Efficient Optimal Recompression*

Shmuel T. Klein

Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 535 3325 e-mail: tomi@cs.biu.ac.il

An efficient variant of an optimal algorithm is presented, which reorganizes data that has been compressed by some on-the-fly compression method, into a more compact form, without changing the decoding procedure. The algorithm accelerates and improves the space requirements of a known technique based on a reduction to a graph-theoretic problem, by reducing the size of the graph, without affecting the efficiency of the solution that is optimal for the given encoding method. The new method can effectively improve any static dictionary compression scheme using a static encoding method, and in particular some LZ77 variants.

Received July 1996

1. INTRODUCTION AND BACKGROUND

Text compression techniques are often divided into statistical methods, such as Huffman coding (Huffman 1952), or arithmetic coding (Witten et al. 1987), and dictionary methods, based generally on the work of Lempel and Ziv (Ziv and Lempel 1977, Ziv and Lempel 1978). The statistical methods assign codewords to the elements making up the text, the lengths of these codewords depending on the frequencies of the corresponding elements. Dictionary methods replace variable length substrings of the text by (shorter) pointers to a dictionary in which a collection of such substrings has been stored. Depending on the application and the implementation details, each method can outperform the other, as long as only the *compression savings* are of concern.

There are, however, other criteria by which the various compression methods should be compared. While the primary concern is generally to reduce the size of the given file as much as possible, the *time complexity* of the coding routines may also be a relevant factor. For certain applications, such as data transmission over a communication channel, both coding and decoding ought to be fast. For other applications, like the storage of the various files in a large static full text information retrieval system, compression and decompression are not symmetrical tasks. Compression is done only once, while building the system, whereas decompression is needed during the processing of every query and directly affects response time. One may thus use extensive and costly preprocessing for compression, provided reasonably fast decompression methods are possible. Finally, the *internal memory* requirements of a proposed algorithm may also be an important criterion, in particular on small machines.

In LZ77 (Ziv and Lempel 1977) and its variants, the dictionary is in fact the previously scanned text, and pointers to it are of the form (d, ℓ), where d is an offset (the number of characters from the current location to the previous occurrence of a substring matching the one that starts at the current location), and ℓ is the length of the matching string. There is therefore no need to store an explicit dictionary.

One of the problems of LZ77 is how to locate previous occurrences of substrings in the text. The simple method of scanning the whole text backwards for each processed character might be prohibitively slow. Many alternatives have been suggested, including, among others, the use of binary trees (Bell 1986), hashing (Brent 1987, Williams 1991), and Patricia trees (Fiala and Greene 1989).

The question of how to parse the original text into a sequence of substrings is a problem common to all dictionary based compression techniques. An optimal technique for a static dictionary is mentioned in Wagner 1973. Storer and Szymanski 1982 give an optimal parsing algorithm for the sliding window method, and Hirschberg and Stauffer 1994 present parallel algorithms for optimal parsing. Generally, for static dictionary techniques, the parsing is done by a *greedy* method, i.e., at any stage, the longest matching element from the dictionary is sought, though non-greedy methods have also been considered (see Horspool 1995) and are used, e.g., in the popular `gzip` program. A greedy approach gives good compression (Katajainen and Raita 1992), and is easy to implement by means of a trie, but is not necessarily optimal. Because the elements of the dictionary are often overlapping, a different way of parsing might yield better compression. For example, assume the dictionary consists of the strings

*Partially supported by grant 8560195 of the Israeli Ministry of Science and Arts.

$D = \{$`abc`, `ab`, `cdef`, `d`, `de`, `ef`, `f`$\}$ and that the text is $S =$ `abcdef`; assume further that the elements of D are encoded by some fixed-length code, which means that $\lceil \log_2(|D|) \rceil$ bits are used to refer to any of the elements of D; then parsing S by a greedy method, trying to match always the longest available string, would yield `abc-de-f`, requiring 3 codewords, whereas a better partition would be `ab-cdef`, requiring only 2. Moreover, for dynamic techniques such as LZ77 variants, for which the dictionary is the encoded text itself, finding at each step the longest matching string may be just as time consuming as finding the optimal parsing.

The various dictionary compression methods differ also by the way they encode the elements. This is most simply done by a fixed length code, as in the above example. A more involved technique (Fiala and Greene 1989), uses a static variable length encoding of the d and ℓ values. Pushing this idea even further, one may use a dynamic variable length code, optimally adapting itself to the frequencies of the occurrences of the different values of d and ℓ: Brent (Brent 1987), suggests the use of Huffman coding for the (d, ℓ) pairs.

We are concerned here with a way of optimally parsing the text, which may be applied to a process called *recompression*. There are many systems today that offer on-the-fly, very fast, compression of files of any kind. These systems are used to better exploit available disk space, by compressing any file before writing it to the disk. But this is only attractive if the time spent on compression is hardly noticeable, and similarly, decompression must be fast, so that a compressed file may be read without delay. Recompression is useful in a situation where a number of files have already been compressed by the fast method, and the user wishes now to reorganize the data on his disk into a more compact form. Time is less critical for this reorganization process, and the new compression algorithm might in fact be independent of the former and start from scratch. But the constraint is that the new encoded form of the recompressed file must be compatible with the original encoding, so that the same decompression method may be used. In other words, a single decoding routine should be able to process a file, regardless of it having been compressed or recompressed.

The method described below has already been mentioned (Schuegraf and Heaps 1974, Katajainen and Raita 1989), and achieves *optimal* recompression in the sense that once the method for encoding the elements is given, it finds the optimal way of parsing the text into such elements. Obviously, different encoding methods might yield different optimal parsings. Returning to the above example of the dictionary D and text S, if the elements `abc`, `d`, `de`, `ef`, `f`, `ab`, `cdef` of D are encoded respectively by 1, 2, 3, 4, 5, 6 and 6 bits, then the parsing `abc-de-f` would need 9 bits for its encoding, and for the encoding of the parsing `ab-cdef`, 12 bits would be needed. The best parsing, however, for the given codeword lengths, is `abc-d-ef`, which is neither a greedy parsing, nor does it minimize the number of codewords, and requires only 7 bits.

The way to search for the optimal parsing is by reduction to a well-known graph theoretical problem. This approach is, however, not recommended in Schuegraf and Heaps 1974, because it is too time-consuming. In Katajainen and Raita 1989, sub-optimal solutions are suggested to improve the execution time.

