A LINEAR ALGORITHM FOR FINDING REPETITIONS
AND ITS APPLICATIONS IN DATA COMPRESSION

by

M. Rodeh*, V. R. Pratt** and S. Even*

Technical Report #72
April 1976

* Dept. of Computer Science, Technion, Haifa, Israel.
** Dept. of Electrical Engineering and Computer Science, Mass. Inst. of Tech.
    Cambridge, Mass. 02139.
ABSTRACT

A linear algorithm for finding the longest repetition for each position in a given string is developed. It is related to Weiner's string processing algorithm [1] but simpler. The algorithm is applied to the problem of evaluating the complexity of a finite string as defined by Lempel and Ziv [2]. Using Elias' representation of the integers [4,5] we produce an asymptotically optimal variable to variable coding scheme for sequential data compression based on the definition of complexity. The scheme requires unbounded memory for strings whose length grows to infinity. To overcome this difficulty Ziv and Lempel [3] invented a variable to block coding scheme that is asymptotically optimal for all ergodic sources with a given finite entropy. We implement their scheme by combining our repetition finder with Weiner's suffix tree construction algorithm. Other related and more practical schemes are described too.
0.1. Introduction.

This paper comprises 3 sections. In Section 1 we throw new light on Welner's string processing algorithm [1] by developing a related algorithm for finding repetitions in a string. In Section 2 we adapt it to constitute a linear implementation of a variable-to-variable data compression scheme based on Lempel and Ziv's [2] definition of the complexity of a finite sequence; this scheme uses unbounded memory. A linear implementation of Ziv and Lempel's [3] variable-to-block coding scheme with bounded memory is described in Section 3; it is a combination of our repetition finder and Welner's algorithm [1] for suffix tree construction.

0.2. Notation.

Let $\Sigma$ be a finite alphabet and $\Sigma^*$ the set of all finite length strings over $\Sigma$. We use the letters $a, b, c, \ldots$ for elements of $\Sigma$ and $\alpha, \beta, \gamma, \ldots$ for strings. We let $\varepsilon$ denote the null string, $l(\alpha)$ the length of $\alpha$, $\alpha(1)$ the $1$-th letter of $\alpha$, and $\alpha \beta$ the concatenation of $\alpha$ and $\beta$.

Let position in a string denote "between-letter" rather than "at-letter" position. Thus $\alpha$ has $l(\alpha)+1$ positions numbered $0, 1, \ldots, l(\alpha)$. Two positions $i$ and $j$ of $\alpha$ define the subword $\alpha(i+1) \ldots \alpha(j-1)\alpha(j)$ which occurs at $j$ and is denoted by $\alpha(i, j)$;
If $1 \leq j$ then $a^{<1,j>}$ is defined to be $\epsilon$. $a^{<0,1>}$ for $0 \leq l \leq z(a)$ is called a prefix of $a$; $a^{<1,z(a)>}$ is called a suffix of $a$.

A repetition at $j$ is an occurrence at $j$ of a subword that also occurs at $1 < j$; thus, every occurrence but the first of a subword is a repetition.

An occurrence of $aa$ at $1$ is the left neighbor of the occurrence of $a$ at $1$, and the occurrence of $ab$ at $1+1$ is the right neighbor.

1. Improvements to Weiner's Algorithm.

1.1. Introduction.

In this section we develop an algorithm for finding repetitions within a string. It is related to Weiner's algorithm but is simpler. We also show that the running time of the algorithm on a RAM can be made independent of the alphabet size. To this end we establish properties of strings which prove the correctness of the algorithm.

1.2. Weiner's Suffix Identifier Finder.

Let $a$ be a fixed string of length $n$. The suffix identifier of position $1$ is the shortest (unique) subword which occurs only at $1$. To make this definition meaningful we have to add a new alphabet letter $\$$ to the left of $a$. Weiner's algorithm
finds all suffix identifiers in $O(n)$ time. (Weiner actually considered prefix identifiers, but the dual case is closer to our algorithm.)

We are interested in repetitions, which is a related concept. Out of the $(n+1)+1$ subwords of $\alpha$, $O(n^2)$ different subwords may be repetitions. Thus in aiming at a linear repetition-finding algorithm we cannot afford to identify every repetition explicitly. The algorithm finds only the longest repetition at each position.

Observe that the longest repetition at position $i$ depends only on $\alpha(0,i)$ and on nothing to the right of $i$. Thus in the production of longest repetitions in a left-to-right scan, we never have to change the repetitions already found. This is a significant property since in our application we are interested in the scanning process itself and not in the final result. This is in contrast to the situation with the suffix identifiers which depend on the entire $\alpha$. Thus in Weiner's algorithm when scanning $\alpha$ from left-to-right, some subword which occurs at $j>i$ might cause an update operation which includes the suffix identifier for $i$, though this does not compromise its $O(n)$ time bound.

