On-line longest fragment first parsing algorithm

Hideo Nagumo, Mi Lu *, Karan Watson

Department of Electrical Engineering, Texas A & M University, College Station, TX 77843-3128, USA

Received 29 March 1996
Communicated by S.G. Akl

Abstract

The central part of dictionary type data compression is the process of parsing the input string into phrases that are to be replaced with the corresponding dictionary indexes. In this article, we study the longest fragment first (LFF) parsing strategy for static dictionary compression. We present an on-line algorithm for LFF parsing that requires lookahead of length $O(L^3)$, where $L$ is the maximum length of the dictionary entries.

Keywords: Data compression; Analysis of algorithms; Static dictionary; Longest fragment first; Parsing; On-line

1. Introduction

We present an on-line longest fragment first (LFF) parsing algorithm for static dictionary compression. An on-line data compression algorithm processes the input string in a one-pass left-to-right scan with limited lookahead. Dictionary type data compression works by replacing groups of consecutive symbols (phrases) in the input string with the corresponding dictionary indexes [1]. The dictionary can be static or dynamic. In static dictionary compression, the dictionary contains a predetermined fixed set of entries. In dynamic dictionary compression, the dictionary changes its entries during compression, reflecting newly encoded phrases in the input string.

There are different strategies for parsing the input string into phrases that are to be replaced with dictionary indexes. Many data compression programs based on dictionary methods used today, e.g., UNIX compress, employ a greedy parsing strategy. In this strategy, the input string is processed from left to right, and at each step the longest phrase starting at the first symbol of the unencoded portion of the string that matches a dictionary entry is encoded. An optimal parsing of the input string is a shortest possible sequence of dictionary indexes such that the concatenation of the corresponding dictionary entries forms the input string. If fixed length dictionary indexes are used, an optimal parsing produces the minimum number of dictionary indexes. The LFF parsing algorithm parses the input string by repeatedly locating the longest phrase in the unencoded portion of the input that matches a dictionary entry. This phrase is replaced with the corresponding dictionary index [4]. In general, the compression performance of LFF lies between the greedy and optimal parsing [4].

On-line capability of a data compression algorithm is important for real-time operations because it allows encoding with a bounded delay. Greedy parsing inherently possesses on-line capability, and the

* Corresponding author.
distance the encoder has to look ahead is equal to the length of the longest dictionary entries only. Optimal parsing, in general, cannot be computed on-line because parsing decisions at the end of an input string may influence parsing decisions at the beginning of the input string (although there is a special property of a dictionary that makes on-line optimal parsing possible [2]). Conventionally, the LFF parsing is not computed on-line because the longest phrase we look for may be located at the end of the input string. In this article, we describe an on-line LFF parsing algorithm for static dictionary compression that requires lookahead of length $\frac{1}{2}(L - 1)(L - 2) + L$, where $L$ is the maximum length of the dictionary entries.

Throughout this article, we assume that the dictionary indexes have fixed length. With this assumption, the number of dictionary indexes in a parsing determines the length of the compressed form. Given two parsings, the one with the smaller number of indexes will have the smaller compressed form. We also assume that the dictionary $D$ is represented as a suffix tree. A suffix tree for a dictionary $D$ is a trie, or digital search tree data structure composed of all the phrases in $D$. With this assumption, the time for searching for a phrase in $D$ is proportional to the length of the phrase, and searching for all the prefixes of a phrase requires the same time complexity as searching for the phrase. Since the input string needs to be completely encoded, $D$ is required to include all members of the input symbol set [5].

An input string has $n$ symbols which are numbered from 1 to $n$. An input position $k$ is said to be on the left (right) of position $i$ if $k < i$ ($k > i$). An input symbol at position $i$ is denoted as $x_i$. If a phrase in the input string is also in $D$, we call it a matched phrase or simply a match. A matched phrase with length $l$ starting at input position $k$ is denoted by $M(k, l)$. A match that has been selected to be replaced with the corresponding dictionary index is called a selected match, and the position of the first symbol of a selected match is called a break point. A match $M(k, l)$ is said to be the left-most match of length $l$ with respect to (w.r.t.) position $i$ if there is no match $M(j, l)$ of length $l$ for which $i \leq j < k$. Notice that if there is a match of length $l$ at position $i$, it is the left-most match of length $l$ w.r.t. position $i$.

