The Effect of Flexible Parsing for Dynamic Dictionary Based Data Compression *

Yossi Matias† Nasir Rajpoot‡ Süleyman Cenk Ṣahinalp§

We report on the performance evaluation of greedy parsing with a single step lookahead (which we call flexible Parsing or $\mathcal{FP}$) as an alternative to the commonly used greedy parsing (with no-lookaheads) scheme. Greedy parsing is the basis of most popular compression programs including `UNIX compress` and `gzip`, however it usually results in far from optimal parsing/compression with regard to the dictionary construction scheme in use. Flexible parsing, however, is recently shown to be optimal [MS99], i.e., partitions any given input to the smallest number of phrases possible, for dictionary construction schemes which satisfy the prefix property throughout their execution.

There is an on-line linear time and space implementation of the $\mathcal{FP}$ scheme via the trie-reverse-trie pair data structure [MS99]. In this paper, we introduce a more practical data structure to implement $\mathcal{FP}$ scheme with the LZW variant of the Lempel-Ziv'78 dictionary construction method [Wel84, ZL78]. The data structure is randomized; its expected theoretical performance matches the worst case performance of the trie-reverse-trie-pair . However, because it is based on Karp-Rabin fingerprints [KR87] it avoids more complex pointer operations and is expected to do better in practice.

We report on the implementations of two compression algorithms which uses (1) $\mathcal{FP}$ with LZW dictionary (LZW-$\mathcal{FP}$), and (2) $\mathcal{FP}$ with an alternative flexible dictionary (FPA as introduced in [Hor95]). We test our implementations on a collection of input sequences which includes textual files, DNA sequences, medical images, and pseudorandom binary files, and compare our results with two most popular compression programs `compress` and `gzip`. Our initial results suggest that flexible parsing is especially useful for non-textual data, for which it improves over the compression rates of `compress` and `gzip` by up to 20% and 35% respectively.

---

*Part of this work was presented in Workshop on Algorithmic Engineering, Saarbrucken, Germany, 1998. Collected works in this workshop can be found in Max-Planck Institute Technical Report MPI-I-98-1-019.

†Department of Computer Science, Tel-Aviv University, Israel and Bell Labs, Murray Hill, USA; `matias@math.tau.ac.il`. Research supported in part by an Alon Fellowship, by the Israel Science Foundation founded by The Academy of Sciences and Humanities, and by the Israeli Ministry of Science.

‡Department of Mathematics, Yale University, USA, and Department of Computer Science, University of Warwick, UK; `rajpoot@noodle.med.yale.edu`.

§Department of Computer Science, University of Warwick, UK and Center for BioInformatics, University of Pennsylvania, USA; `cenk@dcs.warwick.ac.uk`. Research supported by NATO research grant CRG-972175 and ESPRIT LTR Project no. 20244 - ALCOM IT.

1 Introduction

The most common compression algorithms used in practice are the dictionary schemes (a.k.a. parsing schemes [BCW90], or textual substitution schemes [Sto88]). Such algorithms are based on maintaining a *dictionary* of strings that are called *phrases*, and replacing substrings of an input text with pointers to identical phrases in the dictionary.

A dictionary can be constructed in static or dynamic fashion. Almost all practical compression algorithms use *dynamic* schemes (as introduced by Ziv and Lempel [ZL77, ZL78]) in which the dictionary is initially empty and is constructed incrementally. Most of these algorithms construct dictionaries that satisfy the *prefix property* for any input string: in any execution step of the algorithm, all prefixes of any given phrase are also phrases in the dictionary.

Dictionary based compression algorithms of particular interest are the LZ78 method [ZL78], its LZW variant [Wel84], and the LZ77 method [ZL77]. All three algorithms are asymptotically optimal for a wide range of probabilistic sources However, it is recently demonstrated that LZ78 (as well as LZW) approaches the asymptotic optimality faster than LZ77. [1] It is an open question why their relative performances vary in different applications.

Given a dictionary construction scheme, there is more than one way to *parse* the input, i.e., choose which substrings in the input text will be replaced by respective *codewords*. Almost all dynamic dictionary based algorithms in the literature ([ZL77, ZL78, Wel84, MW85, Yok92]) use *greedy parsing* which is fast and can be applied on-line. However, it usually results in far from optimal parsing/compression: for the LZW dictionary method, there are strings T which can be (optimally) parsed to some m phrases, for which the greedy parsing obtains $\Omega(m^{3/2})$ phrases ([MS99]; a similar result for static dictionaries is in [GSS85]).

Perhaps the most fundamental question regarding dictionary compression algorithms is to find good dictionary construction schemes; i.e., schemes that enable good encoding of the input with small redundancy. Unfortunately, a simple counting argument shows that there cannot exist a single dictionary construction scheme that is superior to any other scheme for all inputs. The advantage of one dictionary construction scheme over another can only apply with regard to restricted classes of input texts.

In [Hor95] it was demonstrated that the compression achieved by LZW algorithm on some standard benchmark files can be improved by looking ahead a few steps. However it was noted that: "An optimal parsing scheme would also have to consider the possibility of matching a short first string and then a short second string in order to match a very long third string. We will however reject such possibilities as being too expensive to implement (for minimal expected gain in compression). Our non-greedy version of LZW will only look ahead by one parsed string when choosing its course of action." In fact the algorithm proposed in [Hor95] not only changes the

[1] The average number of bits output by LZ78 or LZW, for the first n characters of an input string created by an i.i.d. source is only $O(1/\log n)$ more than its entropy [JS95, LS95]. A similar result for more general, unifilar, sources has been obtained by Savari [Sav97] - for the average case. For the LZ77 algorithm, this redundancy is as much as $O(\log\log n/\log n)$ [Wyn95]. Also, for low entropy strings, the worst case compression ratio obtained by the LZ78 algorithm is better (by a factor of 8/3) than that of the LZ77 algorithm [KM97].

parsing scheme but also constructs an entirely different dictionary from that of LZW on a given string - hence compression improvement over LZW is not guaranteed, and the notion of optimality is not clear. The worst case running time of this algorithm is $O(|T^{3/2}|)$, where $|T|$ is the input size.