The suffix identifier for $i$ is longer than the longest repetition at $i$. However, scanning the string left-to-right, when we reach position $i$, they differ only by the leftmost letter of the suffix identifier which does not appear in the
longest repetition.

It is worth noting here that neither Weiner's nor our algorithm proceeds from position $i$ to position $i+1$ at constant time although both operate within a time bound linear in $n$.

1.3. An Algorithm for Finding Repetitions.

As pointed out above, we are interested in longest repetitions only. For example, in $abcabcbe$, the longest
repetitions at each position 0 through 8 respectively are \( \varepsilon, \varepsilon, \varepsilon, \varepsilon, a, ab, abc, b, bc \). We shall always use *longest repetition* to mean "longest repetition at a given position".

Notice that in this example later longest repetitions do not start in earlier positions. This is always true:

**Theorem 1:** Let \( \beta, \gamma \) be longest repetitions at \( l, j \) respectively. If \( l < j \) (i.e. if \( \beta \) ends before \( \gamma \)) then \( \gamma \) does not start before \( \beta \).

**Proof:** Any prefix of this occurrence of \( \gamma \) must be a repetition. Thus if \( \gamma \) started before \( \beta \), \( \beta \) could not be a longest repetition.

This theorem suggests the basic algorithm, which simply moves two pointers (i.e. increments two counters) through \( \alpha \). We represent the pointers as brackets and call the in-between subword *bracketed*.

**Algorithm 1.1:**

start with \([]\alpha;\)

while the picture is \( ...[\beta]\alpha... \) do

if \( \beta \alpha \) is not a repetition then move \( [\]

else move \( ] \).

"move [" and "move ]" mean move the indicated bracket one position to the right. If \( \varepsilon \) is bracketed, "move [" also implies "move ]" to avoid having "[" overtake "]".

Notice that when "]" first arrives at a position, the
One question remains: How do we tell efficiently whether $\beta a$ is a repetition?

1.3. Testing for repetitions.

A remarkable property of longest repetitions supplies the key to testing whether $\beta a$ is a repetition.

**Lemma 1:** If no occurrence of $\beta$ in a string is a longest repetition then every occurrence of $\beta$ has the same left neighbor.

**Proof:** We use induction on the number of occurrences of $\beta$ in the string. The Lemma is vacuously true for zero or one occurrence. If there are two or more, the rightmost occurrence must have the same left neighbor as some other occurrence of $\beta$, since it is not a longest repetition. By induction all occurrences have the same left neighbor. $\Box$

Putting the Lemma in its contrapositive form we have:

**Lemma 1':** Let $\beta$ occur at $i$ and at $j$, $i < j$, not with the same left neighbor. Then there exists $k < j$ such that $\beta$ is the longest repetition at $k$.

**Proof:** Apply Lemma 1 to the first $j$ characters of the string. $\Box$

**Theorem 2:** If $\beta a$ is a longest repetition at $j$, there is some $k < j$ such that $\beta$ is the longest repetition at $k$. 

bracketed subword is the longest repetition for that position.
Proof: Let the left neighbor of the occurrence of $\beta a$ at $j$ be $b\beta a$. There must be an earlier occurrence of $\beta a$ (since the one at $j$ is a repetition) whose left neighbor is not $b\beta a$ (since the occurrence of $\beta a$ at $j$ is a longest repetition). Then $\beta$ satisfies the conditions of Lemma 1', yielding the desired $k$. 

In order to test whether $\beta a$ is a repetition (and therefore a longest repetition by the correctness of Algorithm 1,1), we ask whether $\beta$ is a previously recorded longest repetition, and if so whether some earlier occurrence of $\beta$ has right neighbor $\beta a$. To answer the first question we keep a record of longest repetitions seen so far, represented as vertices of a graph $G$. Each vertex corresponds to a distinct element $\beta$ of $\Sigma^*$, and so represents all repetitions of a given subword $\beta$. In our discussion we refer to a vertex by underlining the string which it represents. To answer the second question, we associate with each vertex $\beta$ a record for each letter $a$ of whether $\beta a$ has occurred, and if so where it first occurred. We let $\beta.a$ name a register associated with $\beta$, which we may inspect and update. If $\beta.a$ is undefined, $\beta a$ has not occurred before; if defined, $\beta a$ first occurs at $\beta.a$. (The single dot suggests the first occurrence.) There are $|\Sigma|$ such registers associated with each vertex, all initially undefined.

We postpone for the moment the question of how we keep this information about right neighbors up to date, as it depends on the details of how we navigate in and update the
graph \( G \).

In order to locate \( \beta \) when asking whether \( \beta \alpha \) has occurred before, we shall maintain a pointer to the vertex in \( G \) representing the currently bracketed subword. Clearly we need enough information in \( G \) to allow us to trace a trajectory through \( G \) corresponding to the sequence of bracketed words resulting from moving the brackets.