2. Conventional LFF parsing algorithm

In this section, we review the conventional LFF parsing algorithm. We will modify this algorithm later to derive our on-line LFF parsing algorithm.

The LFF parsing algorithm was originally developed for data base compression by Schuegraf and Heaps [4]. Stauffer and Hirschberg later presented a parallel LFF parsing algorithm in [5]. They gave a high level description of the LFF algorithm as shown in Fig. 1.

We start from Stauffer and Hirschberg's algorithm in Fig. 1 to derive our on-line LFF parsing algorithm because they provide the most detailed description of the LFF parsing algorithm among those that are known to us. We summarize the procedure of each step of their algorithm in the next paragraph.

In Step 1, they compute match lists at each input position. Since the length of a dictionary entry is limited to $L$, there are at most $L$ matches at each input position. They use an array $m(i, j)$ ($1 \leq i \leq n, 1 \leq j \leq L$) for the lists. If there is a match of length $j$ at position $i$, $m(i, j)$ is set to 1, otherwise $m(i, j)$ is 0. In Step 2, they find the maximum match length $L^*$ in the input string. In Step 3, they iteratively compute the selection and elimination steps, starting from the match length $l = L^*$ down to 1. That is, in Step 3a, they select the matches of length $l$ so that there are a maximum number of selected matches of the length, and in Step 3b they essentially eliminate the matches shorter than $l$ that overlap the selected ones from the match lists. If a match $M(i, l)$ is selected, they mark the positions covered by $M(i, l)$ as selected positions, and set $a_k := i - k$ for the $l - 1$ preceding positions from $k = i - l + 1$ to $k = i - 1$ to indicate that all matches at position $k$ exceeding length $l - i$ have been eliminated from the match lists at position $k$.

1. Compute match lists at each input position
2. Compute maximum match length, $L^*$
3. For each match length $l := L^*$ down to 1 do
   a. Find a maximum collection $C$ of non-overlapping matches of length $l$
   b. Update match lengths at positions overlapping $C$

Fig. 1. Stauffer and Hirschberg's LFF parsing algorithm.
3. for each match length $l := L$ down to 1 do
   begin
   $i :=$ left-most unselected position;
   while $i \leq n$ do
     begin
     if there is a match $M(i, l)$ then
       begin
       select $M(i, l)$ and mark positions $i, \ldots, i + l - 1$ as selected positions;
       update match lengths at positions $i - l + 1, \ldots, i - 1$;
       end
     $i :=$ next unselected position;
     end
   end

Fig. 2. Modified Step 3 of the LFF parsing algorithm.

If we compute the algorithm sequentially, each of Step 1 and Step 3 requires $O(nL)$ time, and Step 2 can be incorporated in Step 1. Thus this algorithm requires $O(nL)$ time.

Although they provide Step 2 for minimizing overall computation time, we skip this step for simplicity because the theoretical time bound of the algorithm is the same without it. Instead, we start from the match length $l := L$ in Step 3.

We assume that the maximum collection of non-overlapping matches of length $l$ in Step 3a is computed from left to right. That is, we repeatedly select the left-most match of length $l$ and eliminate the matches of the same length that overlap the selected one until all the matches of length $l$ are either selected or eliminated.

In the algorithm in Fig. 1, the elimination of matches with length less than $l$ that overlap the selected ones is done after the selection of all matches with length $l$ has been completed. This, however, can be done immediately after each match of length $l$ is selected. Thus we can modify Step 3 as in Fig. 2, without altering the result of the algorithm.

3. Lemmas

In this section, we show three lemmas that are necessary for modifying the conventional LFF parsing algorithm to the on-line version.