Recently [MS99] resolved the issue of on-line optimal parsing in this context by showing that *for all dictionary construction schemes with the prefix property, greedy parsing with a single step lookahead is optimal on all input strings* - this scheme is called *flexible parsing* or $\mathcal{FP}$. A new data structure is also introduced in [MS99] which implements the algorithm that uses LZW dictionary construction with $\mathcal{FP}$ in $O(|T|)$ time and space proportional to the number of phrases in the dictionary. The space and time complexity of this data structure is identical to that of the original LZW implementation, hence optimal compression can be achieved without any overhead in complexity. [2]

In this study, we report an experimental evaluation of $\mathcal{FP}$ in the context of LZW dictionary construction scheme. We can summarize our results as follows.
(1) We demonstrate that optimality of $\mathcal{FP}$ (in the context of LZW dictionary construction) translates into considerable improvement over greedy parsing in practice - we denote this algorithm LZW-$\mathcal{FP}$. We compare LZW-$\mathcal{FP}$ with `compress`, `gzip` and the algorithm which uses the flexible dictionary construction of [Hor95] and $\mathcal{FP}$ (we call this algorithm FPA). On practical data files (including DNA/protein sequences, medical images, and files from the Calgary corpus and Canterbury corpus benchmark suites) both LZW-$\mathcal{FP}$ and FPA perform better than LZW (UNIX `compress`) in terms of compression, up to 20%. LZW-$\mathcal{FP}$ and FPA are also superior to `gzip` for most non-textual data and all types of data of size more than $1MB$. For pseudo-random strings and DNA sequences the improvement is up to 35%. On shorter text files, `gzip` is still the champion, which is followed by FPA and LZW-$\mathcal{FP}$. Curiously, for most files the flexible dictionary construction proves to be better than the greedy (LZW) dictionary construction.
(2) We introduce a new data structure based on Karp-Rabin fingerprints [KR87] to efficiently implement $\mathcal{FP}$. Currently our algorithms run about $3-5$ times slower than `compress` which is the fastest among all algorithms, both during compression and decompression. We are in the process of improving our implementations and hence leave reporting on explicit timing results to the full paper.
(3) We investigate whether better asymptotic properties of LZ78 based algorithms in comparison to LZ77 translate into improved compression. We demonstrate that on pseudorandom bit streams (with various distributions) the redundancy in the output of each of the four programs approach to the expected asymptotic behavior very fast - requiring less than 1KB for each of the different distributions; better asymptotic properties of LZW in comparison to LZ77 is very visible. For files of size $1MB$, `compress` can improve over `gzip` up to 20% in compression achieved.

All the software, documentation, and detailed experimental results reported in this paper are available on the WWW [Sou]. The readers are encouraged to check updates to the web site and

[2]Note that a parsing scheme can be optimal with respect to a fixed dictionary construction scheme that constructs the same dictionary for any given input. Although the flexible dictionary of [Hor95] is different from that of LZW, $\mathcal{FP}$ is still optimal as this dictionary satisfies the prefix property as well.

try our software package.

2 The Compression Algorithms

In this section we describe how each of the algorithms of our consideration, i.e., (1) the LZ77 algorithm (the basis for `gzip`), (2) the LZW variant (the basis for `UNIX compress`) of the LZ78 algorithm, (3) LZW-$\mathcal{FP}$ algorithm and (4) FPA algorithm, work. Each of the algorithms fit in a general framework that we describe below.

We denote a compression algorithm by $\mathcal{C}$, and its corresponding decompression algorithm by $\mathcal{C}^{\leftarrow}$. The input to $\mathcal{C}$ is a string T, of n characters, chosen from a constant size alphabet Σ; in our experiments Σ is either ascii or is $\{0, 1\}$. We denote by $T[i]$, the i^{th} character of T $(1 \leq i \leq n)$, and by $T[i : j]$ the substring which begins at $T[i]$ and ends at $T[j]$; notice that $T = T[1 : n]$.

The compression algorithm $\mathcal{C}$ compresses the input by reading the input characters from left to right (i.e. from $T[1]$ to $T[n]$) and by partitioning it into substrings which are called blocks. Each block is replaced by a corresponding label that we call a codeword. We denote the j^{th} block by $T[b_j : b_{j+1} - 1]$, or shortly T_j, where $b_1 = 1$. The output of $\mathcal{C}$, hence, consists of codewords $C[1], C[2], \ldots, C[k]$ for some k, which are the codewords of blocks $T_1, T_2, \ldots, T_k$ respectively.

The algorithm $\mathcal{C}$ maintains a dynamic set of substrings called the dictionary, $\mathcal{D}$. Initially, $\mathcal{D}$ consists of all one-character substrings possible. The codewords of such substrings are their characters themselves. As the input T is read, $\mathcal{C}$ adds some of its substrings to $\mathcal{D}$ and assigns them unique codewords. We call such substrings of T phrases of $\mathcal{D}$. Each block T_j is identical to a phrase in $\mathcal{D}$: hence $\mathcal{C}$ achieves compression by replacing substrings of T with pointers to their earlier occurrences in T.