It should be emphasized that the optimality of the algorithm referred to in this paper is only *relative* to a given encoding method for the elements into which the original string has been parsed. The resulting method is not claimed to be globally optimal, and using another scheme, one might well get better compression. The contribution of this paper is a variant of the optimal algorithm that is *efficient* in terms of both time and space: a pruning technique is applied to the graph, which generally reduces the number of both edges and vertices, but still enables the evaluation of an optimal solution for the original graph.

The optimal method and its new variant apply to any static dictionary based compression method with static (fixed or variable length) encoding. The elements to be encoded can be of any kind: strings, characters, (d, ℓ) pairs, etc, and any combination thereof. The proposed technique thus improves a very broad range of different methods, many of which have been published in the scientific literature or as patents.

In the next section we mention some simple recompression methods and present the new method and implementation details. Examples of encoding functions that have been used and satisfy the required conditions are given in Section 3. Section 4 presents some experimental results.

2. RECOMPRESSION

Every recompression algorithm corresponds to another tradeoff between the speed of the encoding process and the compression efficiency. Consider a given location in the text, to be encoded by a dictionary compression method. At certain locations, there might be more than one possible choice for the dictionary element to be substituted for the following characters. The algorithm used for scanning the dictionary (linear search, binary search, hashing, etc.) induces an order on the dictionary elements. The range of tradeoff alternatives extends from finding, relative to the ordering at hand, the *first* appropriate element (fastest method, but yielding inferior compression), through considering the k first such elements of the dictionary, for some integer $k > 1$, and choosing the best element among these, up to scanning *all* possible alternatives and selecting the locally opti-

mal element (slower, but giving improved compression).

2.1. Simple recompression methods

For LZ77 and many of its variants, the (d, ℓ) = (*distance, length*) pointers are restricted to $d \leq N$ for some fixed N, that is, a string is considered as recurrent only if its previous occurrence is within a finite window of fixed size preceding the current location. A simple recompression heuristic is therefore to increase N, which increases the probability of finding a good earlier match. However, the compression performance is not necessarily improved, since $\lceil \log_2 N \rceil$ bits are used to encode d.

In Bell 1986, Fiala and Greene 1989, Whiting et al. 1992, the previous occurrences of the current substring are searched for by means of hashing: the current two (or three) characters are hashed to a location in a hash table, which contains a pointer to the previous occurrence of a couple (or triplet) of characters that hashed to the same location. Since hash functions are not injective, different character pairs or triplets may hash to the same location. It is thus possible that the hash table does not provide a pointer to a previous occurrence, although such an occurrence might exist. There are several ways to use simple recompression in this case. Using a larger hash table will reduce the number of collisions and thereby increase the probability of locating a string if it appeared earlier. Taking this idea a step further, and if enough memory space is available, one could get rid of the hashing altogether, and keep, say, for *every* possible character pair, a pointer to its last occurrence.

In the basic LZ77 algorithm, the *longest* substring is sought which matches the current characters. In the implementations using hashing, this is usually approximated by finding first a matching pair or triplet, and then trying to extend the match as far as possible. This obviously does not guarantee that the longest match will be detected. For instance, if the text that has already been scanned is $T = \ldots$`abcde`$\ldots$`abcx`$\ldots$, and the following characters are `abcde`, then applying hashing to the character pair `ab` will yield, in the better case, a pointer to the last occurrence of `ab`, which can only be extended to form a 3-character match, whereas a 5-character match would have been possible; in the worse case, even that 3-character match will be missed, if another character pair, different from `ab` but yielding the same hash value, has appeared after `abcx` in T.

The compression efficiency can be improved, if not only the last occurrence is remembered, but the k last occurrences, for some constant $k > 1$. For example, one could store pointers to the k last occurrences of each character, using a cyclic list for each. The drawback of this method is that the same number of memory locations is reserved for each character, whereas in many applications, in particular in natural language texts, certain characters appear much more frequently than others.

Combining this idea of saving multiple references with the hashing approach above, one could store a cyclic list of k elements for each entry of the hash table, or, which is equivalent, have k different hash tables of identical size. If a good hashing function is chosen, the distribution of the hashed addresses will be close to uniform, even if the single character distribution is not. In case there is a strong bias even after hashing, one could use linked lists for each entry of the hash table, thus allowing lists of varying length (up to some predetermined upper limit, induced by the time constraints), without wasting memory locations. However, since hashing functions are non-injective, all the above lists may now contain pointers to different elements.

Consider for example the following scheme: assume the character set consists of the 256 possible 8-bit strings, that a hash table of $2^{12} = 4K$ entries is used, and that hashing is to be applied on character pairs. One could then hash by truncating the two least significant bits of each character, i.e.:

$$h(a_7 \cdots a_1 a_0\,,\, b_7 \cdots b_1 b_0) \;=\; a_7 \cdots a_2 b_7 \cdots b_2.$$

If the addresses in the lists are stored in 16-bit words, one could even remove the ambiguity resulting from the hashing, by explicitly storing the truncated bits $a_1 a_0 b_1 b_0$ in each element of the list. This leaves 12 bits for the address itself, which is equivalent to setting N, the size of the window into which back references should point, to $4K$.

In the next section we consider even better recompression, which recognizes the fact that the longest-matching-string heuristic not necessarily yields an optimal partition.

2.2. Improved optimal recompression

Consider a text string S consisting of a sequence of n characters $S_1 S_2 \cdots S_n$, each character S_i belonging to a fixed alphabet Σ. Substrings of S are referenced by their limiting indices, i.e., $S_i \cdots S_j$ is the substring starting at the i-th character in S, up to and including the j-th character. We wish to compress S by means of a dictionary D, which is a set of character strings $\{\sigma_1, \sigma_2, \ldots\}$, with $\sigma_i \in \Sigma^+$. The dictionary may be explicitly given and finite, as in the example in the introduction, or it may be potentially infinite, e.g., for the LZ77 variants, where any previously occurring string can be referenced.

The compression process consists of two independent phases: parsing and encoding. In the *parsing* phase, the string S is broken into a sequence of consecutive sub-strings, each belonging to the dictionary D, i.e., an

increasing sequence of indices $i_0 = 0, i_1, i_2, \ldots$ is found, such that

$$S \quad = \quad S_1 S_2 \cdots S_n \quad = \quad S_1 \cdots S_{i_1} S_{i_1+1} \cdots S_{i_2} \quad \cdots,$$

with $S_{i_j+1} \cdots S_{i_{j+1}} \in D$ for $j = 0, 1, \ldots$. One way to assure that at least one such parsing exists is to force the dictionary D to include each of the individual characters of Σ. The second phase is based on an *encoding* function $\lambda : D \longrightarrow \{0,1\}^*$, that assigns to each element of the dictionary a binary string, called its encoding. The assumption on λ is that it produces a code which is uniquely decipherable (UD). This is most easily obtained by a fixed length code, but such a code is only possible for a finite dictionary, and even then it is only efficient from the compression point of view if the distribution of the occurrences of the elements of D in S is nearly uniform. Compression can often be improved by the use of variable-length codes, assigning shorter codewords to elements with higher probability of occurrence. A sufficient condition for a code being UD is to choose it as a prefix code (see Even 1979).