At first, we give some definitions to simplify the descriptions of the lemmas. If a match $M(k_b, l_b)$ is eliminated after another match $M(k_a, l_a)$ is selected, we say $M(k_a, l_a)$ eliminates $M(k_b, l_b)$. In the modified Step 3 in Fig. 2, $M(k_a, l_a)$ may eliminate $M(k_b, l_b)$ if $l_a > l_b$ and they overlap ($k_a - l_a + 1 \leq k_b \leq k_a + l_a - 1$), or if $l_a = l_b$ and $k_a < k_b$ and they overlap ($k_b \leq k_a + l_a - 1$). A match is selected if it is not eliminated until it becomes the left-most one among the longest matches that have been neither selected nor eliminated. Eventually, a match $M(k_b, l_b)$ will be eliminated or selected. However, if the result would be different when assuming that another match $M(k_a, l_a)$ did not exist, we say $M(k_a, l_a)$ affects $M(k_b, l_b)$. If a match at an input position is eliminated, it is as though the match did not take place at this particular position. Thus a match that will be eliminated will not affect any other match. Also, the influence of a match starts when it is selected. Thus a match will not affect any match that is longer than itself because the longer match has been selected earlier.

Let us consider the LFF parsing algorithm with the modified Step 3 in Fig. 2. Suppose that on a particular iteration of the while loop, a match $M(k, l)$ has been selected and the matches of length less than or equal to $l$ that overlap $M(k, l)$ have been eliminated. Then, the matches on the left of $M(k, l)$ and those on the right are isolated, and further parsing of the left side is not affected by further parsing of the right side because there is no match on one side that overlaps any match on the other side. Thus we obtain the following lemma.

**Lemma 1.** If a match $M(k, l)$ has been selected in a process of the LFF parsing, any match $M(k_i, l_i)$ for which $l_i < l$ and $k_i + l_i - 1 < k$ will not be affected by any match $M(k_r, l_r)$ for which $l_r < l$ and $k_r > k$.

Suppose that there are $q$ matches $M(i, l_1), M(i, l_2), \ldots, M(i, l_q)$ at position $i$, and that position $i$ is a break point, then one of them will be selected. Using Lemma 1, we prove that only the left-most match w.r.t. position $i$ for each match length can affect these matches at position $i$. This means that if position $i$ is a break point we need to check only the left-most match w.r.t. position $i$ for each match length to compute the next break point. This is
formally stated in the next lemma.

**Lemma 2.** Suppose that input position $i$ is a break point. A match $M(k, l)$ will affect the matches at position $i$ only if $M(k, l)$ is the left-most match of length $l$ w.r.t. position $i$.

**Proof.** We prove this lemma by exhausting all possible cases. Suppose that $M(k, l)$ is not the left-most match of length $l$ w.r.t. position $i$ and that the left-most match of length $l$ is $M(j, l)$ ($i \leq j < k$).

**Case 1:** $M(k, l)$ overlaps $M(j, l)$ ($j < k \leq j + l - 1$).

1.1. if $M(j, l)$ is not eliminated by any longer match, then $M(k, l)$ will be eliminated by $M(j, l)$ and will not affect any match at position $i$;

1.2. if $M(j, l)$ is eliminated by $M(g, h)$ for which $h > l$ and $g \geq j$, then $M(k, l)$ will also be eliminated by $M(g, h)$ and will not affect any match at position $i$;

1.3. if $M(j, l)$ is eliminated by $M(g, h)$ for which $h > l$ and $g < j$, then, by Lemma 1, $M(k, l)$ will not affect any match at position $i$ because $l < h$ and $k > g$.

**Case 2:** $M(k, l)$ does not overlap $M(j, l)$ ($j + l - 1 < k$).

2.1. if $M(j, l)$ is not eliminated by any longer match, then $M(j, l)$ is selected and, by Lemma 1, $M(k, l)$ will not affect any match at position $i$;

2.2. if $M(j, l)$ is eliminated by a longer match $M(g, h)$ for which $h > l$ and $g - l + 1 \leq j \leq g + h - 1$, then, by Lemma 1, $M(k, l)$ will not affect any match at position $i$ because $l < h$ and $k > g$.