The decompression algorithm $\mathcal{C}^{\leftarrow}$ that corresponds to $\mathcal{C}$, takes $C[1 : k]$ as input and computes $T[1 : n]$ by replacing each $C[j]$ by its corresponding block T_j. Because the codeword $C[j]$ is a function of $T[1 : b_j - 1]$ only, the decompression can be correctly performed in an inductive fashion.

Below, we provide detailed descriptions of the compression algorithms considered, both the new and the old for the sake of completeness .

Description of the LZ77 Algorithm. The LZ-77 algorithm reads the input characters from left to right while inserting all its substrings in $\mathcal{D}$. In other words, at the instance it reads $T[i]$, all possible substrings of the form $T[j : \ell]$, $j \leq \ell < i$ are in $\mathcal{D}$, together with all substrings of size one. The codeword of the substring $T[j : \ell]$, is the 2-tuple, $(i - j, \ell - j + 1)$, where the first entry denotes the relative location of $T[j : \ell]$, and the second entry denotes its size. LZ77 uses greedy parsing: the m^{th} block $T_m = T[b_m : b_{m+1} - 1]$ is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1} - 1]$.

Description of the LZW Algorithm. The LZW algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : b_{m+1}]$. Hence the phrases of LZW are the substrings obtained by concatenating the blocks of T with the next character following them, together with all possible substrings of size one. The codeword of the phrase $T[b_m : b_{m+1}]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. Thus, the codewords of substrings do not change in LZW algorithm. LZW uses greedy parsing as well: the m^{th} block T_m is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1} - 1]$. Hence, no two phrases can be identical in the LZW algorithm. Please see the appendix for an example execution of the LZW algorithm.

Description of the LZW-$\mathcal{FP}$ Algorithm. The LZW-$\mathcal{FP}$ algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b'_m : b'_{m+1}]$, where b'_m denotes the beginning location of block m if the compression algorithm used were LZW. Hence for dictionary construction purposes LZW-$\mathcal{FP}$ emulates LZW: for any input string LZW and LZW-$\mathcal{FP}$ build identical dictionaries. The output generated by these two algorithms however are quite different. The codeword of the phrase $T[b'_m : b'_{m+1}]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. LZW-$\mathcal{FP}$ uses flexible parsing: intuitively, the m^{th} block T_m is recursively defined as the substring which results in the longest advancement in the next iteration. More precisely, let the function f be defined on the characters of T such that $f(i) = \ell$ where $T[i : \ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\ell]$. Then, given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\alpha - 1]$. Please see the appendix for an example execution of the LZW-$\mathcal{FP}$ algorithm.

Description of the FPA Algorithm. The FPA algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : f(b_m)]$, where the function f is as described in LZW-$\mathcal{FP}$ algorithm. Hence for almost all input strings, FPA constructs an entirely different dictionary with that of LZW-$\mathcal{FP}$. The codeword of the phrase $T[b_m : f(b_m)]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. FPA again uses flexible parsing: given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$. Please see the appendix for an example execution of the decompression algorithm for LZW-$\mathcal{FP}$.

3 Data Structures and Implementations of Algorithms

In this section we describe both the trie-reverse-trie data structure, and the new fingerprints based data structure for efficient on-line implementations of the LZW-$\mathcal{FP}$, and FPA methods. The trie-reverse-trie pair is a deterministic data structure, and hence guarantees a worst case linear running time for both algorithms as described in [MS99]). The new data structure based on *fingerprints* [KR87], is randomized, and guarantees an average case linear running time for the algorithm.

The two main operations to be supported by these data structures are (1) insert a phrase to $\mathcal{D}$ (2) search for a phrase, i.e., given a substring S, check whether it is in $\mathcal{D}$. The standard data structure used in many compression algorithms including LZW, the compressed trie $\mathcal{T}$ supports both operations in time proportional to $|S|$. A compressed trie is a rooted tree with the following properties: (1) each node with the exception of the root represents a dictionary phrase; (2) each edge is labeled with a substring of characters; (3) the first characters of two sibling edges can not be identical; (4) the concatenation of the substrings of the edges from the root to a given node is the dictionary phrase represented by that node; (5) each node is labeled by the codeword corresponding to its phrase. Dictionaries with prefix properties, such as the ones used in LZW and LZ78 algorithms, build a regular trie rather than a compressed one. The only difference is that in a regular trie the substrings of all edges are one character long.

In our data structures, inserting a phrase S to $\mathcal{D}$ takes $O(|S|)$ time as in the case of a trie. Similarly, searching S takes $O(|S|)$ time if no information about substring S is provided. However, once it is known that S is in $\mathcal{D}$, searching strings obtained by concatenating or deleting characters to/from both ends of S takes only $O(1)$ time. More precisely, our data structures support two operations *extend* and *contract* in $O(1)$ time. Given a phrase S in $\mathcal{D}$, the operation extend(S, a) for a given character a, finds out whether the concatenation of S and a is a phrase in $\mathcal{D}$. Similarly, the operation contract(S), finds out whether the suffix $S[2 : |S|]$ is in $\mathcal{D}$. Notice that such operations can be performed in a suffix tree, if the phrases in $\mathcal{D}$ are all the suffixes of a given string as in the case of the LZ77 algorithm [RPE81]. For arbitrary dictionaries (such as the ones built by LZW) our data structures are unique in supporting contract and extend operations in $O(1)$ time, and insertion operation in time linear with the size of the phrase, while using $O(|\mathcal{D}|)$ space, where $|\mathcal{D}|$ is the number of phrases in $\mathcal{D}$.