The problem is the following: given the dictionary D and the encoding function λ, we are looking for the optimal partition of the text string S, i.e., the sequence of indices $i_1, i_2, \ldots$ is sought, that minimizes $\sum_{j \geq 0} |\lambda(S_{i_j+1} \cdots S_{i_{j+1}})|$.

To solve the problem, a directed, labeled graph $G = (V, E)$ is defined for the given text S. The set of vertices is $V = \{1, 2, \ldots, n, n+1\}$, with vertex i corresponding to the character S_i for $i \leq n$, and $n+1$ corresponding to the end of the text; E is the set of directed edges: an ordered pair (i, j), with $i < j$, belongs to E if and only if the corresponding substring of the text, that is, the sequence of characters $S_i \cdots S_{j-1}$, can be encoded as a single unit. In other words, the sequence $S_i \cdots S_{j-1}$ must be a member of the dictionary, or more specifically for LZ77, if $j > i+1$, the string $S_i \cdots S_{j-1}$ must have appeared earlier in the text. The label L_{ij} is defined for every edge $(i, j) \in E$ as $|\lambda(S_i \cdots S_{j-1})|$, the number of bits necessary to encode the corresponding member of the dictionary, for the given encoding scheme at hand. The problem of finding the optimal parsing of the text, relative to the given dictionary and the given encoding scheme, therefore reduces to the well-known problem of finding the shortest path in G from vertex 1 to vertex $n+1$.

Dijkstra's algorithm (Dijkstra 1959), may be used to find the shortest path. Its worst case complexity varies, depending on the data structures used, from $O(|V|^2)$ to $O(|E| + |V| \log |V|)$ (see Cormen et al. 1990), which would be particularly disturbing for our intended application. However, in our case the directed graph contains no cycles, since all edges are of the form (i, j) with $i < j$. Thus by a simple dynamic programming method, the shortest path can be found in $O(|E|)$. Nevertheless, when the text includes long runs of repeated characters (like strings of zeros or blanks), the number of possibilities to parse these runs, and hence the number of edges, is quadratic in the number of vertices. This motivated the search for sub-optimal alternatives in Katajainen and Raita 1989.

We suggest here to adhere to the optimal parsing, and to circumvent the worst case behavior by combining the shortest path algorithm with a *pruning* method intended to eliminate a priori such parts of the graph that cannot possibly be part of an optimal path. The pruning process may be applied in all cases for which the labeling function L satisfies the triangle inequality,

$$L_{ij} \leq L_{ik} + L_{kj} \text{ for all } i, k, j \text{ such that } i < k < j,$$

which holds for many practical encoding schemes (see next section for examples).

The set of edges E is constructed dynamically by the algorithm itself. We start with $E = \emptyset$, and adjoin, in order, the edges emanating from vertices $1, 2, \ldots$, unless they fail to pass the following test. When a vertex i is reached, consider the set of its predecessors $\text{Pred}(i) = \{j \mid (j, i) \in E\}$. We then scan the substrings of the text starting at S_i. Suppose that the substring $S_i \cdots S_{j-1}$ is a member of the dictionary, so that the pair (i, j) is a candidate to be adjoined to E. Before adding this edge, check if it is possible to reach vertex j directly from every vertex in $\text{Pred}(i)$, without passing through vertex i. If so, then there is no need to add the edge (i, j) to E, since, because of the triangle inequality, there is no loss in taking the direct edge from the element of $\text{Pred}(i)$ to vertex j. However, if there is even one element in $\text{Pred}(i)$ that has no direct edge to j, then (i, j) must be added to E.

If, after having checked all the edges emanating from vertex i, none of these have been adjoined to E, then there is no need to keep the vertex i in the graph, since it obviously cannot be part of an optimal path from 1 to $n+1$. Thus all the incoming edges on vertex i may be pruned from the graph, and i itself may also be eliminated.

The formal definition of the algorithm is given below.

The algorithm seems non-symmetric with regard to the predecessors and successors of a vertex. This is because the vertices are scanned sequentially. Therefore, when processing vertex i, $\text{Pred}(i)$ is already defined, but not yet $\text{Succ}(i) = \{j \mid (i, j) \in E\}$, the set of i's successors. The set of the *potential* successors of i, Succ_Candidates(i), is defined as the set of those vertices to which there would have been a direct edge if no pruning were used. Some of these edges might ultimately not be adjoined to the graph. Others might in a first stage be added, but might later be deleted.

Note that the triangle inequality is a sufficient condition for reaching an optimal solution with the improved

```
Algorithm Prune
{
    E ←— ∅
    V ←— {1, ..., n, n+1}
    for i ←— 1 to n
    {
        Pred(i) ←— {k | (k,i) ∈ E}
        Succ_Candidates(i) ←— {j | S_i ··· S_{j-1} ∈ D}
        added_edge ←— FALSE
        for all j ∈ Succ_Candidates(i)
        {
            all_connected ←— TRUE
            for all k ∈ Pred(i)
            {
                if (k,j) ∉ E
                then all_connected ←— FALSE
            }
            if not all_connected then
            {
                E ←— E ∪ {(i,j)}
                added_edge ←— TRUE
            }
        }
        if not added_edge then
        {
            V ←— V \ {i}
            E ←— E \ {(j,i) | j ∈ Pred(i)}
        }
    }
}
```

After having applied the pruning algorithm, many edges may have been deleted, as well as some of the vertices. Figure 2 depicts the graph obtained for the same text as for Figure 1, but with the use of the pruning algorithm. The character corresponding to the transition from vertex i to $i+1$ is indicated, even if the corresponding edge has been deleted. Edges drawn as dotted lines were first adjoined to the graph, but later deleted because there was no edge emanating from their endpoints. In this example, the number of edges was first reduced from 21 to 14, and finally to 10.