By Theorem 2, Algorithm 1.1 must keep moving "[" until some word is bracketed for which there is a vertex in \( G \). Hence for each vertex \( \theta \) in \( G \), all we need is an edge (which we label \( S \) for suffix) from \( \theta \) to the vertex which represents the longest proper suffix of \( \beta \) in \( G \). We denote by \( S(\theta) \) a register associated with \( \theta \) containing this edge (i.e. pointer). In this way, we may jump directly from \( \theta \) to \( S(\theta) \), following possibly several moves of "]".

When we move "]", we go from \( \beta \) to \( \beta \alpha \). If \( \beta \alpha \) is already in \( G \), an edge (labelled \( :a \)) from \( \theta \) to \( \beta \alpha \) suffices; for each letter \( a \) we associate with \( \theta \) a register named \( \theta :a \) to contain \( \beta \alpha \). (The two dots suggests that \( \beta \alpha \) has occurred at least twice). If \( \beta \alpha \) is not yet in \( G \), we must create it. This is sufficiently complicated to warrant its own section.

Except for the details of vertex creation, and of keeping \( \theta :a \) up to date for all \( a \) and for all \( \beta \), we may now flesh out Algorithm 1.1 to yield Algorithm 1.2.

In the following, \( \beta \) denotes the currently bracketed
subword and its position (thus \( i \) represents "\["]"). The function \( \text{GENERATE} \) returns a new vertex, with all associated registers undefined. Note that we do not have to represent "\["] explicitly because \( \beta \) keeps track of it implicitly.

**Algorithm 1.2:**

Initialize 
\[ \beta \leftarrow \text{GENERATE}; \]
\[ \beta \leftarrow \xi; i+1; \]
Iterate 
\[ \text{while } l < n \text{ do} \]
\[ \begin{align*}
\text{begin } & a+a(<i>); \\
\text{Repetition?} & \text{ if } \beta.a \text{ undefined then} \\
\text{Move [} & \text{ if } \beta=x \text{ then } \beta+\mathcal{S}(\beta) \\
\text{or []} & \text{ else } i+1+1 \\
\text{Create?} & \text{ else begin if } \beta:a \text{ undefined then } \text{CREATEVERTEX; } \\
& \beta+\beta:a; i+1+1 \\
& \text{ end} \\
\text{end; } \\
\end{align*} \]

We turn now to vertex creation, and postpone the details of updating right neighbors for the moment.

**1.5. Creating Vertices.**

Merely generating a new vertex \( \beta a \) is easy, as is updating \( \beta:a \). It is more difficult to correctly update the \( S \) edges leading to and from \( \beta a \). By definition, only one edge leads from \( \beta a \). Fortunately, when it is created, at most one edge leads to
8α, as Corollary 3.2 shows.

**Theorem 3:** Given a and τ, there exists at most one subword π such that πατ is in G and S(πατ)=I.

**Proof:** Let πατ and π'ατ be two such distinct subwords, both represented in G. Neither may be a suffix of the other, since S(πατ)=S(π'ατ)=I. Hence πατ and π'ατ must be of the forms πηρατ and π'ηρατ (f=g) respectively where any of η, η' and ρ may be null. Thus by lemma 1, πατ occurs as a longest repetition no later than the later occurrence of πατ and π'ατ, contradicting S(πηρατ)=I.

**Corollary 3.1:** For each letter c and vertex β we may introduce an edge (register) labelled *c: from β to the (necessarily unique) vertex γ of the form ψcβ such that S(γ)=β (The "*" denotes the ψ in ψcβ). Thus S(*c:β)=β, provided *c:β is defined.

**Corollary 3.2:** When creating a vertex βα, at most one vertex γ exists such that S(γ)=βα.

**Proof:** By Lemma 1 all previous occurrences of βα must have the same left neighbor bβα, if any. Hence, if βα were a suffix of a bracketed subword, it should have bβα as left neighbor and by Theorem 3, there exists at most one vertex γ such that S(γ)=βα.

Hence our problem is reduced to that of finding the γ which is S(βα), and the δ, if any, for which S(δ)=βα, and
updating all the edges between \( \hat{a}, \beta a \) and \( \gamma \).

Either \( \gamma = \epsilon \) or \( \gamma = \gamma' a \), for some \( \gamma' \). If the latter, then by Theorem 2, since \( \gamma \) is in \( G \), so is \( \gamma' \). To find \( \gamma' \) we simply follow along \( S \) edges, starting from \( \hat{a} \), until we find a vertex \( \pi \) for which \( \pi : a \) is defined; \( \pi : a \) is the required \( \gamma \). If there is no such \( \pi \) (the search ends with \( \pi = \epsilon \) and \( \epsilon : a \) is undefined) then we are in the first case and \( \gamma = \epsilon \).

