The Effect of Flexible Parsing for Dynamic Dictionary Based Data Compression

Yossi Matias* Nasir Rajpoot† Süleyman Cenk Ṣahinalp‡

Abstract

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 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.

We focus on the application of $\mathcal{FP}$ in the context of the LZW variant of the Lempel-Ziv'78 dictionary construction method [Wel84, ZL78], which is of considerable practical interest. We implement two compression algorithms which use (1) $\mathcal{FP}$ with LZW dictionary (LZW-$\mathcal{FP}$), and (2) $\mathcal{FP}$ with an alternative flexible dictionary (FPA as introduced in [Hor95]). Our implementations are based on novel on-line data structures enabling us to use linear time and space. 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 of the most popular compression programs `UNIX compress` and `gzip`. Our results demonstrate that flexible parsing is especially useful for non-textual data, on which it improves over the compression rates of `compress` and `gzip` by up to 20% and 35%, respectively.

*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; `Nasir.Rajpoot@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 has recently been 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

[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].

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 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. Our implementations are based on a new data structure based on Karp-Rabin fingerprints [KR87]. We first 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 by 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. We then 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 outputs of each of the four pro-

[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

grams approaches 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 are 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].

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 fits 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, 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 possible one-character substrings. 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 ones, 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 $|\Sigma| + m$. Thus, the codewords of substrings do not change in LZW algorithm. LZW uses greedy parsing as well. Hence, no two phrases can be identical in the LZW algorithm.

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

Table 1: LZW compression algorithm for $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$. We have $\Sigma = \{a, b\}$, $\mathcal{D}_0 = \{a:0, b:1\}$, and $\mathcal{D}_{i+1} \setminus \mathcal{D}_i$ represent the phrases added to $\mathcal{D}$ in iteration i, with its corresponding codeword. The output is: $C(T[0:16]) = 0, 1, 2, 4, 5, 3, 0, 2$.

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 j for which $f(j)$ is the maximum among all j such that $T[b_m : j-1]$ is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[j-1]$.

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

Table 2: LZW-$\mathcal{FP}$ algorithm: We demonstrate how the flexible parser can be used with the dictionary construction of LZW . As in the example of table 2, let $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$, $\Sigma = \{a, b\}$ and $\mathcal{D}_0 = \{a : 0, b : 1\}$. The output of LZW dictionary with flexible parsing: $C(T[0 : 15]) = 0, 1, 2, 4, 4, 5, 2$. Comparing the output of LZW algorithm in table 2, we observe that the $\mathcal{FP}$ output is one codeword less (7 compared to 8), due to its more efficient parsing of $T[8 : 16]$. Specifically, by parsing a shorter prefix of $T[8 : 16]$ ($T[8 : 10]$ rather than $T[8 : 11]$), $\mathcal{FP}$ parses a much larger string, $T[11 : 14]$, later, which is represented by two codewords by greedy parsing.

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

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
1	0	{}	a
2	1	$ab : 2$	b
3	2	$ba : 3$	ab
4	4	$aba : 4$	aba
5	4	$abaa : 5$	aba
6	5	$abaab : 6, baa : 7$	$abaa$
7	2	$aa : 8$	ab

Table 3: 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 $i = j = 1$, 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 4 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.

again uses flexible parsing.

We note that the codewords resulting from all of the compression algorithms described in this section are encoded using Welch type variable-length codes. The length of these codes is essentially equal to $\lceil \log |\mathcal{D}| \rceil$, where $|\mathcal{D}|$ is number of phrases in the dictionary at the time of outputting the codeword. Equipeed with the correct number of dictionary entries at each step, the corresponding decompression algorithm can uniquely determine which codeword it receives. It is possible to use one form of *entropy coding* such as dynamic Huffman coding or arithmetic coding to exploit the frequency distribution of codewords to achieve further compression. However, because of the fact that such coding schemes are usually slow to update and that the usefulness of such coding schemes are not very clear in this context, we refrained from using any entropy coding scheme in our implementations.