The example also indicates how the algorithm could be improved. Note that since all the successors of 10 are also successors of 9, the triangle inequality in fact implies that the edge (9,10) could also be eliminated. The reason this is not done in the algorithm is because the loop with running index k, passing over the predecessors, is internal to the loop with running index j, passing over the potential successors. Therefore, when (9,10) is adjoined, the set of the successors of 10 is not known yet. An additional loop checking also such cases might further improve the algorithm, but is not necessarily justified.

The routine evaluating then the shortest path from 1 to $n+1$ uses an array SPL(i), for storing the Shortest Path Length from 1 to i. In iteration i, the values of SPL(j) for $j < i$ are already known. The algorithm now scans only those vertices and edges that remain after the pruning process.

algorithm, but when the condition does not hold for every triple (i,j,k), one can easily adapt the algorithm to deal also with these cases: replace the test **if** $(k,j) \notin E$ in the inner loop by

$$\textbf{if } (k,j) \notin E \textbf{ or } L_{kj} > L_{ki} + L_{ij}.$$

In other words, even if we can reach j from all the predecessors k of i, we still might have to add the edge (i,j) to the graph.

Figure 1 displays a small example of a graph, corresponding to the text `abbaabbabab`, including all the vertices and edges. We now assume that LZ77 is used. The edges connecting vertices i to $i+1$, for $i = 1, \ldots, n$, are labeled by the character S_i. Note that the example clearly displays the main problem with long recurring strings: if $S_i \cdots S_{j-1}$ did occur earlier, so did also all its substrings $S_k \cdots S_\ell$, for $i \le k \le \ell < j$, therefore the corresponding sub-graph with vertices $\{i, \ldots, j\}$ is a full graph. For example, the sub-graph on vertices $\{5,6,7,8,9\}$ corresponds to the second occurrence of the string `abba`, and $\{9,10,11,12\}$ corresponds to the suffix `bab`. If such recurring strings form a major part of the text, the number of edges might be $\Theta(|V|^2)$. Even if such a time complexity is still acceptable, a quadratic *space* complexity will often be prohibitive.

```
Shortest path
{
    SPL(1) ←— 0
    for i ←— 2 to n+1
        if i ∈ V then
        {
            SPL(i) ←— ∞
            for all j ∈ Pred(i)
            {
                t ←— SPL(j) + L_ji
                if SPL(i) > t then
                    SPL(i) ←— t
            }
        }
}
```

Since for each i and j for which the edge (j,i) remained in the graph, the label L_{ji} is referenced exactly once by the shortest path algorithm, its time complexity is clearly $O(|E|)$, E referring here to the dynamically built (reduced) set of edges.

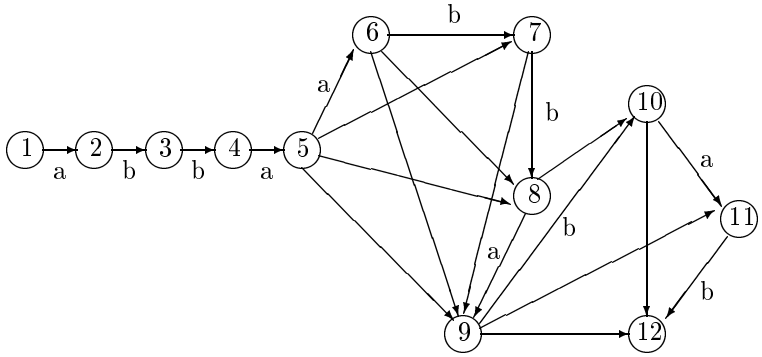

Figure 1: Original graph corresponding to text `abbaabbabab`

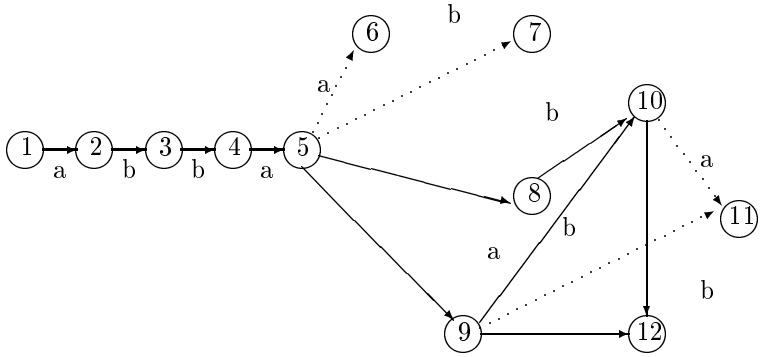

Figure 2: Graph for text `abbaabbabab` after pruning

2.3. Implementation Details

One problem in the implementation is to have an easy way to keep track of the predecessors of each vertex. The natural way to implement the graph is by keeping, for each vertex, its successor list. This requires only space $O(|E|)$, as opposed to $\Theta(|V|^2)$ for keeping the graph as an incidence matrix. One way to get access to the predecessors is to invert the successor list globally, and keep both successor and predecessor lists. But we need the predecessor list already *during* the construction of the graph, which requires a dynamic method. A possible alternative for getting the elements of $\mathrm{Pred}(i)$ is thus to check, for all values $j < i$, whether $i \in \mathrm{Succ}(j)$. This, however, requires time $\Theta(|V|^2)$ if no other bound for j is known.

If no pruning is used, every preceding vertex may be a predecessor of the current vertex i, but in fact, when scanning backwards, we may stop as soon as a vertex is found which is not connected to i, since the predecessors of a vertex must immediately precede it. When the pruning algorithm is applied, the fact that the predecessors of a vertex immediately precede it is not necessarily true. For example, in the graph of Figure 2, $\mathrm{Pred}(12) = \{9, 10\}$. Nevertheless, we show that the predecessors still form a contiguous block, so that while scanning backwards, once at least one predecessor j of i has been detected, we may continue sequentially to $j-1$, $j-2$, etc., and stop as soon as the first vertex $j-k$ is found, with $k > 0$, which is not a predecessor of i.

For the theorems below, we need to differentiate between two kinds of predecessors and successors of the vertices. There are two kinds of edges missing from the graph: those that have not been added at all (the edges which appear in Figure 1, but are missing from Figure 2), and those that have been added, but were deleted later (the dotted edges of Figure 2). Define $\mathrm{Pred'}(i)$ as the set of vertices which were predecessors of i at some stage of the algorithm, and $\mathrm{Succ'}(i)$ as the set of vertices which were successors of i at some stage of the algorithm. The sets $\mathrm{Pred}(i) = \{j \mid (j,i) \in E\}$ and $\mathrm{Succ}(i) = \{j \mid (i,j) \in E\}$ defined earlier refer to the vertices which are predecessors or successors of i even after the algorithm has completed its task. Clearly, $\mathrm{Pred}(i) \subseteq \mathrm{Pred'}(i)$ and $\mathrm{Succ}(i) \subseteq \mathrm{Succ'}(i)$.