Trie-reverse-trie-pair data structure. Our first data structure builds the trie, $\mathcal{T}$, of phrases as described above. In addition to $\mathcal{T}$, it also constructs $\mathcal{T}^r$, the compressed trie of the *reverses* of all phrases inserted in the $\mathcal{T}$. Given a string $S = s_1, s_2, \ldots, s_n$, its reverse S^r is the string $s_n, s_{n-1}, \ldots, s_2, s_1$. Therefore for each node v in $\mathcal{T}$, there is a corresponding node v^r in $\mathcal{T}^r$ which represents the reverse of the phrase represented by v. As in the case of the $\mathcal{T}$ alone, the insertion of a phrase S to this data structure takes $O(|S|)$ time. Given a dictionary phrase S, and the node n which represents S in $\mathcal{T}$, one can find out whether the substring obtained by concatenating S with any character a in is $\mathcal{D}$, by checking out if there is an edge from n with corresponding character a; hence extend operation takes $O(1)$ time. Similarly the contract operation takes $O(1)$ time by going from n to n', the node representing reverse of S in $\mathcal{T}^r$, and checking if the parent of n' represents $S[2 : |S|]^r$.

Fingerprints based data structure. Our second data structure is based on building a hash table H of size p, a suitably large prime number. Given a phrase $S = S[1 : |S|]$, its location in H is computed by the function h, where $h(S) = \ (s[1]|\Sigma|^{|S|} + s[2]|\Sigma|^{|S|-1} + \ldots + s[|S|]) \ mod \ p$, where $s[i]$ denotes the lexicographic order of $S[i]$ in Σ [KR87]. Clearly, once the values of $|\Sigma|^k \ mod \ p$ are calculated for all k up to the maximum phrase size, computation of $h(S)$, takes $O(|S|)$ time. By taking p sufficiently large, one can decrease the probability of a collision on a hash value to some arbitrarily small $1/\epsilon$ value; thus the average running time of an insertion would be $O(|S|)$

as well. Given the hash value $h(S)$ of a string, the hash value of its extension by any character a can be calculated by $h(Sa) = (h(S)|\Sigma| + lex(a))\ mod\ p$, where $lex(a)$ is the lexicographic order of a in Σ. Similarly, the hash value of its suffix $S[2 : |S|]$ can be calculated by $h(S[2 : |S|]) = (h(S) - s[1]|\Sigma|^{|S|})\ mod\ p$. Both operations take $O(1)$ time.

In order to verify if the hash table entry $h(S)$ includes S in $O(1)$ time we (1) give unique labels to each of the phrases in $\mathcal{D}$, and (2) in each phrase S in H, store the label of the suffix $S[2 : |S|]$ and the label of the prefix $S[1 : |S| - 1]$. The label of newly inserted phrase can be $|\mathcal{D}|$, the size of the dictionary. This enables both extend and contract operations to be performed in $O(1)$ time on the average: suppose the hash value of a given string S is h, and the label of S is ℓ. To extend S with character a, we first compute the hash value h' of the string Sa. Among the phrases whose hash value is h', the one whose prefix label matches the label of S gives the result of the extend operation. To contract S, we first compute the hash value h'' of the string $S[2 : |S|]$. Among the phrases whose hash value is h'', the one whose label matches the suffix label of S gives the result of the contract operation. Therefore, both extend and contract operations take expected $O(1)$ time.

Inserting a phrase in this data structure can be performed as follows. An insert operation is done only after an extend operation on some phrase S (which is in $\mathcal{D}$) with some character a. Hence, when inserting the phrase Sa in $\mathcal{D}$ its prefix label is already known: the label of S. Once it is decided that Sa is going to be inserted, we can spend $O(|S| + 1)$ time to compute the suffix label of Sa. In case the suffix $S[2 : |S|]a$ is not a phrase in $\mathcal{D}$, we temporarily insert an entry for $S[2 : |S|]a$ in the hash table. This entry is then filled up when $S[2 : |S|]$ is actually inserted in $\mathcal{D}$. Clearly, the insertion operation for a phrase R takes expected $O(|R|)$ time.

A linear time implementation of LZW-$\mathcal{FP}$. For any input T LZW-$\mathcal{FP}$ inserts to $\mathcal{D}$ the same phrases with LZW. The running time for insertion in both LZW and LZW-$\mathcal{FP}$ (via the data structures described above) are the same; hence the total time needed to insert all phrases in LZW-$\mathcal{FP}$ should be identical to that of LZW, which is linear with the input size. Parsing with $\mathcal{FP}$ consists of a series of extend and contract operations. We remind that: (1) the function f on characters of T is described as $f(i) = \ell$ where $T[i : \ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$. (2) given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$. In order to compute b_{m+1}, we inductively assume that $f(b_m)$ is already computed. Clearly $S = T[b_m : f(b_m)]$ is in $\mathcal{D}$ and $S' = T[b_m : f(b_m)]$ is not in $\mathcal{D}$. We then contract S by i characters, until $S' = T[b_m + i : f(b_m) + 1]$ is in $\mathcal{D}$. Then we proceed with extensions to compute $f(b_m + i)$. After subsequent contract and extends we stop once $i > f(b_m)$. The last value of i at which we started our final round of contracts is the value b_{m+1}. Notice that each character in T participates to exactly one extend and one contract operation, each of which takes $O(1)$ time via the data structures described above. Hence the total running time for the algorithm is $O(n)$.

A linear time implementation of FPA. Parsing in FPA is done identical to LZW-$\mathcal{FP}$ and hence takes $O(n)$ time in total. The phrases inserted in $\mathcal{D}$ are of the form $T[b_m : f(b_m) + 1]$.

