the source. The time and space complexities of the algorithm are also analysed.

3.1 The algorithm

In the general algorithm for shortest paths in acyclic networks [18] we first perform the initialisations: distance $(v_0) = 0$ and parent $(v_0) = 0$. Then all the vertices, except the source, are scanned in topological order, and for each vertex v_j we calculate

$$\text{distance } (v_j) = \min \{\text{distance } (v_i) + \text{weight } (v_i, v_j) \mid (v_i, v_j) \text{ is an arc in the network}\},$$

and assign parent (v_j) to be the vertex v_i that corresponds to this minimum (in what follows, the parent is expressed indirectly using an index to the dictionary). It is easy to see that each vertex and each arc is considered only once in this process. After computing the shortest path by using the parent pointers.

In our application, the topological ordering of the nodes is implicitly defined as the order in which the characters are read from an input file. The existence of the cut vertices reduces the storage requirements, because the shortest path can be buffered and output immediately after a cut vertex has been found. If the buffer becomes full and no cut vertex has yet been found, we have to cut the network at a vertex which possibly does not belong to the shortest path. This situation is discussed in more detail in the next section.

The encoding algorithm consists of three parts: the computation of the shortest path, the string matching process for determining the arcs of the network, and the output process.

The shortest-path process and its communication with the other processes is shown in Fig. 1. The algorithm operates similarly to that of Wagner,[20] but processes the string in the forward direction and uses the cut vertices to make the text file compression feasible.[13] The correctness of the algorithm follows directly from the correctness of the general algorithm for finding shortest paths in acyclic networks.

The arcs of the network are produced with the extended trie structure (or the pattern-matching machine) introduced by Aho and Corasick.[1] First a trie (Ref. 8, section

```
procedure encode(S: string; D: dictionary);
% This procedure will encode the text S = s1 s2 ... sn
% with the dictionary D = [(li, ci) | i = 1, 2, ..., k}
Create an extended trie containing the dictionary substrings li;
Initialise the buffer;
distance(v0): = 0;  parent(v0): = 0; cutpoint: = 0;
for each character sj in S do
  Consult the extended trie to find the set IND of indices which
    defines those substrings which match with the original text
    and end at the position j;
  distance(vj): = ∞;
  for each index i in IND do
    d: = |li|; w: = ||ci||;
    if distance(vj) > distance(vj-d) + w then distance(vj): = dis-
      tance(vj-d) + w; parent(vj): = i;
  if j-lmax > cutpoint then
    move parent(vj-lmax) into the output buffer;
    if vj-lmax is a cut vertex then
      Traverse the shortest path in the buffer outputting the code
        words;
      Reset the buffer; cutpoint: = j-lmax;
for j: = lmax-1 downto 0 do move parent(vn-j) into the output
buffer; output the shortest path codes from the buffer;
end encode;
```

Figure 1. The encoding algorithm. The block structure is implied by the indentation.

(*a*) T, H, I, S, _, TH, HIS, IS, IS_, IT

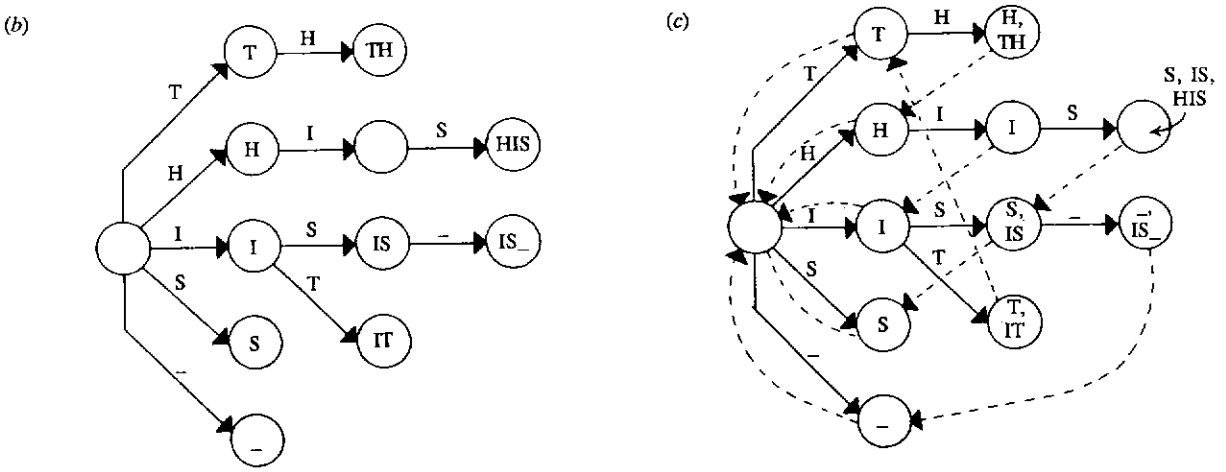


Figure 2. (*a*) The dictionary strings. (*b*) The trie consisting of the strings which are shown inside the nodes (these strings initialise the output sets). (*c*) The extended trie with the final output sets. The broken lines represent the failure transitions.