THEOREM 1. *If pruning is used, then for each node x,* $\mathrm{Succ'}(x)$ *consists of consecutive elements, i.e.:*

$$|\mathrm{Succ'}(x)| = k \quad \rightarrow \quad \exists j \geq 0$$
$$\mathrm{Succ'}(x) = \{x+j+1, x+j+2, \ldots, x+j+k\}$$

Proof: By induction on x. For $x = 0$, $\mathrm{Succ'}(0) = \{1\}$.

Suppose the claim is true for indices smaller than x, and let y be the largest index such that $y \in \text{Succ'}(x)$ but $y-1 \notin \text{Succ'}(x)$. If $y = x+1$, then Succ'(x) is a contiguous block of the numbers immediately following x. Suppose then that $y \geq x+2$. We have to show that x is not connected to any vertex $t < y-1$. The fact that y is in Succ'(x) means that the string $S_x \cdots S_{y-1}$ appeared earlier, and thus also its prefix $S_x \cdots S_{y-2}$ appeared earlier. Therefore the reason that $y-1$ is not in Succ'(x) must be that the edge $(x, y-1)$ has not been added to the graph because of the pruning process. That is, for all $z \in \text{Pred'}(x)$, the edge $(z, y-1)$ did exist. So since for each of these z, both x and $y-1$ belong to Succ'(z), it follows from the inductive hypothesis that all elements t such that $x < t < y-1$ are also in Succ'(z). But then, by the pruning process, the edge (x,t) will not be added. ∎

THEOREM 2. *If pruning is used, then for each node $x > 0$,* Pred'(x) *consists of consecutive elements, i.e.:*

$$|\text{Pred'}(x)| = k \quad \rightarrow \quad \exists j \geq 0$$
$$\text{Pred'}(x) = \{x-j-k, \ldots, x-j-2, x-j-1\}.$$

Proof: By induction on x. For $x = 1$, Pred'$(1) = \{0\}$. Suppose the claim is true for indices smaller than x, and let y be the smallest index such that $y \in \text{Pred'}(x)$ but $y+1 \notin \text{Pred'}(x)$. If $y = x-1$, then Pred'(x) is a contiguous block of the numbers immediately preceding x. Suppose then that $y \leq x-2$. We have to show that no vertex $t \geq y+1$ is connected to x. The fact that $y \in \text{Pred'}(x)$ means that the string $S_y \cdots S_{x-1}$ appeared earlier, and thus also its suffix $S_{y+1} \cdots S_{x-1}$. Therefore the reason that $y+1$ is not in Pred'(x) must be that the edge $(y+1, x)$ has not been added to the graph because of the pruning process. That is, for all $z \in \text{Pred'}(y+1)$, the edge (z, x) did exist, hence

$$z \in \text{Pred'}(x). \tag{1}$$

But this means that both $y+1$ and x are in Succ'(z); therefore, either $y+2 = x$, in which case the proof is completed, or if $y+t < x$, then by Theorem 1, $y+t$ is also in Succ'(z), which is a set of consecutive numbers. It follows that

$$z \in \text{Pred'}(y+t). \tag{2}$$

So since all the elements in Pred'$(y+1)$ have a direct edge to $y+t$, it follows from the pruning process that the edge $(y+1, y+t)$ will not be added to the graph, i.e.,

$$y+1 \notin \text{Pred'}(y+t). \tag{3}$$

We want to show that there is no edge from $y+t$ to x. Obviously, the string $S_{y+t} \cdots S_{x-1}$ could be encoded, since it is a suffix of $S_y \cdots S_{x-1}$ which can be encoded, so we have to show that $(y+t, x)$ will not be added because of the pruning. In other words, we need to show that all elements in Pred'$(y+t)$ have an edge to x. We know this fact already for the elements of Pred'$(y+1)$. So suppose there is an element

$$z' \in \text{Pred'}(y+t) - \text{Pred'}(y+1) \tag{4}$$

which is not connected to x. Then the string $S_{z'} \cdots S_{y+t-1}$ has appeared, so did also the string $S_{z'} \cdots S_y$; thus the fact that $z' \notin \text{Pred'}(y+1)$ is due to the pruning process. That is, all $z'' \in \text{Pred'}(z')$ are connected to $y+1$, thus $z'' \in \text{Pred'}(y+1)$. But these z'' are smaller than z', hence applying the inductive hypothesis to Pred'$(y+1)$, it follows that *every* element z of Pred'$(y+1)$ is smaller than z', since $z' \notin \text{Pred'}(y+1)$.

But every element z of Pred'$(y+1)$ is also in Pred'(x) by (1) and z' is not in Pred'(x). It follows that z' is larger than all these z and that there must be an element z'', $z \leq z'' < z'$ such that $z'' \in \text{Pred'}(x)$ and $z''+1 \notin \text{Pred'}(x)$. But since y has been chosen as the smallest number having this property, we must have $z'' \geq y$ and therefore $z' \geq y+1$. But $z' \in \text{Pred'}(y+t)$ by (4) and $y+1 \notin \text{Pred'}(y+t)$ by (3), so that in fact $z' > y+1$.

We have thus found three numbers $z < y+1 < z'$ such that the first and the third belong to Pred'$(y+t)$ by (2) and (4) respectively, whereas the second does not by (3); this is a contradiction to the inductive hypothesis, by which Pred'$(y+t)$ is a set of consecutive numbers.

We conclude that there cannot possibly be an element z' as defined in (4), which is not connected to x. Therefore, all elements in Pred'$(y+t)$ are connected to x, so by the pruning process, the edge $(y+t, x)$ will not be added to the graph. ∎

The fact that the previous theorem is about Pred'(x) and not about Pred(x) is no real restriction, since anyway, the stage where the algorithm has to scan the predecessors of a vertex x is prior to the moment where any edges incident on x may be deleted.