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 Karp and Rabin's *fingerprinting* method [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 in $\mathcal{D}$, i.e. given a substring S, check whether it is in $\mathcal{D}$. The standard data structure used in many compression algorithms including LZW is the compressed trie $\mathcal{T}$, which 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 the following operations in $O(1)$ time: (1) *extend* a phrase, and (2) *contract* a phrase. 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 als h, where o 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 h, where

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 consists of a linear hash table H with p entries, where p is a randomly chosen prime number in the range $\{1, 2, \ldots, M\}$. Given a phrase $S = S[1:|S|]$, we define $h(S)$, its hash table entry, as $h(S) = (s[1]|\Sigma|^{|S|-1} + s[2]|\Sigma|^{|S|-2} + \ldots + s[|S|])\ mod\ p$; here $s[i]$ stands for the lexicographic order of $S[i]$ in the alphabet Σ [KR87]. We upper bound the probability of a collision on any given hash table entry with some arbitrarily small value ϵ, by selecting an appropriately large M.

Given string S whose hash value $h(S)$ is already known, we can compute the hash value of its extension by any character a, with lexicographic order $lex(a)$, as $h(Sa) = (h(S)|\Sigma| + lex(a))\ mod\ p$, in $O(1)$ time. Thus, starting with $h(S[1])$, we can iteratively compute $h(S[1:2]), h(S[1:3]), \ldots, h(S)$ in $O(1)$ time each; hence the computation of $h(S)$ takes $O(|S|)$ time. To compute the hash value of the contraction of S in $O(1)$ expected time, we pre-compute $|\Sigma|^k\ mod\ p$ for all k up to the maximum phrase size. The hash value of $S[2:|S|]$ can then be computed as $h(S[2:|S|]) = (h(S) - s[1]|\Sigma|^{|S|-1})\ mod\ p$.

An important feature of our data structure is that it can verify whether a given hash table entry k actually represents a string S for which $h(S) = k$. To achieve this, we give a unique label $l(S)$ to each phrase S in $\mathcal{D}$ [3], and for each such phrase S, we store the label of its suffix $S[2:|S|]$ and the label of its prefix $S[1:|S|-1]$.

Our data structure performs extend and contract operations in expected $O(1)$ time. Let S be a string whose label $l(S)$ and hash value $h(S)$ are known. To compute the extension of S with character a, we first compute the hash value $h(Sa)$ as described above. Among the phrases whose hash value is $h(Sa)$, 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

[3] This can be achieved by setting $l(S) = |\mathcal{D}|$, at the time of its insertion

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.

Let f be a function on characters of T such that $f(i)=\ell$ for which $T[i:\ell]$ is the longest phrase in $\mathcal{D}$ that starts at $T[i]$. Let $b_0=1$, and define b_m inductively as the integer j for which $f(j)$ is the maximum among all j such that $T[b_{m-1}:j-1]$ is in $\mathcal{D}$.

LZW-$\mathcal{FP}$ works in several steps. In a given step $m+1$ there are a number of iterations. Each iteration comprises of a series of contractions followed by a series of extensions. For the first iteration consider $S=T[b_m:f(b_m)]$ and $S'=T[b_m:f(b_m)+1]$. By definition, S is in $\mathcal{D}$ and S' is not in $\mathcal{D}$. We contract S' character by character until it (i.e. $T[b_m+i:f(b_m)+1]$) is in $\mathcal{D}$. Then we obtain the longest phrase in $\mathcal{D}$ that starts at $T[b_m+i]$ by performing a series of extensions. This gives $T[b_m+i:f(b_m+i)]$ which we set to be S. We then set $S'=T[b_m+i:f(b_m+i)+1]$ and proceed with the next iteration. The step $m+1$ is complete once $b_m+i>f(b_m)$. The last value of b_m+i at which the last round of contracts were made gives the value b_{m+1}.

Running time: 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 $O(n)$.

Correctness: In a given iteration of step $m+1$, the algorithm checks whether subsequent locations in the last parsed phrase provides a further reach to the longest reaching phrase already known (we remind that each prefix of the last parsed phrase is guaranteed to be in the dictionary). If it does, then by subsequent extensions, this reach is pushed further. If it does not, then the a contact operation is performed to check whether the next character provides a further reach. Hence, a given step correctly computes the longest reaching phrase that starts at one of the characters of the phrase computed in the previous step. Notice that the prefix property of the dictionary is key to the correctness of the algorithm; every prefix of the last parsed phrase is also in the dictionary.