6.3) representing the substrings of the dictionary is created. Fig. 2*b* shows the trie for the strings of Fig. 2*a*. Then the trie is extended to allow fast string matching as follows.[1]

(1) For all nodes v, except the root, a failure transition links v to the node w, whose associated string (from the root to w) is the longest proper suffix of the string associated with v. The failure transition is used when there is no child for the current node to match with the input string.

(2) For all nodes, an output set (possibly empty) is defined which contains all the substrings of the dictionary that are either equal to the string associated with the node or are proper suffixes of it. In our case the output sets contain pointers to the dictionary.

The extended trie of Fig. 2*b* is shown in Fig. 2*c*.

Observe that in order to find the minimum distance from the source to the current vertex we need to maintain only *lmax* previous distance values. Therefore we use a set of *lmax* elements, consisting of (distance, parent)-pairs. The parent field is actually a pointer to the dictionary, and it is used indirectly to find (*a*) the difference in the indexes of the current and the parent vertices, (*b*) the weight of the arc leading from the parent vertex to the current vertex, and (*c*) the actual code word attached to the arc. When a vertex is transferred from the set to the output buffer, only the parent field is moved.

The size of the output buffer is not known beforehand and therefore a dynamic storage allocation is needed. There are three buffer operations: first, an empty buffer is created by initialising a free-space indicator, which indicates where the next parent value is to be stored. Secondly, a new element is inserted in the next vacant slot. Thirdly, to output the contents of the buffer, we traverse the buffer from right to left by following the parent pointers, turn the pointers, and then traverse the buffer from left to right, outputting the shortest-path code words.

The cut vertices are found by maintaining a cut pointer, which continuously indicates a possible-cut vertex v_k. Each time the procedure finds a new arc (v_{j-d}, v_j), $j-d < k < j$, the cut pointer is updated to the value j. When the cut pointer has a value *lmax* characters behind the current character, a true cut vertex has been found and the contents of the output buffer can be stored and the buffer reset. After this the pointer is initialised to the current vertex. The initialisation prevents us from finding all cut vertices (at most *lmax* in each output phase). However, in practice this has not even been necessary because of the numerous cut vertices and the large size of the buffer compared to the value *lmax*.

3.2 A complexity analysis of the optimal algorithm

It is a well-known fact[18] that the single-pair shortest-path problem for acyclic networks can be solved in $O(n+m)$ time, where n is the number of vertices of the input network and m is the number of arcs. This, however, requires that the model of computation be a random access machine (RAM). In the case of text files, the corresponding networks are very large compared to those in the analyses given in Refs. 15 and 19. Therefore we assume here a two-level machine model: computation in the internal memory is much cheaper than the cost of transferring data between the internal and the external memories. In this model the best algorithm is that which needs only a single left-to-right scan over the input network to produce the compressed form of the string. Our algorithm will in fact do just that because of the numerous close-cut vertices. However, in the worst case the buffer may have to be stored temporarily in the external memory.

Let us assume that the dictionary and the extended trie are small enough to fit into the internal memory simultaneously (in practice they are to a high degree overlapping data structures). Then the string matching can be done at the same time as the shortest-path search and does not demand any extra accesses to the external memory. For the dictionary and the trie, a working space of $O(L)$ storage locations is needed.

In the best case, external storage is used only for input and output yielding a total of $(1+r)*n*Bt$ bits. In the worst case, no cut vertex is found and therefore $n*\lceil \log_2 k \rceil$ bits, where k denotes the number of dictionary strings, are needed for the output buffer (Fig. 3). As the final outputting and the buffer handling are interleaved