As a consequence of Theorem 2, we can keep track of all sets Pred(i) by storing two arrays, PF$[i]$ and PL$[i]$, giving, respectively, the indices of the first and last of the predecessors of i. The inner loop on k in Algorithm Prune, which checks if all predecessors of i may be connected to j, can thus be replaced by the question whether the interval $[\text{PF}[i], \text{PL}[i]]$ is included in the interval $[\text{PF}[j], \text{PL}[j]]$, which is simply done by checking that

$$\text{PF}[i] \geq \text{PF}[j] \quad \wedge \quad \text{PL}[i] \leq \text{PL}[j].$$

Should this not be the case, the edge (i, j) will be adjoined to E, and the vectors for j will be updated: if this is the first appearance of an edge incident on j, PF$[j]$ is set to i; in any case, PL$[j]$ is set to i. Both time and space of the Prune Algorithm are thus $O(|E'|)$, where E' is the reduced set of edges.

3. ENCODING FUNCTION EXAMPLES

This section brings examples of algorithms that have been proposed and are used in commercial compression systems. We show that their encoding functions λ obey the triangle inequality, which is a sufficient condition for applying the above pruning algorithm.

The first example, based on Whiting et al. 1992,, is a variant of LZ77 known as LZSS, Storer and Szymanski 1982, using hashing on character pairs to locate (the beginning of) recurrent strings, like in Williams 1991. The output of the compression process is thus a sequence of elements, each being either a single (uncompressed) character, or an offset-length pair (d, ℓ). The elements are identified by a flag bit, so that a single character is encoded by a zero, followed by the 8-bit ASCII representation of the character, and the encoding of each (d, ℓ) pair starts with a 1. The method, referred to hereafter as Algorithm S, may easily be improved by recompression: a hashing function is not one-to-one, so that a character pair may be missed, even if it appeared earlier; even if the pair is located, the resulting matching string is not necessarily the longest possible; finally, even the longest match would not guarantee global optimality of the parsing.

The sets of possible offsets and lengths are split into classes as follows: let $B_m(n)$ denote the standard m-bit binary representation of n (with leading zeros if necessary), then, denoting the encoding scheme by λ_S:

$$\lambda_S(\text{offset } d) = \begin{cases} 1B_7(d) & \text{if } d \le 127 \\ 0B_{11}(d) & \text{if } 127 < d \le 2047 \end{cases}$$

$$\lambda_S(\text{length } \ell) = \begin{cases} B_2(\ell - 2) & \text{if } 2 \le \ell \le 4 \\ 11B_2(\ell - 5) & \text{if } 5 \le \ell \le 7 \\ (1111)^{\lceil (\ell-7)/15 \rceil} B_4((\ell - 8) \bmod 15) & \text{if } \ell \ge 8 \end{cases}$$

For example, the first few length encodings are: 00, 01, 10, 1100, 1101, 1110, 11110000, 11110001, ..., 11111110, 111111110000, etc. Including the flag-bit, each offset is thus encoded by 9 or 13 bits, and the number of bits used to encode the length ℓ is $2\lceil \frac{\ell}{4} \rceil$ for $\ell < 8$ and it is $4\lceil \frac{\ell+8}{15} \rceil$ for $\ell \ge 8$. It is of course wasteful to use an encoding of linearly growing length for the values of ℓ, but the decoding speed is enhanced, since only half-byte blocks are processed (except for $\ell \le 4$).

THEOREM 3. *The function λ_S satisfies the triangle inequality.*

Proof: Let E_1 and E_2 be two consecutive elements encoded by the algorithm, where E_i may be either a single character, or a string of characters encoded by an (offset, length) pair (d, ℓ). Denote by E the concatenation of E_1 with E_2, and assume that E may be encoded as a single element. Let $L(x)$ be the function giving the length, in bits, of the encoding $\lambda_S(x)$, where, as above, we shall apply L to both the offset or the length part, or even to an element E_i. We have to show that $L(E) \le L(E_1) + L(E_2)$.

Case 1: Both E_i are single characters, then $L(E_1) = L(E_2) = 9$. But E is a string of two characters and will be encoded by an (d, ℓ) pair. The *offset* part is encoded by at most 13 bits, the *length* part by exactly two bits (since $\ell = 2$). Thus $L(E) \le 13 + 2 < 9 + 9$.

Case 2: One of the E_i is a single character, the other a string encoded by (d, ℓ). Then there exists a d' such that E is encoded by $(d', \ell + 1)$. Thus

$$L(E_1) + L(E_2) - L(E) \ge 9 + (9 + L(\ell)) - (13 + L(\ell + 1)) \ge 1,$$

since the difference in the lengths of the encodings of consecutive lengths ℓ and $\ell + 1$ never exceeds 4 bits.

Case 3: Both E_i are strings, encoded by (d_i, ℓ_i), respectively. Then there exists a d' such that E is encoded by $(d', \ell_1 + \ell_2)$.
If both ℓ_1 and ℓ_2 are smaller than 8, then they are encoded together by at least 4 bits, but $\ell_1 + \ell_2$ is at most 14, so it is encoded by at most 8 bits. If, say, ℓ_1 is smaller than 8, but ℓ_2 is not, then they are encoded together by at least $L(\ell_2) + 2$ bits, but $\ell_1 + \ell_2$ is at most $\ell_2 + 7$, so it is encoded by at most $L(\ell_2) + 4$ bits. If both ℓ_i are larger than 7, then

$$\begin{aligned} L(\ell_1) + L(\ell_2) &\ge 4\left(\frac{\ell_1 + 8}{15} + \frac{\ell_2 + 8}{15}\right) \quad \text{and} \\ L(\ell_1 + \ell_2) &\le 4\left(\frac{\ell_1 + \ell_2 + 8}{15} + 1\right), \end{aligned}$$

so that $L(\ell_1 + \ell_2) - (L(\ell_1) + L(\ell_2)) \le 4 - \frac{32}{15} < 2$. Thus for all values of ℓ_1 and ℓ_2, $L(\ell_1 + \ell_2)$ exceeds $L(\ell_1) + L(\ell_2)$ by at most 4 bits. Therefore
$L(E_1) + L(E_2) - L(E) \ge$
$(9 + L(\ell_1)) + (9 + L(\ell_2)) - (13 + L(\ell_1 + \ell_2)) \ge 1$. ∎

The second example comes from the on-the-fly compression routine recently included in a popular operating system, and will be referred to as Algorithm M. It is again based on Williams 1991, but uses simpler hashing and a different encoding scheme λ_M. Single characters are again encoded by 9 bits, and the sets of offsets and lengths are encoded as follows:

$$\lambda_M(\text{offset } d) = \begin{cases} 1B_6(d - 1) & \text{if } 1 \le d \le 64 \\ 01B_8(d - 65) & \text{if } 64 < d \le 320 \\ 11B_{11}(d - 321) & \text{if } 320 < d \le 2368 \end{cases}$$

$$\lambda_M(\text{length } \ell) = \begin{cases} 0 & \text{if } \ell = 2 \\ 1^{j+1}\ 0\ B_j(\ell - 2 - 2^j) & \text{if } 2^j \le \ell - 2 < 2^{j+1}, \\ & \text{for } j = 0, 1, 2, \ldots \end{cases}$$

For example, the first few length encodings are: 0, 10, 1100, 1101, 111000, 111001, 111010, 111011, 11110000, etc. Offsets are thus encoded by 8, 11 or 14 bits, and the number of bits used to encode the lengths ℓ is 1 for $\ell = 2$ and $2\lceil \log_2(\ell - 1) \rceil$ for $\ell > 2$.