The next lemma states that we do not need to search for the left-most matches for arbitrarily long distances, but it is enough if we look for a match of length $l$ up to $\frac{1}{2}(l - 1)(l - 2)$ positions to the right.

**Lemma 3.** Suppose that input position $i$ is a break point and that $M^i(k, l)$ is the left-most match of length $l$ w.r.t. position $i$. If $k_i - i > \frac{1}{2}(l - 1)(l - 2)$, then $M^i(k, l)$ will not affect any match at position $i$.

**Proof.** We consider only the left-most matches w.r.t. position $i$ in this proof because, by Lemma 2, these are the only matches that may affect the matches at position $i$. Also, we consider only the matches that are shorter than or equal to $l$ because if $M^i(k, l)$ is to affect the matches at position $i$ it affects them through the selection and elimination process after $M^i(k, l)$ is selected, and by the time $M^i(k, l)$ is selected all the longer matches must have already been either eliminated or selected. Let us denote these left-most matches of length $1, 2, \ldots, l$ by $M^i(k_1, 1)$, $M^i(k_2, 2)$, $\ldots$, $M^i(k_l, l)$ respectively ($k_i = i$ by assumption). In this proof, left-most means left-most w.r.t. position $i$.

We first prove the following statement: if a left-most match $M^i(k, l)$ affects a match at position $i$, then for any left-most match $M^i(k, a)$ for which $a < l$ and $k_a < k_i$ there exists at least one left-most, match $M^i(k, b)$ for which $a < b < l$ that overlaps $M^i(k, a)$. This means that we must have an overlapping chain of left-most matches from $M^i(k, l)$ to a match at position $i$ in descending order in length. We prove this statement by contradiction. Suppose that there is a left-most match $M^i(k, a)$ for which $a < l$ and $k_a < k_i$, but there is no left-most match among $M^i(k_{a+1}, a+1)$, $M^i(k_{a+2}, a+2)$, $\ldots$, $M^i(k, l)$ that overlaps $M^i(k, a)$. Then, $M^i(k, a)$ will be selected, and, by Lemma 1, $M^i(k, l)$ will not affect any match at position $i$. Thus the above statement is proved.

The longest distance that $M^i(k, l)$ can be away from position $i$ without violating the condition in the above statement is obtained when $M^i(k, 1)$ and $M^i(k, 2)$ are at position $i$, and for any left-most match $M^i(k, c)$ for which $3 \leq c \leq l - 1$ the left-most symbol overlaps with the right-most symbol of $M^i(k_{c-1}, c-1)$ and the right-most symbol overlaps with the left-most symbol of $M^i(k_{c+1}, c+1)$. The longest distance is given by $1 + 2 + \cdots + (l - 2) = \frac{1}{2}(l - 1)(l - 2)$.

A more general description of Lemma 3 can be given as a relation between two matches as follows: a match $M(k, l)$ will not affect another match $M(k_a, l_a)$ for which $l_a > l$ if $k_a - k_a > \frac{1}{2}(l_a - l_a) \cdot (l_a + l_b - 3)$.

Since the longest match length is $L$, Lemma 3 means that no match that is more than a distance of $\frac{1}{2}(L - 1)(L - 2)$ away from position $i$ will affect the matches at position $i$. 


4. On-line LFF parsing algorithm

In this section, we describe our on-line LFF parsing algorithm.

Suppose that an input string is processed from left to right and has been encoded up to position \( i - 1 \). Thus we have to select a match at position \( i \), and position \( i \) is a break point. By Lemma 2, selection of a match at position \( i \) can be done by manipulating the positions of the left-most matches w.r.t. position \( i \), and by Lemma 3, we can disregard the left-most match of length \( l \) w.r.t. position \( i \) if its starting position is farther right than \( i + \frac{1}{2}(l - 1)(l - 2) \). Thus if we have lookahead of length \( \frac{1}{2}(L - 1)(L - 2) + L \) we can compute LFF parsing on-line. There is an addition of \( L \) because we may have to find a match of length \( L \) at the right-most position that can affect the matches at position \( i \).