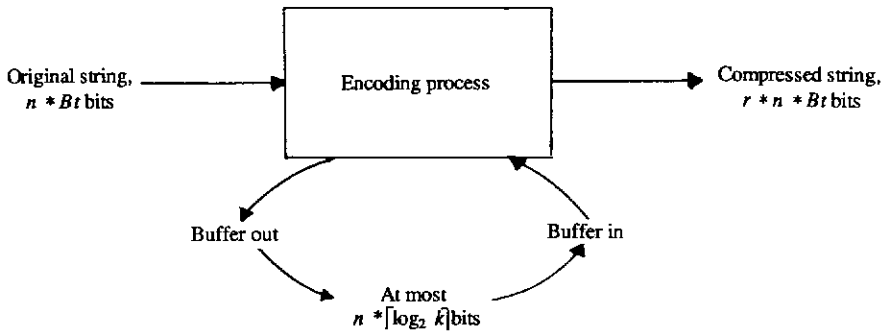


Figure 3. Space requirements for the encoding process.

with each other, the total storage space for these actions is in practice substantially less than $r*n*Bt+n*\lceil \log_2 k \rceil$.

The running times of the shortest-path process and the output process are $O(n+m)$ and $O(n)$, respectively, in the standard RAM model (2, section 1.3). The time complexity of the string-matching process depends on the implementation of the trie. A node of the trie could be implemented by an array of size a, a linked list, or a binary tree. In our application the maximum number of children of a node, denoted by c, is always small (except for the root). This is due to the nature of a typical text. Aho and Corasick[1] proved that the construction of the extended trie takes at most $O(L*c)$ time and that the total number of transitions during the pattern matching process is at most $O(n*c)$, if the root is implemented as an array and all other nodes as linked lists. If all other nodes, except the root, are implemented as binary trees, the construction time would be $O(L*c*\log_2 c)$, whereas the processing time would be only $O(n*\log_2 c)$. The former alternative yields a total time of $O(L*c+n*c+m)$ and the latter $O(L*c*\log_2 c+n*\log_2 c+m)$.

Let us next determine the number of storage blocks transferred between the internal and the external memories. Let p denote the block (page) size in bits. In the best case, the buffer never overflows into the external memory and only $[(1+r)*n*Bt/p+O(1)]$ memory blocks have to be transferred between the internal and the external memories. In the worst case, the output buffer must be twice stored in and retrieved from the external memory because of the two traversals through the buffer. However, if the file system allows memory blocks to be accessed randomly, only one buffer swapping is needed. The output buffer is managed as a stack (Fig. 4): the contents of the buffer are written into the external memory block by block until a cut vertex is found. Then the blocks are read in the reverse order and the shortest path is output. The latter process is again reversed in order to arrange the code sequence properly. This can be done without leaving any gaps inside the blocks because the block boundaries can be computed, using the distance values, which are maintained for each vertex in the shortest path. The technique totals $\{[n*Bt+2*n*\lceil \log_2 k \rceil+r*n*Bt]/p+O(1)\}$ block transmissions.

In the above we have assumed that random access to the external memory blocks is possible; we can explicitly state the absolute address in the file operations. However, as can be seen from Fig. 4, only sequential access is needed. Further, the reverse processing of the memory blocks can be restricted to either reading or writing.

In a sense the algorithm proposed by Wagner[19] is a mirror image of our algorithm: after reversing the dictionary substrings, the shortest-path calculation proceeds from the sink to the source and the compressed form of the string is output without reversing the parent pointers. If we compress phrases of limited length, the output buffer does not overflow. In practice, the existence of the cut vertices, which Wagner did not exploit, usually leads to a non-overflowing buffer even in the case of large text files.

There is a further interesting variant of our algorithm if we maintain the parent pointers forwards instead of backwards. This leads in a natural way to the solution of Rubin:[13] maintain at most *lmax* partial codings simultaneously, and when a cut vertex is met, output the shortest partial coding. This approach clearly demands more internal memory than the solution we have