Because in parsing step m, the phrase $T[b_m : f(b_m)]$ is already searched for, it takes only $O(1)$ time per phrase to extend it via our data structures. Hence the total running time for insertions is $O(n)$ as well.

Linear time implementations of decompression algorithms for LZW-$\mathcal{FP}$ and FPA. The decompression algorithms for both methods simply emulate their corresponding compression algorithms, hence run in $O(n)$ time.

4 The Experiments

In this section we describe in detail the data sets we used, and discuss our test results verifying how well our theoretical expectations were supported.

4.1 The test programs

We used `gzip`, `compress`, LZW-$\mathcal{FP}$ and FPA programs for our experiments. The `gzip` and `compress` programs are standard features of UNIX operating system. In our LZW-$\mathcal{FP}$ implementation we limited the dictionary size to 2^{16} phrases, and reset it when it was full as in the case of `compress`; we also experimented with the extended version of LZW-$\mathcal{FP}$ which allows 2^{24} phrases. Similarly we experimented with two versions of FPA: one with 2^{16} and the other with 2^{24} phrases maximum.

4.2 The data sets

Our data sets come from three sources: (1) Data obtained via UNIX `drand48()` pseudorandom number generator. (2) DNA and protein sequences provided by Center for BioInformatics, University of Pennsylvania and CT and MR scans provided by the St. Thomas Hospital, UK [Sou]. (3) Text files from two data compression benchmark suites: the new Canterbury corpus and the commonly used Calgary corpus [Sou].

The first data set was designed to verify the theoretical convergence properties of the redundancy in the output of the algorithms and measure the constants involved. The second data set was designed to measure the performance of our algorithms for emerging bio-medical applications where no loss of information in data can be tolerated. Finally the third data set was chosen to demonstrate whether our algorithms are competitive with others in compressing text.

Specifically, the first data set includes three binary files generated by the UNIX `drand48()` function. The data distribution is i.i.d. with bit probabilities (1) $0.7 - 0.3$, (2) $0.9 - 0.1$, and (3) $0.97 - 0.03$. The second data set includes two sets of human DNA sequences from chromosome 23 (*dna1, dna2*), one MR (magnetic resonance) image of human (female) breast (*mr.pgm*), and one CT (computerized tomography) scan of a fractured human hip *ct.pgm* in uncompressed `pgm` format in ASCII [Sou]. The third set includes the complete Calgary corpus [3]. It also includes all files of size $> 1MB$ from the new Canterbury corpus: a DNA sequence from E-coli bacteria, *E.coli*, the complete bible *bib.txt* , and *world192.txt*.

4.3 Test results

In summary, we observed that LZW-$\mathcal{FP}$ and FPA implementations with maximum dictionary size 2^{24} performs the best on all types of files with size $> 1MB$ and shorter files with non-textual content. For shorter files consisting text, `gzip` performs the best as expected.

Also the theoretical expectations for the convergence rate in the redundancy of the output for i.i.d. data were verified. We observed that the constants involved in the convergence rate for FPA and LZW-$\mathcal{FP}$ were smaller than that of LZW, and `gzip` was asymptotically worse than all.

Our tests on the human DNA sequences with LZW-$\mathcal{FP}$ and FPA show similar improvements over *compress* and *gzip* - with a dictionary of maximum size 2^{16}, the improvement is about 1.5% and 5.7% respectively. Some more impressive results were obtained by increasing the dictionary size to 2^{24}, which further improved the compression ratio to 9%. The performance of LZW-$\mathcal{FP}$ and FPA on *mr* and *ct* scans differ quite a bit: LZW-$\mathcal{FP}$ was about $4\% - 6\%$ better than `compress` and was comparable to `gzip`; FPA's improvement was about 15% and 7% respectively. As the image files were rather short, we didn't observe any improvement by using a larger dictionary. One interesting observation is that the percentage improvement achieved by both FPA and LZW-$\mathcal{FP}$ increased consistently with increasing data size. This suggests that we can expect them to perform better in compressing massive archives as needed in many biomedical applications such as the human genome project.

Our tests on pseudorandom sequences verified our theoretical expectations. All LZW based schemes performed asymptotically better than `gzip`, which is based on LZ77. Our plots show that the redundancy in the output is indeed proportional to $1/\log n$ with the smallest constant achieved by FPA - in both cases, the constant is very close to 1.0; the constant for LZW-$\mathcal{FP}$ and LZW are about 1.5 and 2.0 respectively. This suggests that for on-line entropy measurement, FPA provides a more reliable alternative to LZ78/LZW or LZ77 (see [FNS$^+$95] for applications of LZW and LZ77 for entropy measurement in the context of DNA sequence analysis).

Our results on text strings varied depending on the type and size of the file compressed. For short files with long repetitions, `gzip` is still the champion. However, for all text files of size $> 1MB$, the large dictionary implementation of FPA scheme outperforms `gzip` by $4.7\% - 8.5\%$, similar to the tests for DNA sequences.

We believe that this curiosity can be explained as follows. When compressing English text, LZW needs process a sufficiently long prefix of the input to build an effective dictionary that captures its information content. Assuming that the average word (or word combination) size in English which is likely to repeat is about 6, and assuming that the effective alphabet size in English is again 6

[3] Which includes a bibliography file, *bib*, two complete books, *book1, book2*, two binary files *geo, pic*, programs in `c, lisp, pascal` languages *progc, progl, progp*, and the transcript of a login session, *trans*

(derived from the empirical entropy estimates of about 2.5 bits per character), LZW requires to process a prefix of $6 * 6^6 = 273K$ characters from the input before it starts to be competitive with LZ77 approach. After this point, LZ78 may start to show its better asymptotic properties in terms of redundancy - as suggested by our experiments. Hence limiting the dictionary size in LZW to $2^{16} << 273K$ as in the case of LZW does not give the chance to LZW to perform as good as or better than the LZ77 based approaches.