To find \( \delta \), note that before \( \beta a \) is created, \( S(\delta) = \gamma \). If \( \beta a = n c \gamma \) for some \( n \) and \( c \), then \( \delta = * c : \gamma \). To see that \( * c : \gamma \) must yield a \( \delta \) of which \( \beta a \) is a suffix, note that if it didn't, then \( * c : \gamma \) would have to point to both \( \beta a \) and \( \delta \), contrary to Corollary 3.1.

So we wish to compute \( c \). In the absence of a representation of \( c \) in \( G \), we look in \( a \). For this purpose we introduce two registers, for each vertex \( \hat{a} \); \( \text{loc}(\hat{a}) \) gives the location of the first occurrence of \( \beta \) and \( \text{len}(\hat{a}) \) holds the length of \( \beta \).

Now we can easily find \( c \). Let \( j = \text{loc}(\beta a) - \text{len}(\gamma) \); then \( c = \alpha < j > \). When setting \( * d : \beta a \) to \( \delta \) we find the \( d \) in the same way except that \( j = \text{loc}(\delta) - \text{len}(\beta a) \), \( d = \alpha < j > \).

Let us review all the registers associated with each vertex \( \hat{a} \) used in \( G \).

\( \hat{a} : a \) = the position of the earliest occurrence of \( \beta a \).
\( \hat{a} : a \) = a pointer to the vertex \( \beta a \) in \( G \).
\( S(\hat{a}) \) = a pointer to the longest proper suffix of \( \beta \).
represented in G.

*a:A = a pointer to the A for which δ=aB and S(A)=A.
loc(A) = the position of the first occurrence of B.
len(A) = the length of B.

The remaining question is that of updating right neighbors of subwords represented in G.

1.6. Updating Right Neighbors.

For the algorithm's correctness it is enough to show a way to update B's right neighbors no later than the first occurrence of [B]a. We keep B's right neighbors up to date from the moment of creating A. This is useful in certain applications.

First assume that the earliest occurrence of B_a is later than the earliest bracketed occurrence of B. Then when ] first arrives between B and a, [ must be to the left of this occurrence of B. Now ] may not move while [ is to the left of B since this is the first occurrence of B_a. But in the process of moving to the right, ] must come to bracket B since A is already in G. At this time we are in a position to define A.a, which can be done with A.a+1. By only performing this assignment when A.a is undefined, we avoid erroneously redefining A.a at a non-initial occurrence of B_a.

Now assume that the first occurrence of B_a is not later
than the first bracketed occurrence of $\beta$. We have to show that when $\beta$ is created we can immediately record the position of the first $\beta a$.

Two cases arise:

(I) The second occurrence of $\beta$ is bracketed. In this case, only one right neighbor is involved, namely that of the first occurrence of $\beta$. Thus when $\beta$ is first bracketed, $\text{loc}(\beta)$ may be used to determine this right neighbor. In this case there exists no $\delta$ such that $S(\delta) = \beta$, since the presence of such a $\delta$ implies at least two earlier occurrences of $\beta$.

(II) The third or later occurrence of $\beta$ is the first to be bracketed. In this case, there must be some $\delta$ such that $S(\delta) = \beta$, because the second occurrence of $\beta$ can avoid being bracketed only by not being the longest repetition at that position. In this case, there may be several right neighbors to be recorded; however, every previous occurrence of $\beta$ must be at the same position as some occurrence of $\delta$, and since all of $\delta$'s right neighbors are up to date (by induction), we can simply copy their positions, which will then serve as the positions of the corresponding right neighbors of $\beta$.

Now we collect all the pieces to yield the following final algorithm.
1.7. The Algorithm in Full Detail.

Algorithm 1.3

Initialize $e$ $\leftarrow$ GENERATE;
len($e$) $\leftarrow$ 0;
loc($e$) $\leftarrow$ 0;
S($e$) $\leftarrow$ $e$;
$\delta$ $\leftarrow$ $e$;
l $\leftarrow$ 1;

Iterate while l $\leq$ n do
begin
$a$ $\leftarrow$ $\langle l \rangle$;
Repetition? if $\delta$.a undefined then
New rt. nbr begin $\delta$.a+1;
Move [ if $\delta$.e then $\delta$.S($\delta$)
or [] else l $\leftarrow$ l+1
end
Create? else begin if $\delta$.a undefined then CREATEVERTEX;
Move ] $\delta$.a $\leftarrow$ $\delta$.a;
l $\leftarrow$ l+1
end;
end;
CREATEVERTEX: \( \text{Ba}+\text{GENERATE}; \)

\[
\text{len(Ba)}+\text{len(\text{a})}+1;
\]

\[
\text{loc(Ba)}+\text{a}.
\]

\[
\text{Link } \text{a,Ba } \text{a}+\text{Ba};
\]

\[
\text{Find } x \quad x'+\text{S(\text{a})}; \quad \text{comment: we need } S(\text{a})=\text{a} \text{ here;}
\]

\[
\text{while } x':a \text{ undefined and } x'\neq\text{a} \text{ do } x'+\text{S(x')};
\]