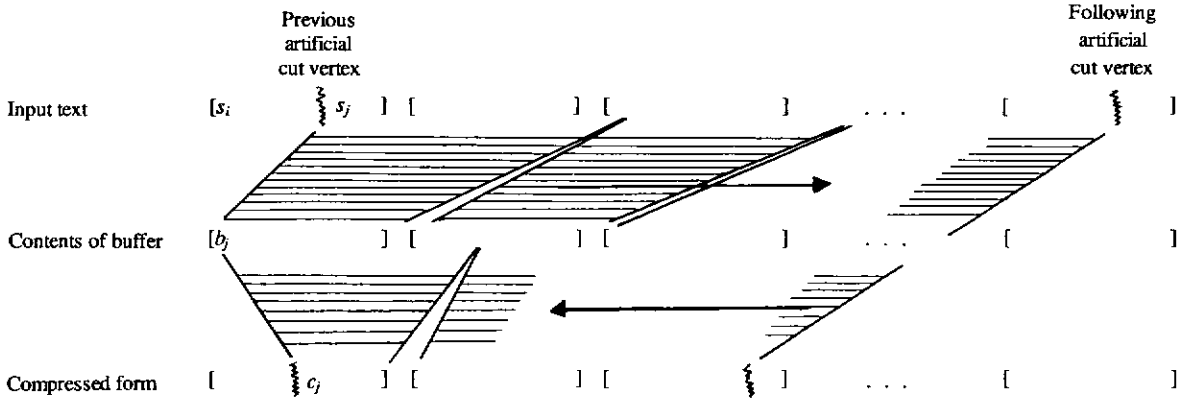


Figure 4. The phases of the encoding as seen on the external storage device. The page boundaries are denoted by [ ] and the buffer boundaries by ⦚. The direction of the processing for the memory blocks above and below is indicated by an arrow.

proposed, and in the worst case the storage requirements are too great.

4. THE PERFORMANCE OF THE OPTIMAL ENCODING ALGORITHM WITH A RESTRICTED BUFFER

The output buffer of the space-optimal encoding algorithm of Section 3 is unlimited in size. In this section we analyse the effect of a fixed-length buffer on the compression gain. As long as the buffer is small enough to fit into the internal memory, no data has to be moved between the memory levels and we have a one-pass encoding algorithm not unlike the longest match heuristic.

The encoding algorithm with a restricted buffer (OPT_{buf}) is very similar to the algorithm in Fig. 1, the only difference being that we need a procedure for handling a full output buffer with no cut vertex. In OPT_{buf} this is done by determining an artificial cut vertex which is the vertex with the smallest distance value among the latest *lmax* vertices of the set, which serves as the lengthening piece to the buffer. Ties are resolved in an arbitrary fashion. The buffer contents are output up to the artificial cut vertex, and the computation of the shortest path is continued with the artificial cut vertex as a new source. Before filling up the buffer again, we need to update the remaining distance values of the set (those to the right of artificial cut vertex) according to the changed situation. Therefore we have to maintain also the corresponding input characters for the vertices in the set.

We study next the worst-case compression gains of OPT_{buf}. The four dictionary types to be considered are the following.

(1) A dictionary with fixed-length code words, called here also a code-uniform dictionary.

(2) A non-lengthening dictionary, in which the length of any code word never exceeds that of the corresponding dictionary substring, i.e. $||c_i|| \leqslant |l_i|$ times Bt for all $i = 1, 2, \ldots, k$. This guarantees that the length of the encoded form of a text string can never be longer than the original string.

(3) A suffix dictionary, which contains, in addition to the l_i-strings, also all their proper suffixes.

(4) A general dictionary, in which none of the above restrictions need be present.

In all cases we assume that every character of the input alphabet is a dictionary string. Thus the bandwidth-limited network corresponding to an input text will always have a base path. This guarantees that, although a heuristic may at some point choose a path different from the optimal, it will not fail to proceed.

Let $OPT_{buf}(D, S)$ and $OPT(D, S)$ denote the compressed form of a text string S obtained when using the dictionary D in the optimal algorithm with a buffer of size buf (measured in $\lceil \log_2 k \rceil$-bit units) and the optimal algorithm with an unrestricted buffer. Further, let R denote the ratio $||OPT_{buf}(D, S)||/||OPT(D, S)||$. For an unbounded buffer we have trivially $R = 1$. The relative loss caused by a bounded buffer is only of the order $lmax^2/\text{buf}$. The next theorem shows also that the OPT_{buf} algorithm is not sensitive to the type of dictionary used.

Theorem. Let D be a dictionary and S a text string. The following bounds are valid for the compressed forms obtained by the optimal algorithm with a restricted buffer of size buf.

(*a*) If D is an arbitrary dictionary, then

$$1 \leqslant R \leqslant 1 + \frac{lmax(lmax-1)cmax}{\text{buf } cmin}.$$

(*b*) If D is a code-uniform dictionary, then

$$1 \leqslant R \leqslant 1 + \frac{lmax(lmax-1)}{\text{buf}}.$$

(*c*) If D is a non-lengthening dictionary, then

$$1 \leqslant R \leqslant \begin{cases} 1 + \dfrac{lmax(lmax-1)Bt}{\text{buf } cmin}, & cmin \leqslant Bt < cmax \\ 1 + \dfrac{lmax(lmax-1)cmax}{\text{buf } cmin}, & cmin < cmax \leqslant Bt \end{cases}$$