File	*Size* (KB)	*gzip* (KB)	*compress* (KB)	*LZW-FP* $\uparrow_g$ (%)	*LZW-FP* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FP-24* $\uparrow_g$ (%)	*FP-24* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
bib	109	34	45	-26.01	5.04	-19.69	9.80	-26.01	5.04	-19.69	9.80
book1	751	306	324	-3.81	2.03	-2.48	3.29	2.45	7.94	3.94	9.34
book2	597	202	244	-16.61	3.89	-12.32	7.42	-11.10	8.43	-7.47	11.42
geo	100	67	76	-11.90	1.46	-11.66	1.67	-11.90	1.46	-11.66	1.67
pic	501	55	61	-6.64	3.25	-5.31	4.47	-6.64	3.26	-5.31	4.47
progc	39	13	19	-37.25	4.85	-33.40	7.52	-37.23	4.86	-33.40	7.52
progl	70	16	26	-56.83	5.99	-49.29	10.51	-56.82	6.00	-49.29	10.51
progp	48	11	19	-59.70	6.54	-53.75	10.02	-59.69	6.54	-53.75	10.02
trans	91	18	37	-87.12	7.12	-73.96	13.65	-87.11	7.13	-73.96	13.65

Table 1: Compression evaluation using files in the Calgary corpus
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, *FP-24*, and *FPA-24* over *gzip* and *compress*

File	*Size* (KB)	*gzip* (B)	*compress* (B)	*LZW-FP* $\uparrow_g$ (%)	*LZW-FP* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FP-24* $\uparrow_g$ (%)	*FP-24* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
E.coli	4530	1341245	1255647	6.91	0.56	6.43	0.05	8.84	2.63	8.48	2.24
bible.txt	3953	1191063	1401885	-12.87	4.11	-7.79	8.42	0.13	15.15	4.68	19.01
world192.txt	2415	724595	987035	-31.70	3.32	-20.36	11.64	-2.38	24.84	6.54	31.39

Table 2: Compression evaluation using files in the Canterbury corpus (Large Set)
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, *FP-24*, and *FPA-24* over *gzip* and *compress*

References

[BCW90] T. Bell, T. Cleary, and I. Witten. *Text Compression.* Academic Press, 1990.

[FNS+95] M. Farach, M. Noordeweir, S. Savari, L. Shepp, A. J. Wyner, and J. Ziv. The entropy of DNA: Algorithms and measurements based on memory and rapid convergence. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[GSS85] M. E. Gonzales-Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, 32(2):344–373, April 1985.

File	*Size* (KB)	*gzip* (KB)	*compress* (KB)	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
				$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
dna1	3096	977	938	5.59	1.54	5.75	1.70	8.73	4.82	8.91	5.00
dna2	2877	846	813	4.64	0.75	4.33	0.43	6.09	2.26	5.89	2.05
mr.pgm	260	26	29	-7.23	3.60	6.38	15.84	-7.22	3.61	6.38	15.84
ct.pgm	1039	110	110	4.10	3.61	14.56	14.12	4.10	3.61	14.56	14.12

Table 3: Compression evaluation using experimental biological and medical data
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, *FP-24*, and *FPA-24* over *gzip* and *compress*

[Hor95] R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *IEEE Data Compression Conference*, 1995.

[JS95] P. Jacquet and W. Szpankowski. Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees. *Theoretical Computer Science*, (144):161–197, 1995.

[KM97] S. R. Kosaraju and G. Manzini. Some entropic bounds for Lempel-Ziv algorithms. In *Sequences*, 1997.

[KR87] R. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[LS95] G. Louchard and W. Szpankowski. Average profile and limiting distribution for a phrase size in the Lempel-Ziv parsing algorithm. *IEEE Transactions on Information Theory*, 41(2):478–488, March 1995.

[MS99] Y. Matias and S. C. Sahinalp. On optimality of parsing in dynamic dictionary based data compression. ACM-SIAM Symposium on Discrete Algorithms, 1999.

[MW85] V.S. Miller and M.N. Wegman. Variations on a theme by Lempel and Ziv. *Combinatorial Algorithms on Words*, pages 131–140, 1985.

[RPE81] M. Rodeh, V. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

[Sav97] S. Savari. Redundancy of the Lempel-Ziv incremental parsing rule. In *IEEE Data Compression Conference*, 1997.

[Sou] http://www.dcs.warwick.ac.uk/people/research/Nasir.Rajpoot/work/fp/index.html.

[Sto88] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, January 1984.