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, London, UK [Sou]. (3) Text files from two data compression benchmark suites: the new Canterbury corpus and the commonly used Calgary corpus (these commonly available data sets are also provided at [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. Each data entry is represented as a byte, although it can only take the values 0 or 1 [4]. 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 [5]. 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

The test results for our pseudorandom and biomedical data sets as well as files from the Calgary corpus and Canterbury corpus are presented in Tables 4, 5, 6, 7 respectively. The tables compare the size of the compressed files obtained by our algorithms to those obtained by `gzip` and `compress` by reporting the improvement as % of the uncompressed file size.[6]

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

We also verified the theoretical expectations for the convergence rate in the redundancy of the output for i.i.d. data. 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* test images differ quite a bit: LZW-$\mathcal{FP}$ was up to 4% better than `compress` when compared with `gzip` it was around 5% better, with the exception of the `mr.pgm` file on which `gzip` performed better. FPA's improvement

[4] This is for compatibility with `compress` and `gzip` whose initial alphabet sizes can not be altered. Thus although these compression algorithms have 256 in their dictionaries, such data lets them insert strings consisting of characters 0 and 1.

[5] 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*

[6] The improvement over `gzip` is denoted by $\uparrow$ g and the improvement over `compress` is denoted by $\uparrow$ c; a negative % improvement indicates a weaker performance than `gzip` or `compress`.

was about 6% and 16% over `gzip` and `compress`, respectively, for the MR image, whereas its improvement was about 14% over both schemes for the CT image. 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 when the dictionary size was increased from 2^{16} to 2^{24} [7]. This suggests that we can expect the algorithms to perform better in compressing massive archives by relaxing the bound on the maximum dictionary size. This is very useful in many biomedical applications, particularly 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. We note that the x axis in the plots denote the number of input bits read by each of the compression algorithms in logarithmic scale with base 1.1. The y axis denote $1/(\textit{source entropy} - \textit{output entropy})$, where the entropy for the binary source is defined to be $-(Prob(0) \cdot \log Prob(0) + Prob(1) \cdot \log Prob(1))$, and the output entropy is the average number of bits output for every input bit, between the last data point and the current one. The 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 (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 (i.e. the number of phrases in the dictionary) in LZW to $2^{16} << 273K$ as in the case of UNIX `compress` does not give the

[7] The files for which the compression achieved with maximum dictionary size of 2^{16} is equal to that with maximum dictionary size of 2^{24}, used less than 2^{16} phrases.

chance to LZW to perform as good as or better than the LZ77 based approaches.

In summary, our test results demonstrate that the compression performance of LZW-$\mathcal{FP}$ and FPA schemes on data of practical interest are considerable. We hope that our results will encourage program developers to work on efficient implementations of our algorithms. In table 4.3, we provide some preliminary timing results in which we compare the running times of our implementations with that of `compress` and `gzip` on some large data files. The implementations we report on are as follows: (1) LZW is the standard trie implementation, which we use as the basis of all other implementations, (2) LZW-$\mathcal{FP}$ is a trie based implementation, (3) FPA-Direct is again trie based, and (4) FPA-New is based on fingerprints. The first observation we make is `compress` is about 7 times faster than our LZW-$\mathcal{FP}$ implementation. This is not surprising as `compress` is a very popular program involving thousands of lines of code that has been under development for many years. Our LZW implementation compares well with `gzip`. When we compare our flexible parsing based implementations to our LZW implementation, we again see about a factor of 7 difference. Part of this (possibly up to a factor of 4) is due to more complex search involved in flexible parsing; we believe that a more efficient search routine can decrease this gap. By incorporating the implementation of the trie based data structure used in `compress` we may be able to further improve the running time of all flexible parsing based programs.

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
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
news	368	141	177	-22.43	2.63	-19.22	5.18	-17.30	6.71	-14.42	9.00
obj1	21	10	13	-33.32	2.03	-31.43	3.42	-33.32	2.03	-31.43	3.42
obj2	241	79	125	-48.95	5.50	-41.87	9.99	-48.95	5.50	-41.87	9.99
paper1	51	18	24	-28.78	4.60	-26.14	6.56	-28.78	4.60	-26.14	6.56
paper2	80	29	35	-16.96	3.77	-14.66	5.66	-16.95	3.77	-14.66	5.66
paper3	45	17	21	-18.23	3.46	-16.79	4.64	-18.23	3.47	-16.79	4.64
paper4	12	5	6	-21.59	3.25	-20.92	3.78	-21.59	3.26	-20.92	3.78
paper5	11	4	6	-27.41	3.28	-25.93	4.41	-27.40	3.31	-25.93	4.41
paper6	37	12	18	-34.96	4.48	-32.38	6.31	-34.96	4.49	-32.38	6.31
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	-37.25	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 4: 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* ($\uparrow$ g) and *compress* ($\uparrow$ c).