and

(*d*) If D is a suffix dictionary, then

$$1 \leqslant R \leqslant 1 + \frac{lmax\, cmax}{\text{buf } cmin}.$$

Proof. (*a*) Let $G = (V, E)$ be the bandwidth-limited network corresponding to the string S and the dictionary D. Let

$$v_0 = v_{i_0}, v_{i_1}, v_{i_2}, \ldots, v_{i_h}, v_{i_{h+1}} = v_n$$

be the artificial cut vertices produced by the OPT_{buf} algorithm. Further, let

$$v_{r_1}, v_{r_2}, \ldots, v_{r_h}, v_{r_{h+1}}$$

be those optimal path vertices which are nearest and to the right of the corresponding artificial cut vertices v_{i_j}. This implies that the vertices v_{i_j} and v_{r_j} may coincide for some j. Finally, let

$$v_{s_1}, v_{s_2}, \ldots, v_{s_{h+1}}$$

be those optimal path vertices which have been nearest (to the right or to the left) to the corresponding v_{i_j} vertices and reside in the set (which serves as a lengthening piece to the buffer) at the moment the buffer overflows. Again, v_{s_j} and v_{r_j} may coincide for some j. By the definition of the bandwidth-limited networks, at least one of any *lmax* successive vertices is always on the optimal path. The situation is illustrated in Fig. 5.

The distance computed by OPT_{buf} is denoted by dist (v) to distinguish it from the corresponding quantity distance (v) of the optimal algorithm. By the definition of the artificial cut vertex, it is obvious that

$$\text{dist}(v_{i_1}) = \text{distance}(v_{i_1}) \leqslant \text{distance}(v_{r_1}).$$

Let us approximate the distance between two consecutive artificial cut vertices v_{i_j} and $v_{i_{j+1}}$. When the output buffer overflows, dist ($v_{i_{j+1}}$) is the minimum dist value of the vertices in the set. Hence,

$$\begin{aligned} \text{dist}(v_{i_{j+1}}) - \text{dist}(v_{i_j}) &\leqslant \text{dist}(v_{s_{j+1}}) - \text{dist}(v_{i_j}) \\ &\leqslant \text{distance}(v_{s_{j+1}}) - \text{distance}(v_{r_j}) \\ &\quad + (lmax - 1)\, cmax \\ &\leqslant \text{distance}(v_{r_{j+1}}) - \text{distance}(v_{r_j}) \\ &\quad + (lmax - 1)\, cmax. \end{aligned}$$

The second inequality follows from the fact that the difference in the dist values must not exceed the length of

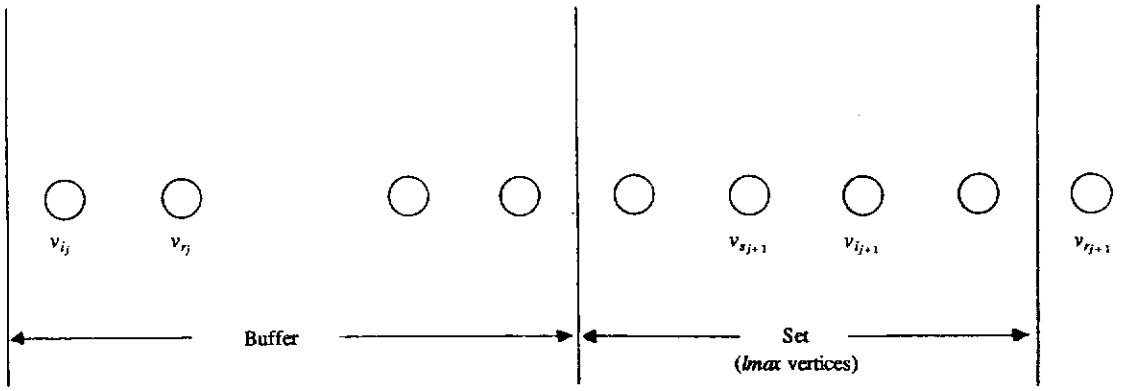


Figure 5. The artificial cut vertices v_{i_k}, their right neighbours v_{r_k} and set neighbours v_{s_k}.