\[
x'+\text{if } x':a \text{ defined and } \text{a}\neq\text{a} \text{ then } x':a \text{ else } \text{a};
\]

\[
\text{Find } \delta \quad j+\text{loc(Ba)}-\text{len(x)};\text{a}
\]

\[
c+\text{a}(j); \text{end}
\]

\[
\delta+*c:x;
\]

\[
\text{Link } \text{a,x } \text{S(Ba)}+x;
\]

\[
*c:x+\text{Ba};
\]

\[
\text{if } \delta \text{ defined then begin}
\]

\[
\text{Link } \delta,\text{Ba } \quad j+\text{loc(\delta)}-\text{len(Ba)};\text{a}
\]

\[
d+\text{a}(j); \text{end}
\]

\[
\text{S(\delta)}+\text{Ba}; \text{end};
\]

\[
\text{Rt. nbrs } \text{if } \delta \text{ undefined then } \text{Ba.a}(\text{a}+1)+\text{a}+1
\]

\[
\text{else for each letter } \text{b do }
\]

\[
\text{Ba.b}+\text{b.b}
\]

end;
1.8. Timing.

We wish to show that the running time of the algorithm is linear in \( n=\ell(\alpha) \), independently of the alphabet size. The argument proceeds in two stages: At first we assume that the alphabet is fixed and finite. Then we consider the influence of the alphabet size, \(|\Sigma|\).

The main routine moves one or both brackets to the right at each iteration of the single loop. Hence the body of that loop may not be executed more than \( 2n \) times.

CREATEVERTEX contains two loops. The second, which transcribes right neighbors is executed \(|\Sigma|\) times, thus constant. The first loop searches for \( \gamma' \). The search is done by moving from \( \beta \) along the \( S \) arcs. We introduce a third pointer, in addition to the two brackets; it points to the starting position of the occurrence of \( \gamma' \) at \( l \), during this search. We claim that this pointer never moves left through \( \alpha \), even between successive vertex creations, and moves right at every iteration of the search loop. The latter claim is non-controversial. To see the former, observe that when the search that started from \( \beta \) terminates, a search that starts from \( \beta a \) must begin with the just-found \( \gamma \). A similar observation applies even when the next search does not start from \( \beta a \). This establishes the linear time bound for a finite alphabet.

Now consider an unbounded alphabet. The right neighbor transcription loop then becomes a potential time-waster. We
shall show that the cost of copying all defined right neighbors but one can be charged to the account of moving "[" through \( a \).

Let us assume that the algorithm is at position \( i \) and transcribes the right neighbors of \( \alpha a \) from \( \delta \). For every \( c \) such that \( \delta . c \) is defined, let \( k(c)+1 \) be the first occurrence of \( \beta ac \). We divide the cost of copying \( \delta \)'s right neighbors into two cases:

(i) \( c \) is the letter for which \( k(c) \) is minimum, i.e. \( k(c)=\text{loc}(\alpha a) \). The cost of transcribing \( \delta . c \) is charged to \( \alpha a \), and since the number of vertices is bounded by \( n \), so is the total cost of these transcriptions.

(ii) For any other letter \( d \), consider the moment when "[" is first in position \( k(d) \). Since \( \beta a \), at that time, is already a repetition (but not a longest one) "[" must be to the left of \( \beta a \) (but not adjacent to it). However "[" cannot move to the right before "[" enters \( \beta a \) since \( \beta ad \) is not a repetition at \( k(d)+1 \). Also, "[" cannot be located at \( k(d)-\ell(\beta a) \), since \( \beta a \) is not defined yet. Thus, "[" makes a jump over location \( k(d)-\ell(\beta a) \). We charge the transcription of \( \delta . d \) (to vertex \( \beta a \)) to the jump over this location.

To take advantage of this observation we must devise a method of ignoring undefined right neighbors. This is easily done by maintaining for each vertex \( \delta a \) a list of letters such that for each letter \( b \) in the list \( \beta \beta b \) is defined. It is easy
to see that a duplication can never exist in such a list. To access each letter in constant time we need an array of length $|\Sigma|$ for each vertex $\&$ and each of the tables which contain $\&.a$, $\&:a$ and $*a:\&$. Thus, if we have a storage preset to zero, we achieve an independence of the alphabet size. Instead we can simulate the storage preset to zero in real time ([7], p.71, Ex.2.12). Unfortunately this can be very wasteful of space; in fact the factor of $|\Sigma|$ has simply crossed over to space bound. In order to keep the space requirement independent of $|\Sigma|$ we have to pay a time overhead of at most $\log|\Sigma|$.
2.1. Introduction.

In this section we present a linear scheme for sequential data compression in two steps:

**Step 1:** We design an algorithm for translating strings to words over a certain infinite alphabet \( \Theta \). The translation method is based on Lempel and Ziv's definition of the complexity of finite strings [2]. The implementation uses the repetition finding algorithm. The handling of the data structures is irrelevant here. Therefore we shall refer to Algorithm 1.1 and not to the full detailed version. The linearity will come out as an immediate consequence of the timing considerations of the repetition finding algorithm.