We modify Step 1 of the algorithm in Fig. 1 so that the match lists are generated only for the first \( \frac{1}{2}(L - 1)(L - 2) + 1 \) positions from position \( i \). We also find the positions of the left-most matches w.r.t. position \( i \) in this step. We use the distance \( d_j \) from position \( i \) to specify the position of the left-most match of length \( l \). If \( d_j \) is greater than \( \frac{1}{2}(l - 1)(l - 2) \), we simply assign \( \infty \) to it to indicate that this left-most match will not affect the matches at position \( i \).

We then modify the step in Fig. 2 so that only the left-most matches w.r.t. position \( i \) participate in the selection and elimination step as shown in Fig. 3. We go through the left-most matches one by one from the longest one toward the shortest one until a selected match is found at position \( i \). For each left-most match, we check if it has been eliminated by the longer selected matches. If it has not, then it is selected and eliminates the shorter left-most matches that overlap it. If the left-most match of length \( l \) is eliminated, \( \infty \) is assigned to \( d_j \).

The while loop in Fig. 3 starts from the longest match length \( L \), and in each iteration the length is decremented. On a particular iteration, we check the distance \( d_j \) of the left-most match of length \( l \). If \( d_j = 0 \), the match of length \( l \) is at position \( i \) and has not been eliminated. Thus the match of length \( l \) at position \( i \) is encoded, and the while loop stops here. If \( 0 < d_j < \infty \), then the match of length \( l \) at position \( i + d_j \) is selected. Therefore in the next for loop, the left-most matches of length less than \( l \) that overlap or that are on the right of the selected match of length \( l \) are eliminated. We eliminate the left-most matches that are on the right of the selected match as well because, by Lemma 1, they will not affect the matches at position \( i \). The while loop will eventually terminate because \( d_j = 0 \) by assumption.

Since the same match may be eliminated more than once in the step shown in Fig. 3, we further modify this step so that each match is eliminated at most once. In the step shown in Fig. 3, to determine if a match \( M^j(k_j, l) \) is eliminated, we check the condition \( d_j < \infty \). However, we can also determine this by going through the matches longer than \( l \) \( M^j(k_{L-1}, L - 1), \ldots, M^j(k_{l+1}, l + 1) \), and by checking if any of them would eliminate \( M^j(k_j, l) \). We show that to determine if \( M^j(k_j, l) \) is eliminated by any of the longer left-most matches, we need to check if \( M^j(k_j, l) \) would be eliminated by only one left-most match: the shortest match among the selected left-most matches that are longer than \( l \) (the shortest selected match). Suppose that \( l < a < b \) and that both \( M^j(k_a, a) \) and \( M^j(k_b, b) \) are selected. We need to prove that if \( M^j(k_b, b) \) would eliminate \( M^j(k_j, l) \), then \( M^j(k_a, a) \) would also eliminate \( M^j(k_j, l) \). We have \( a - l - 1 < k_b \) because both \( M^j(k_a, a) \) and \( M^j(k_b, b) \) are selected. We also have \( k_a + l - 1 > k_b \), because \( M^j(k_b, b) \) would eliminate \( M^j(k_i, l) \). Combining the two, we get \( k_a + l - 1 > k_b \). Considering \( k_a < k + a - 1 \), we get \( k_a > k + a - 1 \), we get \( k_a + l - 1 > k_b \). Thus \( M^j(k_a, a) \) would eliminate \( M^j(k_j, l) \).