Table 1. Summary of the performance of the optimal algorithm with a restricted buffer of size buf. × indicates that the property exists in the dictionary

Dictionary D			
Suffix	Code-uniform	Non-lengthening	Upper bound for the ratio $\|\|\text{OPT}_{\text{buf}}(D,S)\|\|/\|\|\text{OPT}(D,S)\|\|$
—	—	—	$1+lmax(lmax-1)\,cmax/(\text{buf}\ cmin)$
×	—	—	$1+lmax\,cmax/(\text{buf}\ cmin)$
—	×	—	$1+lmax(max-1)/\text{buf}$
—	—	×	$1+\dfrac{lmax(lmax-1)*min\{Bt,cmax\}}{\text{buf}\,cmin}$
×	×	—	$1+lmax/\text{buf}$
×	—	×	$1+lmax\,cmax/(\text{buf cmin})$
—	×	×	$1+lmax(lmax-1)/\text{buf}$
×	×	×	$1+lmax/\text{buf}$

the path which goes from v_{i_j} to v_{r_j} and after that takes the optimal path to $v_{s_{j+1}}$. In the worst case, we have to pay the cost $(lmax-1)\,cmax$ to join v_{i_j} and v_{r_j} along the base path.

Thus we have

$$\begin{aligned}||\text{OPT}_{\text{buf}}(D,S)|| &= \sum_{j=0}^{h}(\text{dist}(v_{i_{j+1}})-\text{dist}(v_{i_j}))\\ &\leqslant \text{distance}(v_{r_1})\\ &+\sum_{j=1}^{h}(\text{distance}\,v_{r_{j+1}})-\text{distance}\,v_{r_j})+(lmax-1)\,cmax)\\ &=\text{distance}(v_n)+h(lmax-1)\,cmax.\end{aligned}$$

The number of artificial break points is at most

$$h\leqslant\left\lceil\frac{|S|+1}{\text{buf}}\right\rceil-1\leqslant\frac{|S|}{\text{buf}}.$$

Further, $|S|\,cmin/lmax$ is the minimal length of the compressed form of the string achieved with any algorithm and dictionary. Therefore,

$$R\leqslant 1+\frac{lmax(lmax-1)\,cmax}{\text{buf}\,cmin}.$$

The parts (*b*)–(*d*) of the theorem can be proved in the same way by estimating the term $(lmax-1)\,cmax$ in the distance calculation according to the dictionary type. □

Table 1 summarises the results we have derived for the OPT_{buf} algorithm.

5. EXPERIMENTS

In order to find out whether our network-formulated algorithm encodes large text files efficiently, we implemented the OPT_{buf} and the LM algorithms.

5.1 Implementation

The implementation of the longest match algorithm is straightforward. The substrings of the dictionary are organised into a trie structure. To accomplish fast string matching, the two top levels of the trie are recorded in a two-dimensional table.[15] Thus the table forms a digram hash table for a collection of subtries, i.e. the element $[s_i, s_j]$ points to the subtrie with the prefix $s_i s_j$. The input string is buffered to accomplish a one-pass procedure in the two-level machine model sense.

The OPT_{buf} algorithm takes some time to construct the extended trie before performing the actual encoding process. The top level of the trie, consisting of the first characters of the dictionary substrings, is implemented as an array instead of a list structure. This is advantageous when the trie is constructed, and also in the actual encoding process. The trie was not augmented to form a deterministic finite automation because the additional work and space needed was considered too high in respect of the advantage gained. This means that in order to process a single input symbol, we may have to use the failure function more than once to find a match, whereas in a deterministic automation only one step is needed.

The output buffer is maintained, using (*a*) a threshold size for the buffer string, and (*b*) a tentative cut vertex pointer. The threshold size gives the minimum number of characters the output buffer must contain before the output process can be triggered. The threshold size is fixed at some 'reasonable' minimum value, say 50–200 characters, to ensure that the buffer is not written out too frequently (at every cut vertex). The tentative cut vertex pointer is used to find cut vertices for triggering the output process. The pointer is initialised to the threshold size and updated every time an arc is found whose starting vertex is to the left of the current cut vertex (this, of course, is not done before the encoding process has passed the initialisation value of the pointer). While we are processing a character which is situated $(lmax+1)$ characters ahead of the tentative cut vertex, we know that the tentative cut vertex pointer is actually determining a true cut vertex. It is then that the buffer is output. To avoid complications, the threshold size of the buffer string must always be larger than *lmax*.

The output buffer size is a parametrised to assist in the determination of the losses due to the restricted buffer size. During the course of our experiments, however, we noticed that even a moderate-sized buffer guaranteed an optimal encoding, because cut vertices were repeatedly found close to each other. This phenomenon is, of course, greatly dependent on the dictionary substrings and their relations to each other, as well as the text string to be encoded. It is our belief, however, that in practical situations a buffer for, say, 1000 characters is always enough to guarantee an optimal coding result.

The encoding programs were written in Pascal and run on a DEC 2060. No optimising was done at the source-code level, but the 'optimise' switch was used in the compilation.