THEOREM 4. *The function λ_M satisfies the triangle inequality.*

Proof: We use the same notations as for Theorem 3, $L(x)$ standing now for $|\lambda_M(x)|$, and consider the same three cases.

Case 1: $L(E_1) + L(E_2) = 9 + 9 > 14 + 1 \ge L(d, 2) = L(E)$.

Case 2: $L(E_1) + L(E_2) - L(E) \ge 9 + (8 + L(\ell)) - (14 + L(\ell + 1)) \ge 1$, since the difference in the lengths of the encodings of consecutive lengths ℓ and $\ell + 1$ never exceeds 2 bits.

Case 3: For the case $\ell_1 = \ell_2 = 2$, we have $L(\text{length } 2) + L(\text{length } 2) - L(\text{length } 4) = 1 + 1 - 4 = -2$. If, say, $\ell_1 = 2$ and $\ell_2 > 2$, then we note that $L(\ell_2 + 2)$ exceeds $L(\ell_2)$ by at most 2 bits, thus $L(\text{length } 2) + L(\ell_2) - L(\ell_2 + 2) \ge 1 - 2 = -1$. If both ℓ_i are larger than 2, then using the fact that the logarithmic functions are sub-additive, we get

$$\begin{aligned} & L(\ell_1) + L(\ell_2) - L(\ell_1 + \ell_2) \\ & \quad \ge 2\left(\log_2(\ell_1 - 1) + \log_2(\ell_2 - 1) \right. \\ & \qquad \left. - (\log_2(\ell_1 + \ell_2 - 1) + 1)\right) \\ & \quad \ge 2\left(\log_2(\ell_1 + \ell_2 - 2) - \log_2(\ell_1 + \ell_2 - 1) + 1\right) \\ & \quad = 2\left(\log_2\left(1 - \tfrac{1}{\ell_1 + \ell_2 - 1}\right) + 1\right) \ge 1.3562, \end{aligned}$$

the last inequality following from the fact that $\log_2(1 - \frac{1}{n})$ is increasing with n, and we consider here values $n \ge 5$. Thus for all values of ℓ_1 and ℓ_2, $L(\ell_1 + \ell_2)$ exceeds $L(\ell_1) + L(\ell_2)$ by at most 2 bits. Therefore

$$L(E_1) + L(E_2) - L(E) \ge (8 + L(\ell_1)) + (8 + L(\ell_2)) - (14 + L(\ell_1 + \ell_2)) \ge 0. \blacksquare$$

Remark: A newer variant of the second example extends the size of the window into which a back-reference may point. The last range of $\lambda_M(\text{offset } d)$ is then encoded by $11B_{12}(d - 321)$ if $320 < d \le 4416$, so that offsets may be encoded by up to 15 bits. This, however, affects the triangle inequality. In the special case when two consecutive strings `ab` and `cd` appeared earlier at distances d_1 and d_2, both ≤ 64, but the concatenated string `abcd` appeared earlier at a distance $d_3 > 320$, encoding the string `abcd` requires $L(d_3, 4) = 15 + 4$ bits, which is larger than $L(d_1, 2) + L(d_2, 2) = (8+1) + (8+1)$ bits, needed to encode the pairs `ab` and `cd` individually. As mentioned above, the Prune algorithm may be adapted to deal with such cases too.

4. EXPERIMENTAL RESULTS

To illustrate the performance of the proposed algorithm, we have applied it to files of several types. A first class consists of texts in various natural languages. We took the first 1000 lines of the following files: for English, Finnish and German — the book of *Genesis*; for French — the *Dictionnaire Philosophique* by Voltaire; for Hebrew — the *Brahot* tractate of the Babylonian Talmud. The second set consists of non-textual files. The first four are taken from the Calgary corpus (see Bell et al. 1990): `paper1` — a paper including formatting commands; `progl` — Lisp source code; `trans` — transcript of a terminal session; `bib` — a bibliographic file. The last file in this set is `moricons.dll`, which is part of the standard MS–Windows 3.1 library.

Table 1 is a comparative chart of the compression ratios for the sample files. The column headed *Size* gives the size of the file in bytes. The next four columns give the relative size of the compressed file, expressed as a percentage of the full file, for Algorithms M and S, the longest match heuristic (LMH) and the optimal algorithm, respectively. Recall that we deal here with optimal parsing relative to a given coding function; the method used for the optimal parsing here is the one of Algorithm M. The last column contains the compression results obtained by `gzip`, tuned for maximal compression, and has been added for comparison purposes only, since `gzip` uses another encoding scheme and is thus not a recompression technique in the sense used in this paper.

For all five algorithms, the files were processed by clusters of 8K bytes, as is done in the commercial systems. The values for Algorithms M and S were obtained by simulating their hashing, parsing and coding, rather than by applying the commercial software packages. The latter report only approximate results, and have some additional overhead, so that the simulated values are better by a few percent. This also explains that the ratios for Algorithm M consistently improve those for Algorithm S, whereas the performance of the commercial counterparts are generally roughly equivalent. On our examples, recompression by the longest match heuristic reduced the file sizes by 17–27% relative to the compression by Algorithm M, so the recompression savings on the commercial systems would be even larger. The optimal parsing then saves an additional 3–6%. With `gzip`, compression could still be improved by 10–15%, except for the moricons file, for which the

File	Size	Alg. M	Alg. S	LMH	Optimal	`gzip`
English	36521	54.9	58.2	42.3	39.9	35.9
French	53580	64.8	71.8	51.1	48.1	43.0
Finnish	34615	59.6	64.9	45.1	42.6	38.3
German	55121	59.6	64.5	47.0	44.5	40.4
Hebrew	48193	62.1	69.9	51.2	48.6	43.9
paper1	53161	64.0	68.6	49.3	47.0	41.8
progl	71646	45.4	46.5	33.1	31.4	27.0
trans	93695	49.0	50.5	39.0	37.5	31.7
bib	111261	65.6	67.9	51.6	49.1	42.8
moricons	118864	34.3	38.6	26.7	25.8	25.6

Table 1: *Comparison of compression ratios on sample files*

improvement was only 0.8%.