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(*KB*)	(*B*)	(*B*)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
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 5: Compression evaluation using files in the Canterbury corpus (Large Set)
The original file size, compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, *FP-24*, and *FPA-24* over *gzip* ($\uparrow$ g) and *compress* ($\uparrow$ c).

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(*KB*)	(*KB*)	(*KB*)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
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 6: 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*, em FP-24, and *FPA-24* over *gzip* ($\uparrow$ g) and *compress* ($\uparrow$ c).

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.

[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.

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FP-24*		*FPA-24*	
	(*KB*)	(*B*)	(*B*)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
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 7: 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-FP*, *FPA*, em FP-24, and *FPA-24* over *gzip* ($\uparrow$ g) and *compress* ($\uparrow$ c).

[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/ nasir/work/fp/index.html.

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

File		*LZW*	*LZW-FP*	*FPA*		*compress*	*gzip*
Description	*Size*(KB)	(*secs*)	(*secs*)	*Direct*	*New*	(*secs*)	(*secs*)
(0.7,0.3)	2048	8.5	104.7	69.8	85.4	1.5	10.3
(0.9,0.1)	2048	7.6	188.9	126.5	178.7	1.4	4.9
dna1	3096	12.7	67.1	47.9	65.2	1.7	17.2
dna2	2877	12.1	77.1	59.4	75.5	1.5	16.2
E.Coli	4530	13.4	83.9	68.8	81.2	2.0	26.7
bible.txt	3953	13.3	72.6	56.1	81.3	2.1	8.7
world192.txt	2415	9.3	42.7	33.4	50.2	1.7	3.9

Table 8: Timing results for large data files.
File description and its size, times taken to compress by *LZW*, LZW-$\mathcal{FP}$, two different implementations of FPA, `compress`, and `gzip`. The first three files are psuedorandom binary IID files in nature and are described by (x, y), where $x = p(0)$ and $y = p(1)$.

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

[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.

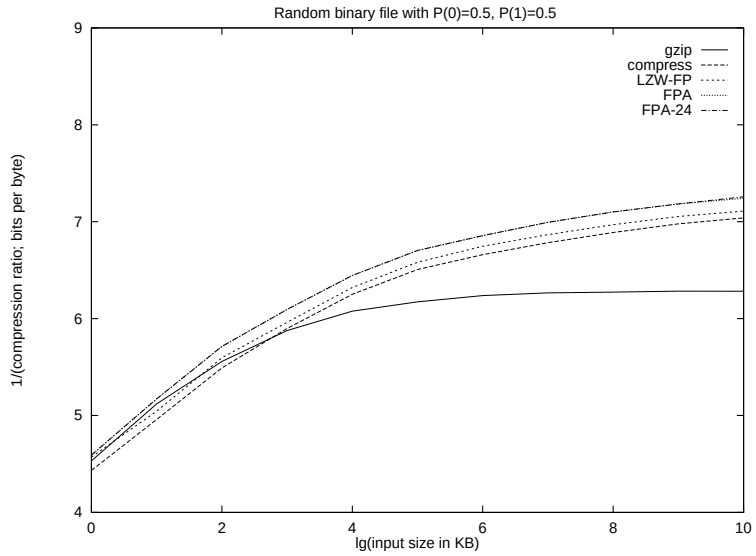

Figure 2: The compression ratio (i.e. output entropy) attained by all five programs on random i.i.d.data with bit probabilities $P(0) = P(1) = .5$.