**Step 2:** We use Elias' universal binary* representation of the integers [4,5] to produce a binary encoding of strings over \( \Theta \). For a given integer \( m \) this representation can be built within a time bound linear in the length of the binary representation of \( m \).

The composition of the two steps yields a universal linear

*The generalization to an arbitrary finite alphabet is straightforward.
variable to variable encoding scheme for strings, whose compression ratio tends to be optimal, for ergodic sources, as the length of the input string grows to infinity. We shall prove this by referring to results of Lempel and Ziv [3]. The disadvantages of the scheme are that the memory requirements grow with the length of the input string and the rate of convergence to the optimal compression ratio is slow. We shall treat the memory problem in Section 3.

2.2. Translating strings to $\Theta^*$. Let $\Theta=\{<i,j,a> | i,j \text{ are integers, } a \in \Sigma\}$. The letters of $\Theta$ are triples whose first two components are integers, and the third is a letter of $\Sigma$. A word over $\Theta$ is a sequence of such triples.

Define the translation of $\alpha$ to $\Theta^*$, denoted by $T(\alpha)$, by the following algorithm:

**Algorithm 2.1.**

$p=0; k=0;$

while $p < \ell(\alpha)$ do

begin $m=$ maximum integer for which there exists a $j<p$

satisfying $\alpha_{j,j+m}=\alpha_{p,p+m}$ and $p+m<\ell(\alpha);$

1+some $j$ satisfying the above;

$k+k+1; p+p+m+1;$

$t_k=<i,m,\alpha_{p}>$

end;
It can easily be seen that $T(\alpha)$ exists and is well defined. Lempel and Ziv [2] define $\ell(T(\alpha))$, the number of triples in the translation of $\alpha$, which is the same as the final value of $k$ in Algorithm 2.1, to be the **complexity of** $\alpha$; denote it by $C(\alpha)$.

Assume that we have translated $\alpha_{0,p}$ into $t_1\ldots t_k$, and computed $t_{k+1} = \langle l, m, \alpha_{p+m+1} \rangle$. Thus $\alpha_{p,p+m}$ is a repetition at $p+m$. If $p+m < \ell(\alpha)$ then $\alpha_{p,p+m+1}$ is not a repetition. In terms of the brackets movements of Algorithm 1.1 we may say that when "]" first arrives at $p+m"["$ is not to the right of $p$ while when "]" is at $p+m+1"["$ is to the right of $p$. The following algorithm implements the ideas above:

We let $p_{[,.]}$ denote the current position of $[\,$ and $\,$ respectively; $\text{loc}(\beta)$ the location of the first occurrence of $\beta$, and $p$ the position immediately after the current translated prefix of $\alpha$. 
Algorithm 2.2.
k+0; p+0;
start with \([\alpha]\);
while the picture is \([\beta]\)a... do
begin if \(\beta a\) is a repetition then move \) else
  if \(\beta \neq \varepsilon\) then begin j+p_j-p;
    l+loc(\(\beta\))-j;
    move [;
  end
  if \(p_j>p\) then begin
    k+k+1;
    t_k+<l,j,a>;
    p+p_j+1
  end
else begin if \(p_j=p\) then begin k+k+1;
    t_k+<0,0,a>;
    p+p+1
  end;
move [ end
end;
if \(p<\ell(\alpha)\) then begin j+\ell(\alpha)-p-1;
  l+loc(\(\beta\))-j;
  k+k+1;
  t_k+<l,j,a<\ell(\alpha)>>
end;
It is easy to see that the movements of "[" and "]" on are identical to those in Algorithm 1.1. The production of $T(a)$ is added to the identification of longest repetitions and does not change the flow of the repetition finding algorithm. To each iteration of the single loop, we add a constant number of operations. Therefore, Algorithm 2.2 is as linear as Algorithm 1.1. We can say a little more: The production cost of each triple of $T(a)$ is linear in the length of the substring which it represents. We refer to the movements of the three pointers introduced in Section 1. Denote two consecutive substrings of $a$ each of which is translated into a single letter of $\Theta$, by $\gamma'$ and $\gamma$. While "]" moves along $\gamma$, "[" moves along $\gamma'$. The third pointer is bounded between these two, and thus can move no more than $l(\gamma')l(\gamma)$ steps. Thus, the total cost is bounded by $O(l(\gamma')l(\gamma))$. The contribution of $l(\gamma')$ may be charged to the translation of $\gamma'$. Thus the cost of translating $\gamma$ into a single triple is $O(l(\gamma))$.