File	Time		Space	
	Opt	Prune	Opt	Prune
English	0.75	0.39	220357	28.9
French	1.55	1.12	222270	47.9
Finnish	1.22	0.47	203772	29.3
German	2.15	1.07	282662	32.9
Hebrew	1.50	1.05	235260	33.7
paper1	1.33	0.76	287574	29.4
progl	3.05	0.94	1187012	10.7
trans	3.94	0.83	2031230	7.2
bib	2.55	1.52	626136	25.2
moricons	10.02	1.57	5320050	16.8

Table 2: *Comparison of time and internal space*

Table 2 lists some time and space measurements. The algorithms were run on a Sparc 10. The values correspond to the time, in seconds, to find the optimal path in the full graph (Opt) and in the reduced graph (Prune). The algorithm for the full graph is generally 1.5 to 3 times slower than on the pruned graph, and more than 6 times in the extreme case on the `moricons` file. Moreover, the Prune algorithm drastically reduces the internal memory requirements. The column for Space headed Opt gives the number of edges in the full graph; the last column gives the percentage of the edges remaining in the pruned graph. Note that while the graphs have similar sizes for the textual files, they are much larger for the others, where long strings reoccur frequently. For example, even though `progl` is only 35% larger than `paper1`, it generates more than four times as many edges. This is because `progl` contains many long runs of semicolons, and frequent long variable names.

5. CONCLUSION

The new technique improves the time and space requirements of an algorithm that is optimal under the constraint of a given encoding function, and may thus efficiently enhance many of the dictionary based encoding methods that have been suggested. Our experimental results show an additional 20%–30% savings obtained by the optimal recompression algorithm, relative to the on-the-fly methods. A large part of these savings is due to simple improvements of the parsing strategy, but 3–6% can be attributed to the use of the optimal method instead of the longest match heuristic. However, even a minor improvement might be worthwhile in certain applications, and has certainly theoretical value, since one can show that it cannot be further improved under our constraints of a predetermined decoding procedure. But because of its complexity, the optimal technique was often not considered worth the effort for the modest improvement it yielded. The contribution of this paper is the reduction of the time and space requirements of the optimal algorithm. The Prune algorithm thus permits, at low cost in time and space, to replace sub-optimal compression heuristics by an optimal method.

REFERENCES

[1] BELL T.C., CLEARY J.G., WITTEN I.A. (1990), *Text Compression*, Prentice Hall, Englewood Cliffs, NJ.

[2] BELL T.C., MOFFAT A., NEVILL-MANNING C.G., WITTEN I.H., ZOBEL J. (1993), Data compression in full-text retrieval systems, *J. Amer. Soc. of Inf. Sci.* **44** 508–531.

[3] BELL T.C. (1986), Better OPM/L text compression, *IEEE Trans. on Communications* **COM–34** 1176–1182.

[4] BRENT R.P. (1987), A linear algorithm for data compression, *The Australian Computer Journal* **19** 64–68.

[5] CHOUEKA Y., FRAENKEL A.S., KLEIN S.T. (1988), Compression of Concordances in Full-Text Retrieval Systems, *Proc. 11-th ACM-SIGIR Conf.,* Grenoble, 597–612.

[6] CHOUEKA Y., FRAENKEL A.S., KLEIN S.T., SEGAL E. (1986), Improved Hierarchical Bit-Vector Compression in Document Retrieval Systems, *Proc. 9-th ACM-SIGIR Conf.,* Pisa, 88–97.

[7] CORMEN T.H., LEISERSON C.E., RIVEST R.L., *Introduction to Algorithms (1990),* MIT Press, Cambridge, MA.

[8] DIJKSTRA E.W. (1959), A note on two problems in connexion with graphs, *Numerische Mathematik* **1** 269–271.

[9] EVEN S. (1979), *Graph Algorithms*, Computer Science Press.

[10] FIALA E.R., GREENE D.H. (1989), Data compression with finite windows, *Comm. of the ACM* **32** 490–505.

[11] HIRSCHBERG D.S., STAUFFER L.M. (1994), Parsing algorithms for dictionary compression on the PRAM, *Proc. Data Compression Conference DCC–94*, Snowbird, Utah, 136–145.

[12] HORSPOOL R.N. (1995), The effect of non-greedy parsing in Ziv-Lempel compression methods, *Proc. Data Compression Conference DCC–95*, Snowbird, Utah, 302–311.

[13] HUFFMAN D. (1952), A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** 1098–1101.

[14] LELEWER D.A., HIRSCHBERG D.S. (1987), Data compression, *ACM Computing Surveys* **19** 261–296.

[15] KATAJAINEN J., RAITA T. (1989), An approximation algorithm for space-optimal encoding of a text, *The Computer Journal* **32** 228–237.

[16] KATAJAINEN J., RAITA T. (1992), An analysis of the longest match and the greedy heuristics in text encoding, *J. ACM* **39** 281–294.

[17] KLEIN S.T., BOOKSTEIN A., DEERWESTER S. (1989), Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** 230–245.

[18] SCHUEGRAF E.J., HEAPS H.S. (1974), A comparison of algorithms for data base compression by use of fragments as language elements, *Information Stor. and Retr.* **10** 309–319.

[19] STORER J.A. (1988), *Data Compression: Methods and Theory,* Computer Science Press, Rockville, Maryland.

[20] STORER J.A., SZYMANSKI, T.G. (1982), Data compression via textual substitution, *J. ACM* **29** 928–951.

[21] WAGNER R.A. (1973), Common phrases and minimum-space text storage, *Comm. of the ACM,* **16** 148–152.

[22] WHITING D.L., GEORGE G.A., IVEY G.E. (1992), Data Compression Apparatus and Method, U.S. Patent 5,126,739.

[23] WILLIAMS R.N. (1991), An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah, 362–371.

[24] WITTEN I.H, NEAL R.M., CLEARY J.G. (1987), Arithmetic coding for data compression, *Comm. ACM* **30** 520–540.

[25] ZIV J., LEMPEL A. (1977), A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** 337–343.

[26] ZIV J., LEMPEL A. (1978), Compression of individual sequences via variable-rate coding, *IEEE Trans. on Inf. Th.* **IT–24** 530–536.