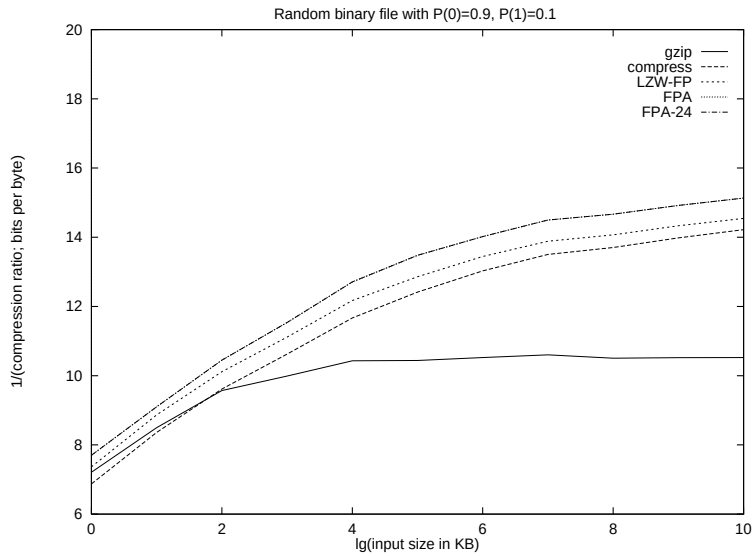

Figure 3: The compression ratio (i.e. output entropy) attained by all five programs on random i.i.d.data with bit probabilities $P(0) = .9$ and $P(1) = .1$.

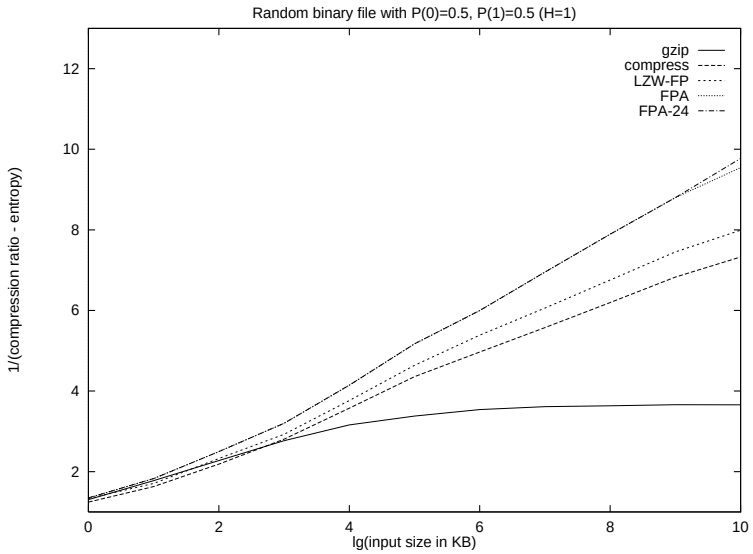

Figure 4: The 1/redundancy of all five programs on random i.i.d.data where redundancy is described as (actual compression ratio)-(entropy). The bit probabilities are $P(0) = P(1) = .5$.

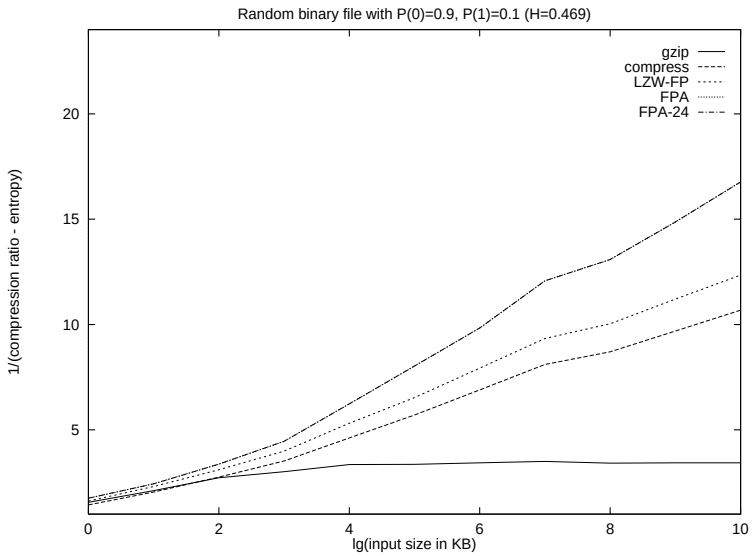

Figure 5: The 1/redundancy of all five programs on random i.i.d.data where redundancy is described as (actual compression ratio)-(entropy). The bit probabilities are $P(0) = .9$ and $P(1) = .1$.