2.3. Encoding of Words Over $\Theta$.

For simplicity we assume that $\Sigma = \{0, 1\}$. Let $\tau = t_1 t_2 \ldots t_k$ be a word over $\Theta$. Each $t_q$ has the form $<i, j, a>$. We use Elias' representation of the integers to encode $i$ and $j$. The code of $t_q$ is formed by the concatenation of these codes and $a$. We get a uniquely decipherable binary encoding of $\tau$.

Let $B(n)$ denote that binary representation of $n$, for which
the most significant bit is 1. LGx is an abbreviation for \([\log(x+1)]\) where for the binary case, the base of the logarithm is 2, and \([x]\) is the least integer \(\geq x\); \(LG^2 x = LG(LGx)\), \(LG^{1+1} x = LG^1(LGx)\).

Define \(R'(n)\) as follows:

\[
R'(n) = \begin{cases} b_1b_2b_3 & n \leq 7 \\ R'(LG^n)B(n) & n > 7 \end{cases}
\]

where \(b_1b_2b_3\) is a binary representation of \(n\).

Next we define \(R(n)\) by

\[
R(n) = R'(n)0.
\]

\(R(n)\) is the representation of \(n\) which we use, and it has the prefix property (there exists no \(m < n\) for which \(R(m)\) is a prefix of \(R(n)\)). It can be proven [5] that for large \(n\),

\[
\frac{LG \cdot n}{3} = 1 + \sum_{i=1}^{LG \cdot n-3} LG^i n
\]

and \(R(n) \leq LGn + 2LG^2 n\).

To produce \(R(n)\) for \(n \geq 4\) perform the following algorithm:

**Algorithm 2.3.**

\(m = n;\)

\(R(n) = 0;\)

\(\text{while } m > 3 \text{ do } \xi = B(m);\)

\(R(n) = \xi R(n);\)

\(m = \text{the number of bits in } \xi\)

end
The cost of producing $R(n)$ is $O(\lambda(R(n)))$. But $\lambda(R(n)) \leq 2L\log n$. Therefore, the production time of $R(n)$ is linear in the length of the binary representation of $n$.

2.4. Encoding of Strings.

Let us show that the compression ratio of the composed encoding scheme tends to be optimal as the length of $\alpha$ grows to infinity.

Denote the complexity of $\alpha$ by $C(\alpha)$. Lempel and Ziv [2] show that for almost all strings emitted by an ergodic source with entropy $h$

$$C(\alpha) \leq \frac{\log n}{\log n}.$$

Shierling [6] shows that $\frac{\log n}{\log n}$ is the limit in probability of $C(\alpha)$, under more relaxed conditions.

Let $T(\alpha) = t_1t_2...t_k$ where $t_q = <l_q, j_q, a_q>$. Then the length of the encoding of $\alpha$ is bounded by

$$k \sum_{q} \log l_q + 2\log^2 l_q + \log j_q + 2\log^2 j_q + 1.$$  

$\log$ is a convex function. Thus, the length is bounded by

$$C(\alpha) \cdot (L\log n + 2\log^2 n + \log \frac{n}{C(\alpha)} + 2\log^2 \frac{n}{C(\alpha)} + 1).$$

For large $n$ this tends to

$$\frac{\log n}{\log n} (L\log n + 2\log^2 n + \log \frac{n}{h} + 2\log^2 \frac{n}{h}).$$

Therefore the compression ratio tends to $h$ for almost all strings, which is the best possible result.

More about the asymptotic complexity of sequences appears in [6].
Clearly, the main handicap of this data compression scheme is that the memory requirement is unbounded. In the next section we shall describe a method more suitable in cases of bounded memory.

3. Algorithm for Data Compression with Bounded Memory.

3.1. Introduction.

Ziv and Lempel [3] invented a compression method which first parses an input string into bounded length substrings and then uses an adaptive coding scheme which maps these substrings sequentially into uniquely decipherable code-words of fixed length over the desired alphabet. For a given \( \varepsilon \) and for all finite memory sources with entropy \( h \), the scheme uses a fixed amount of memory and its compression ratio is proven to be close to optimal up to \( \varepsilon \).

We implement the proposed compression method by combining Weiner's algorithm with the repetition finding algorithm described in Section 1. The two parameters of the algorithm are \( N \), the memory size, and \( F \), a bound on the length of source substring to be represented by a single code word.

3.2. The Compression Method.

Let \( \alpha \) be an input string. We shift a window of size \( N \) along \( \alpha \). It is set initially to hold zeros and a prefix of
length $F$ of $\alpha$. We encode $\alpha$ step by step. At each step we assume that the $N-F$ leftmost letters within the window have been encoded already and we encode as much as we can out of the rightmost $F$ letters. Let $\alpha_1$ denote the longest prefix of these $F$ letters such that there exists an earlier occurrence of $\alpha <0,\ell(\alpha)-1>$ within the window. Denote by $p_1$ the start position of such an occurrence (the window's left boundary is considered to be position 0). Then we encode $\alpha_1$ into

$$C_1 = \langle p_1, \ell(\alpha_1)-1, \alpha_1 <\ell(\alpha_1)>>,$$

and shift the window $\ell(\alpha_1)$ positions to the right.