5.2 Input data

The experiments were performed with five Pascal source program files and five English text files as input data. The files were 3500–44000 characters in length. A code table was generated for both types of file and then used to encode all the files of the same type. The dictionary substrings were generated by using a modification[12] of Rubin's incremental coding algorithm.[13] The algorithm searches first for frequently occurring digrams and replaces them with characters (called codes) which were not present in the original file. Thereafter, the codes are treated as ordinary characters, the frequent digrams and a new set of codes are re-formed and the process continues until either there are no more code characters left or no compression gain is achieved in any substitution. The modified algorithm does not discard any of the substrings once they are coded. This results in a collection of dictionary strings, many of which are proper substrings of others. Thus it only rarely happens that dictionary substrings partially overlap, without one being totally included in another.

The fact that many short dictionary strings are substrings of others reduces the alternative ways to encode a string and thus also the difference between the results obtained by the optimal and the longest match algorithms. In some places, on the other hand, there may be exceptionally many arcs in the problem network. This is the case when we encode a long string consisting of identical characters one after another (e.g. the space character) and when the dictionary contains substrings in which two, three, four or more identical characters are concatenated. It is in such sitations that we get the most benefit from the trie structure of Aho and Corasick.

In our experiments, the longest dictionary substrings had 14 and 8 characters respectively in the source program dictionary and the English text dictionary. The average lengths of the substrings, excluding the single characters, were 3.4 and 2.6. The distribution of the lengths of the blocks that can be coded optimally, i.e. the distances from one cut vertex to another, is shown in Table 2. The distance 1 expresses the percentage of the base path steps over which no other arcs pass. Pascal source programs have many such steps. This is due to the fact that many substrings of the dictionary are associated with the identifiers (e.g. variable names) used in the particular program from which the substrings have been extracted, and they therefore have no correspondence in any other program.

Table 2. Distribution of the distances from one cut vertex to another

Distance between successive cut vertices (characters)	Relative number (%) of blocks in Pascal source programs	Relative number (%) of blocks in English text
1	78.7	68.7
2	10.1	24.2
3	2.9	4.4
4	3.2	0.8
5	1.3	1.3
6–10	2.5	0.6
11–20	0.9	0.1
21–30	0.3	0.0
31–50	0.1	0.0
51–100	0.0	0.0

The distance 2 in Table 2 shows the isolated digram replacements, i.e. those digram codings which are not part of a longer, optimally coded block. In the English texts there are more frequently used digrams and trigrams (of, −ed, in-, the, and, -ion, -ing, pre-...) than in the source programs. However, because the source programs were formatted by means of an indentation, the effect of the different combinations of carriage return, line feed, tabulator and space characters begins to show from distance 4 onwards.

The average block lengths were 1.68 and 1.46 for the source programs and English texts respectively. If the base path steps are excluded, the lengths were 4.27 and 2.46. The maximum block length was 66 for program files and 77 for the English texts.

5.3 Results

Table 3 shows the average encoding times and compression gains for the OPT_{buf} and LM algorithms. The calculations are based on the 7-bit ASCII code, and all the data which are needed to decode the file are stored within the compressed file.

As can be seen, the difference in compression gains is nominal, as was expected in the light of earlier

Table 3. Comparison of the OPT_{buf} and the LM algorithm*

Type of file to be encoded	Algorithm	
	OPT_{buf}	LM
Pascal source programs		
CPU	121.6	108.8
Compression gain	30.6	30.4
English text		
CPU	120.5	110.2
Compression gain	27.3	27.2

CPU refers to the time (ms) needed to encode 1000 bytes, and the compression gain means the average percentage of space saved in the encoding as compared to the original file space.

experiments.[13, 15] In our test case this is largely due to the properties of the dictionary substrings and the uniform coding scheme.[7] The encoding times of the algorithms are now comparable, due to the advanced data structures in the OPT_{buf} algorithm. With short files the OPT_{buf} algorithm performed even more rapidly than the LM heuristic. This was due to the initialisation phase, in which the OPT_{buf} algorithm used only $\frac{2}{3}$ of the time required by the LM implementation. The time needed to encode 1000 bytes was longest for the short files, and as the file size increased the time gradually decreased. The figures thus also validate the two-level machine model: the time needed to transfer the data from the external to the internal memory and vice versa is indeed a significant part of the encoding time.

6. CONCLUSIONS

In this paper we have reduced a text encoding problem to a problem of finding a shortest path in acyclic networks. The network representing the coding alternatives has special characteristics which we can take advantage of in the encoding process for speeding up the shortest-path search. The network is generated during the encoding process. This helps us in managing large networks which are otherwise too extensive to be handled with conventional algorithms.