The further modified selection and elimination step is shown in Fig. 4 as Step 2 of the whole on-line LFF parsing algorithm. As before, we go through the left-most matches from the longest one toward the

\[
\begin{align*}
l &:= L; \\
\text{while } d_j > 0 \text{ do} \\
\quad \text{begin} \\
\quad \quad \text{if } d_j < \infty \text{ then} \\
\quad \quad \quad \text{for } j := l - 1 \text{ down to } 1 \text{ do} \\
\quad \quad \quad \quad \text{if } d_j + j - 1 > d_j \text{ then } d_j := \infty; \\
\quad \quad \quad l := l - 1; \\
\quad \quad \text{end} \\
\text{encode phrase } x_i \cdots x_{i+j-1}; \\
\end{align*}
\]

Fig. 3. Selection and elimination step in the on-line LFF parsing algorithm.
\begin{verbatim}
\textbf{i} := 1;
\textbf{while } \textbf{i} \leq n \textbf{ do }
\hspace{1em} \textbf{begin}
1. \textbf{for } \textbf{l} := 1 \textbf{ to } L \textbf{ do }
\hspace{2em} \textbf{begin}
\hspace{3em} \textbf{find distance } d_l \textbf{ of the left-most match of length } l \textbf{ w.r.t. }
\hspace{3em} \textbf{position } i
\hspace{3em} \textbf{from position } i \textbf{ (if } d_i > \frac{1}{2}(l-1)(l-2), \text{ assign } \infty \textbf{ to } d_i); 
\hspace{2em} \textbf{end}
2. \textbf{l} := L;
\hspace{1em} \textbf{s} := d_L;
\hspace{1em} \textbf{while } d_l > 0 \textbf{ do }
\hspace{2em} \textbf{begin}
\hspace{3em} \textbf{if } d_{l-1} + (l-1) - 1 \geq s \textbf{ then } d_{l-1} := \infty;
\hspace{3em} \textbf{else } \textbf{s} := d_l;
\hspace{3em} \textbf{l} := l - 1;
\hspace{2em} \textbf{end}
\hspace{1em} \textbf{encode phrase } x_i \ldots x_{i+l-1};
\hspace{1em} \textbf{i} := i + l;
\hspace{1em} \textbf{end}
\end{verbatim}

Fig. 4. On-line LFF parsing algorithm.

shortest one until a selected match is found at position \( i \), but we do not have a nested loop in the \textbf{while} loop in Fig. 4. We use variable \( s \) to keep the position of the shortest selected match. For each match length \( l \), we determine whether the left-most match of length \( l - 1 \) is eliminated by the shortest selected match by checking if \( d_{l-1} + (l - 1) - 1 \geq s \). If it is not eliminated, it becomes the new shortest selected match and \( d_{l-1} \) is assigned to \( s \).

In the on-line LFF parsing algorithm in Fig. 4, variable \( i \) keeps the position of the first symbol of the uncoded portion in the input string. Thus 1 is assigned to \( i \) at the beginning of the algorithm. Then, a loop is repeated until \( i \) becomes greater than \( n \), the position of the last symbol in the input string. In each iteration of the loop, one phrase is encoded at the beginning of the uncoded portion in the input string, and the new position of the first symbol of the uncoded portion is assigned to \( i \).

Each iteration of Step 2 requires \( O(L) \) time. Thus this step requires \( O(nL) \) time. It can be easily seen that Step 1 requires \( O(nL) \) time. Therefore in total, our on-line LFF parsing algorithm requires \( O(nL) \) time. This time complexity is the same as that of the conventional LFF parsing algorithm.

5. Conclusions

We presented an on-line LFF parsing algorithm for static dictionary compression that required lookahead of length \( \frac{1}{2}(L - 1)(L - 2) + L \). We did not elaborate on Step 1 of our algorithm in Fig. 4, nor did we show the details that could make this algorithm more efficient (e.g., permanently marking the selected matches that are not at position \( i \) in Step 2 so that they can be encoded at the beginning of the outer \textbf{while} loop). We rather showed the basic modifications that were used to derive the on-line LFF parsing algorithm from the conventional one.

On-line dictionary compression algorithms are important not only for their own applications, but also for designing parallel algorithms of dictionary compression [3].

References