For simplicity assume that the source and code alphabets are binary. To write each $C_1$ we use

$$L = \lceil \log_2(N-F) \rceil + \lceil \log_2(F) \rceil + 1$$

bits (padding with zeros if necessary). The code of $\alpha$ is the concatenation of its $C_1$'s.

### 3.3 Implementation of the Compression Method.

The compression method just described is similar to that in Section 2. There are two main differences:

1. The length of each $\alpha_1$ is bounded by $F$. To handle this constraint all we have to do is to follow the "$]\$" movements and when it goes "too far", produce a new $C_1$.

2. The size of the window is fixed. Thus when we shift its right end, we have to shift its left end too.

Algorithm 1.3 is designed to handle strings given in a
left to right mode. It currently holds information about a portion of the input string which starts at a fixed leftmost position, P₁, and ends in a position that advances to the right. If the fixed start position lies inside the window, it can simulate the window's right side movements. In fact the suffix of length F-1(α₁) of the window's content is ignored in such a simulation.

Weiner's algorithm handles strings in a right to left mode. Assume that Weiner's algorithm is applied from some fixed rightmost position, P₂, to the left and processed K letters. While the algorithm is proceeding, a stack is built which holds all actions on the data structure. Using this stack we can "reverse the algorithm" and get a new algorithm which acts upon the string from left to right in linear time while its right end is fixed. (Actually only a substructure of the data is necessary in our application and the algorithm can be improved accordingly.) If this fixed position lies inside the window the reversed algorithm can simulate the window's left side movements.

To implement the above compression method we have to be able to check for the existence of an occurrence of a string whose length is bounded by F-1 within the window. Therefore we take P₂=P₁+F-2. In this way if there exists such an occurrence, it occurs either on the left side of P₂ or on the right side of P₁; thus it is treated by at least one of the two algorithms.
To ensure that there always exist two positions $P_1$ and $P_2$ inside the window we design a sequence of such couples: Note that the $N-F$ leftmost letters are initially zero. To make the discussion uniform define $\xi$ to be $0^{N-F}$ and put the window on the first $N$ letters of $\xi$. In the sequel "position $j$" abbreviates "position $j$ in $\xi$". Define $P_1$ and $P_2$ by:

$$P_1 = \lfloor \frac{(N-F)/2}{F-2} \rfloor,$$

$$P_2 = \lfloor \frac{(N-F)}{2} \rfloor;$$

For $1 \geq 1$

$$P_2^{j+1} = P_1^j + \lfloor (N-F)/2 \rfloor,$$

$$P_1^{j+1} = P_2^j + 1 - F + 2.$$

Usually $F < N$. More explicitly we assume that $N > 9F-14$. In this case the number of couples inside the window is at most three; the leftmost is being used and the others are being developed. When the couple in use is shifted out, the one next to it is ready, and we start using it. At this moment, $P_2^{j+1}$ is located at position $\lfloor (N-F)/2 \rfloor$. Thus Weiner's algorithm has had enough time to produce the desired data structure. Therefore the composed algorithm is linear.

The assumption that $N > 9F-14$ is sound since for a given entropy $h$, $F$ should satisfy

$$F \leq \log N / h,$$

and, as $N$ increases the ratio between $F$ and $N$ decreases logarithmically. Thus, a reasonable choice of $N$ will satisfy the inequality.
3.4. Remarks.

Ziv and Lempel [3] do not use the shift facility of the window to prove that the compression ratio of their scheme is close to the entropy; the input is partitioned into sections, each of which is of length $N$, which is the buffer size. Thus we can develop an algorithm which is very similar to Algorithm 2.2 but has a bound on the length of substrings that are represented by a single code word. When the current section is encoded the memory is cleared and the encoding of a new section is started. The intuitive disadvantage of such a scheme is that when we restart, we cannot use known substrings and therefore for a while the compression is poor. However we gain simplicity and may hold a larger portion of the input string in core.

A compromise between the above scheme and the one described in 3.2 is the following: Design a sequence of positions $P_i$ defined by:

$$P_1 = 0$$

For $i \geq 1$,

$$P_{i+1} = P_i + 1 \cdot (N-F)/2$$

There are at least two such positions inside a window of size $N$. When $P_i$ first enters into the window, we initiate the repetition finder and stop it when it represents $\varepsilon(P_i, P_i + 1 + F - 2)$. Denote the corresponding data structure by $D_i$. When $P_i$ is shifted out, drop $D_i$ out of memory.
Clearly, we use $D_1$ and $D_{1+1}$ for matching with a new portion of $\xi$. In this way we may always refer to at least half of the window and get a more effective scheme.

The ideas above illustrate two variations of the two basic algorithms; others are possible too.
REFERENCES