An algorithm for space-optimal encoding is presented. A slight variation of this algorithm led to an effective approximation algorithm which almost always guarantees a space-optimal encoding, and runs nearly as fast as the longest match algorithm.

The critical factor in our approximation algorithm is that the performance figures are guaranteed only if the problem network includes the base path. In our experience, there are not many practical applications where this condition is not satisfied or cannot be enforced, for example by adding an extra bit to indicate whether a code word or a character follows.

It seems likely that the ideas expressed in this paper will be applicable when developing an efficient one-pass compression algorithm, that is to say, an algorithm which builds the dictionary gradually during the actual coding process (see for example Ref. 6). In that case a heuristic, which generates new and discards old strings from the dictionary (see Ref. 10), should be introduced. Meyer[11] has proposed an incremental string-matching algorithm which allows insertions into the extended trie. For this, however, we should require a wholly dynamic data structure which would also support deletions.

Interest has recently been expressed in encoding algorithms which work efficiently in parallel computers.[5] Our optimal encoding algorithm includes a high degree of parallelism because the shortest path between two cut vertices can be found independently of the other parts of the problem network. It would be reasonable to divide the network into sections, so that two neighbouring sections overlap with at least *lmax* characters. The computation would then begin with the determination of the cut vertices in the overlapping parts of a section, and if none were found an artificial cut vertex would be created according to the scheme described in connection with the restricted buffer algorithm. Further, when encoding one section, the string-matching process, the shortest path-process and the output process can all be run parallel. However, some synchronization problems may occur during the output phase because of the varied lengths of the sections to be output. The implementation details and the efficiency of this parallel encoding algorithm will depend greatly upon the parallel model of computation, in which the encoding is considered.

Acknowledgements

The authors are grateful to Matti Jokinen and Olli Nevalainen for their helpful comments concerning the subject of this paper. We wish to thank also Jan van Leeuwen for calling our attention to the string-matching algorithm of Aho and Corasick.

REFERENCES

1. A. V. Aho and M. J. Corasick, Efficient string matching: an aid to bibliographic search. *Communications of the ACM* **18** (6), 333–340 (1975).
2. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley (1974).
3. D. Cooper and M. Lynch, Text compression using variable-to fixed-length encodings. *Journal of the American Society for Information Science*, pp. 18–31 (1982).
4. A. Fraenkel, M. Mor and Y. Perl, Is text compression by prefixes and suffixes practical? *Acta Informatica* **20**, 371–383 (1983).
5. M. E. Gonzalez Smith and J. A. Storer, Parallel algorithms for data compression. *Journal of the ACM* **32** (2), 344–373 (1985).
6. M. Jakobsson, Compression of character strings by an adaptive dictionary. *BIT* **25** (4), 593–603 (1985).
7. J. Katajainen and T. Raita, An analysis of the longest match and greedy heuristic for text encoding. Manuscript, University of Turku (1986).
8. D. E. Knuth, Fundamental algorithms. *The Art of Computer Programming*, vol. 1. Addison-Wesley, New York (1973).
9. B. Marien and I. H. Sudborough, Bandwidth constrained

NP-complete problems. *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pp. 207–217 (1981).
10. A. Mayne and E. B. James, Information compression by factorising common strings. *The Computer Journal* **18** (2), 157–160 (1974).
11. B. Meyer, Incremental string matching. *Information Processing Letters* **21**, 219–227 (1985).
12. T. Raita, An automatic system for file compression. *The Computer Journal* **30** (1), 80–86 (1987).
13. F. Rubin, Experiments in text file compression. *Communications of the ACM* **19** (11), 617–623 (1976).
14. F. Rubin, Cryptographic aspects of data compression codes. *Cryptologia* **3** (4), 202–205 (1979).
15. E. J. Schuegraf and H. S. Heaps, A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval* **10**, 309–319 (1974).
16. J. A. Storer, Data compression: methods and complexity issues. *Ph.D. Thesis*, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J. (1978).
17. J. A. Storer and T. G. Szymanski, Data compression via textual substitution. *Journal of the ACM* **29** (4) 928–951 (1982).
18. R. E. Tarjan, Data structures and network algorithms. *Society for Industrial and Applied Mathematics* (1983).
19. R. A. Wagner, Common phrases and minimum-space text storage. *Communications of the ACM* **16** (3), 148–152 (1973).
20. R. A. Wagner, An algorithm for extracting phrases in a space-optimal fashion. *Communications of the ACM* **16** (3), 183–185 (1973).