File	*Size* (*KB*)	*gzip* (*B*)	*compress* (*B*)	*LZW-$\mathcal{FP}$*		*FPA*		*FP-24*		*FPA-24*	
				$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
P(0)=0.7	1	205	208	4.88	6.25	4.88	6.25	4.88	6.25	4.88	6.25
P(1)=0.3	10	1606	1524	7.41	2.43	9.40	4.53	7.41	2.43	9.40	4.53
	100	15713	13481	15.60	1.62	17.89	4.29	15.61	1.64	17.89	4.29
	1024	160206	131692	18.49	0.84	20.49	3.28	18.47	1.14	20.69	3.52
	2048	**320186**	**263659**	**18.53**	**1.07**	**20.46**	**3.41**	**19.44**	**2.17**	**21.20**	**4.31**
P(0)=0.9	1	129	134	3.10	6.72	9.30	12.69	3.10	6.72	9.30	12.69
P(1)=0.1	10	1034	923	14.89	4.66	16.92	6.93	14.89	4.66	16.92	6.93
	100	9740	7750	22.73	2.89	25.86	6.83	22.74	2.90	25.86	6.83
	1024	100080	74147	27.55	2.21	30.44	6.11	27.55	2.22	30.44	6.11
	2048	**199732**	**146630**	**28.10**	**2.07**	**30.95**	**5.95**	**28.43**	**2.50**	**31.20**	**6.28**
P(0)=0.97	1	93	99	6.45	12.12	6.45	12.12	6.45	12.12	6.45	12.12
P(1)=0.03	10	530	518	7.74	5.60	11.51	9.46	7.74	5.60	11.51	9.46
	100	4623	3754	22.39	4.42	28.03	11.37	22.41	4.45	28.03	11.37
	1024	45829	33292	29.83	3.40	34.64	10.02	29.83	3.41	34.64	10.02
	2048	**91410**	**64813**	**31.30**	**3.10**	**35.79**	**9.44**	**31.30**	**3.11**	**35.79**	**9.44**

Table 4: Compression evaluation using independent identically distributed random files containing only zeros and ones with different probability distributions
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-$\mathcal{FP}$*, *FPA*, *FP-24*, and *FPA-24* over *gzip* and *compress*; random data generated by drand48().

[Wyn95] A. J. Wyner. *String Matching Theorems and Applications to Data Compression and Statistics.* Ph.D. dissertation, Stanford University, Stanford, CA, 1995.

[Yok92] H. Yokoo. Improved variations relating the Ziv-Lempel and welch-type algorithms for sequential data compression. *IEEE Transactions on Information Theory*, 38(1):73–81, January 1992.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

A Appendix

In this section we demonstrate how LZW, LZW-$\mathcal{FP}$ and FPA algorithms work by means of tables.

Iteration: h	$T[h]$	$\mathcal{D}_{h+1} \setminus D_h$	$c(h)$	Output
0	a	Φ	-1	Φ
1	b	$ab : 2$	0	$C(T[0:0]) = 0$
2	a	$ba : 3$	1	$C(T[1:1]) = 1$
3	b	Φ	1	Φ
4	a	$aba : 4$	3	$C(T[2:3]) = 2$
5	b	Φ	3	Φ
6	a	Φ	3	Φ
7	a	$abaa : 5$	6	$C(T[4:6]) = 4$
8	b	Φ	6	Φ
9	a	Φ	6	Φ
10	a	Φ	6	Φ
11	b	$abaab : 6$	10	$C(T[7:10]) = 5$
12	a	Φ	10	Φ
13	a	$baa : 7$	12	$C(T[11:12]) = 3$
14	a	$aa : 8$	13	$C(T[13:13]) = 0$
15	b	Φ	13	Φ
16	Φ	Φ	15	$C(T[14:15]) = 2$

Table 5: LZW compression algorithm for $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$. We have $\Sigma = \{a, b\}$, $\mathcal{D}_0 = \{a : 0, b : 0\}$, and $\mathcal{D}_{h+1} \setminus \mathcal{D}_h$ represent the phrases added to $\mathcal{D}$ in iteration h, with its corresponding code word. The output is: $C(T[0:15]) = 0, 1, 2, 4, 5, 3, 0, 2$.

Iteration: i	$T[i]$	$\mathcal{D}_{i+1} \setminus D_i$	$c(i)$	Output
0	a	Φ	-1	Φ
1	b	$ab : 2$	-1	Φ
2	a	$ba : 3$	0	$C(T[0:0]) = 0$
3	b	Φ	0	Φ
4	a	$aba : 4$	1	$C(T[1:1]) = 1$
5	b	Φ	1	Φ
6	a	Φ	1	Φ
7	a	$abaa : 5$	3	$C(T[2:3]) = 2$
8	b	Φ	3	Φ
9	a	Φ	3	Φ
10	a	Φ	3	Φ
11	b	$abaab : 6$	6	$C(T[4:6]) = 4$
12	a	Φ	6	Φ
13	a	$baa : 7$	6	Φ
14	a	$aa : 8$	9	$C(T[7:9]) = 4$
15	b	Φ	9	Φ
16	Φ	Φ	15	$C(T[10:13]) = 5, C(T[14:15]) = 2$

Table 6: LZW-$\mathcal{FP}$ algorithm: We demonstrate how the flexible parser $\mathcal{P}_o$ can be used with the dictionary parser $\mathcal{P}_d$ of LZW As in the example of Table 7, let $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$, $\Sigma = \{a, b\}$ and $\mathcal{D}_0 = \{a : 0, b : 0\}$. The output of LZW dictionary with Flexible Parsing: $C(T[0 : 15]) = 0, 1, 2, 4, 4, 5, 2$. Comparing the to the output of LZW algorithm in Table 7, we observe that the $\mathcal{FP}$ output is one codeword less (7 compared to 8), due to its more efficient parsing of $T[7 : 15]$. Specifically, by parsing a shorter prefix of $T[7 : 15]$ ($T[7 : 9]$ rather than $T[7 : 10]$), $\mathcal{FP}$ parses a much larger string, $T[10 : 13]$ later, which is represented by two codewords by greedy parsing.

LZW parsing

Input: a b a b a b a a b a a b a a a b

LZW Output: 0 1 2 4 5 3 0

LZWFP parsing

Input: a b a b a b a a b a a b a a a b

LZWFP Output: 0 1 2 4 4 5 2

Figure 1: Comparsion of $\mathcal{FP}$ and greedy parsing when used together with the LZW dictionary construction method, on the input string $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$, used in Tables 7 and 8.

Iteration: i	*Input*	$\mathcal{D}_{i+1} \setminus D_i$	Output
0	0	Φ	a
1	1	$ab : 2$	b
2	2	$ba : 3$	ab
3	4	$aba : 4$	aba
4	4	$abaa : 5$	aba
5	5	$abaab : 6, baa : 7$	$abaa$
6	2	$aa : 8$	ab

Table 7: Decompression of LZW-$\mathcal{FP}$ algorithm: We demonstrate the execution of our decompression algorithm on the output of the compression algorithm in the previous example, $C(T[0:15]) = 0, 1, 2, 4, 4, 5, 2$. We show how our decompression algorithm obtains the original input to the compression. Initially $i = j = 0$, and the dictionary consists of characters a and b with respective codewords 0 and 1. $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$. Note that the decompression at iteration 3 is possible as the algorithms knows the prefix of 4 which is necessarily ab. Hence it outputs ab automatically and re-emulates the dictionary parser.

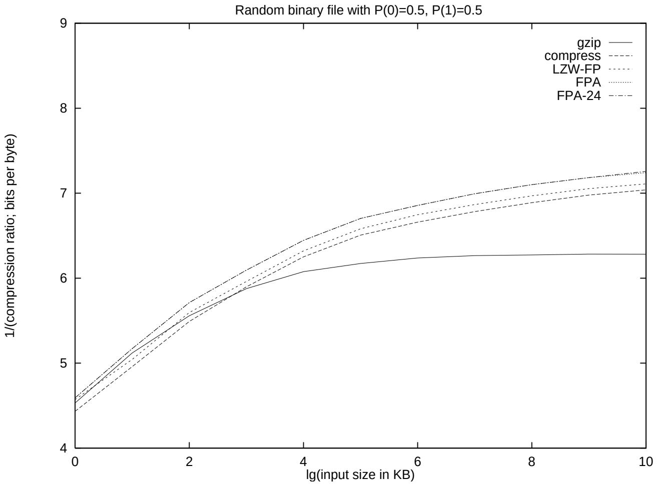

Figure 2: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = P(1) = .5$.

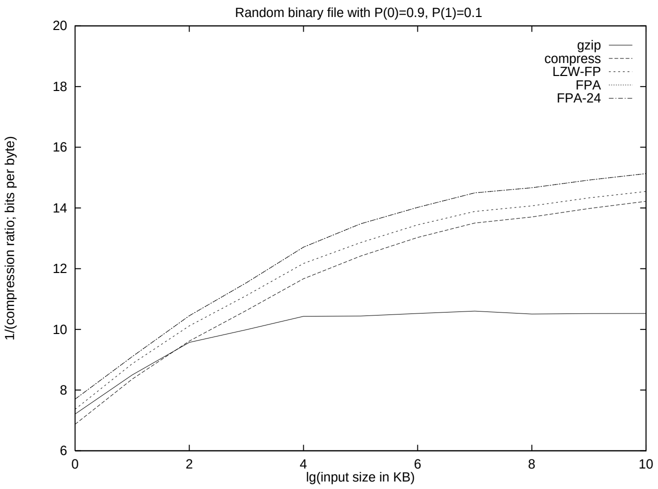

Figure 3: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = .9$ and $P(1) = .1$.

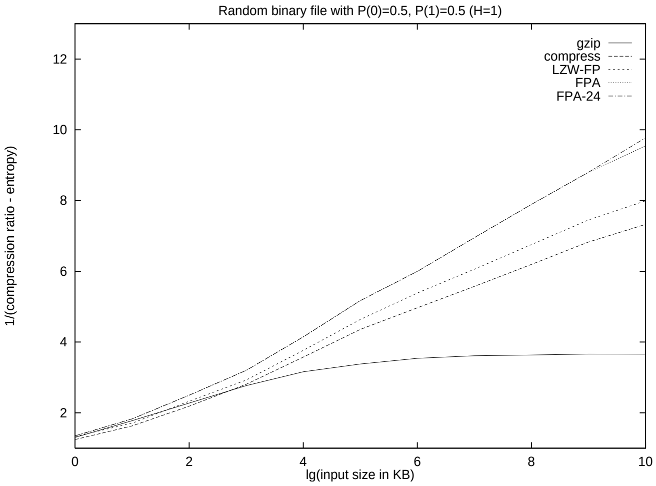

Figure 4: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = P(1) = .5$.

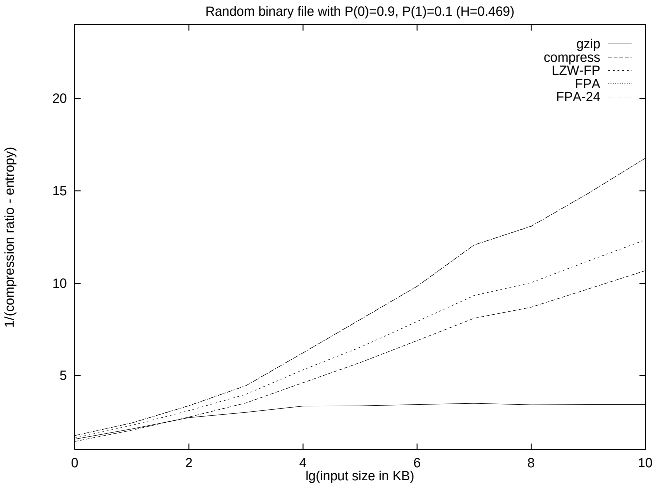

Figure 5: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = .9$ and $P(1) = .1$.