Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 49–61

Implementation and Experimental Evaluation of Flexible Parsing for Dynamic Dictionary Based Data Compression

(extended abstract)

Yossi Matias [1]
Department of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel, and Bell Labs, Murray Hill, NJ, USA
e-mail: `matias@math.tau.ac.il`

Nasir Rajpoot [2]
Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK
e-mail: `nasir@dcs.warwick.ac.uk`

Süleyman Cenk Şahinalp [3]
Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK, and Center for BioInformatics, University of Pennsylvania, Philadelphia, PA, USA
e-mail: `cenk@dcs.warwick.ac.uk`

ABSTRACT

We report on the implementation and performance evaluation of greedy parsing with lookaheads for dynamic dictionary compression. Specifically, we consider the greedy parsing with a single step lookahead which we call Flexible Parsing ($\mathcal{FP}$) 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 does not necessarily achieve optimality with regard to the dictionary construction scheme in use. Flexible parsing, however, is optimal, 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 [MS98]. In this paper, we introduce a more practical, randomized data structure to implement $\mathcal{FP}$ scheme whose expected theoretical performance matches the worst case performance of the trie-reverse-trie-pair. We then report on the compression ratios achieved by two $\mathcal{FP}$ based compression programs we implemented. We test our programs against `compress` and `gzip` on various types of data on some of which we obtain up to 35% improvement.

1. Introduction

The size of data related to a wide range of applications is growing rapidly. Grand challenges such as the human genome project involve very-large distributed databases of text documents, whose effective storage and communication requires a major research and development effort. From DNA and protein sequences to medical images (in which any loss of information content can not

[1]partly supported by Alon Fellowship
[2]supported by Quaid-e-Azam scholarship from the Government of Pakistan
[3]partly supported by NATO research grant CRG-972175 and ESPRIT LTR Project no. 20244 - ALCOM IT

be tolerated) vital data sources that will shape the information infrastructure of the next century require simple and efficient tools for lossless data compression.

A (lossless) compression algorithm $\mathcal{C}$ reads input string T and computes an output string, T', whose representation is smaller than that of T, such that a corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$ can take T' as input and reconstruct T. 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. The task of partitioning the text into phrases is called *parsing* and the pointers replacing the phrases are called *codewords*.

A dictionary can be constructed in static or dynamic fashion. In *static* schemes, the whole dictionary is constructed before the input is compressed. Most practical compression algorithms, however, use *dynamic* schemes, introduced by Ziv and Lempel [ZL77, ZL78], in which the dictionary is initially empty and is constructed incrementally: as the input is read, some of its substrings are chosen as dictionary phrases themselves. The dictionary constructed by most dynamic schemes (e.g., [ZL77, ZL78, Wel84, Yok92]) satisfy the *prefix property* for any input string: in any execution step of the algorithm, for any given phrase in the dictionary, all its prefixes are also phrases in the dictionary.

In this paper we focus only on the the two most popular dictionary based compression methods: LZ78 [ZL78], its LZW variant [Wel84], and LZ77 [ZL77]. A few interesting facts about LZ78 and LZ77:

- The LZW scheme is the basis for UNIX `compress` program, `gif` image compression format, and is used in the most popular fax and modem standards (V42bis). LZ77 algorithm is the basis for all `zip` variants.
  Both algorithms: (1) are asymptotically optimal in the information theoretic sense, (2) are efficient, with $O(1)$ processing time per input character, (3) require a single pass over the input, and (4) can be applied on-line.
- LZ78 (and the LZW) can be implemented by the use of simple trie data structure with space complexity proportional to the number of codewords in the output. In contrast, a linear time implementation of the LZ77 builds a more complex suffix tree in an on-line fashion, whose space complexity is proportional to the size of the input text [RPE81].
- It is recently shown that LZ78 (as well as LZW) approaches the asymptotic optimality faster than LZ77: 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].
- The practical performances of these algorithms vary however depending on the application. For example the LZ77 algorithm may perform better for English text, and the LZ78 algorithm may perform better for binary data, or DNA sequences. [4]

[4]A simple counting argument shows that there cannot exist a single dictionary construction scheme that is superior to other schemes for all inputs. If a compression algorithm performs well for one set of input strings, it is likely that it will not perform well for others. The advantage of one dictionary construction scheme over another can only apply with regard to restricted classes of input texts. Indeed, numerous schemes have been proposed in the scientific literature and implemented in software products, and it is expected that many more will be considered in the future.

Almost all dynamic dictionary based algorithms in the literature including the Lempel-Ziv methods ([ZL77, ZL78, Wel84, MW85, Yok92]) use *greedy parsing*, which takes the uncompressed suffix of the input and parses its longest prefix, which is a phrase in the dictionary. The next substring to be parsed starts where the currently parsed substring ends. Greedy parsing is fast and can usually be applied on-line, and is hence very suitable for communications applications. However, greedy parsing can be far from optimal for dynamic dictionary construction schemes [MS98]: 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.

For static dictionaries -as well as for the off-line version of the dynamic dictionary compression problem-, there are a number of linear time algorithms that achieve optimal parsing of an input string, provided that the dictionary satisfies the prefix property throughout the execution of the algorithm (see, for example, [FM95]). More recently, in [MS98], it was shown that it is possible to implement the one-step lookahead greedy parsing (or shortly flexible parsing -$\mathcal{FP}$) for the on-line, dynamic problem, in amortized $O(1)$ per character. This implementation uses space proportional to the number of output codewords. It is demonstrated that $\mathcal{FP}$ is optimal for dictionaries satisfying the prefix property in every execution step of the algorithm: it partitions any input string to minimum number of phrases possible while constructing the same dictionary. (For instance, the algorithm using the LZW dictionary together with flexible parsing inserts to the dictionary the exact same phrases as would the original LZW algorithm with greedy parsing.) The implementation is based on a rather simple data structure, the *trie-reverse-trie-pair*, which has similar properties with the simple trie data structure used for greedy parsing. It is hence expected that $\mathcal{FP}$ would improve over greedy parsing without being penalized for speed or space.

In this study, we report an experimental evaluation of $\mathcal{FP}$ in the context of LZW dictionary construction scheme. We implement compression programs based on $\mathcal{FP}$ (the implementations are available on the WWW [Sou]), and study to what extent the theoretical expectations hold on "random" or "real-life" data. In particular, we consider the following questions:

1. Is it possible to obtain a new dictionary construction scheme based on $\mathcal{FP}$? If yes, how would it perform in comparison to $\mathcal{FP}$ with LZW dictionary construction or the LZW algorithm itself - both asymptotically and in practice? (Note that the general optimality property of $\mathcal{FP}$ does not apply once the dictionary construction is changed.)

2. The trie-reverse-trie-pair is a pointer based data structure whose performance is likely to suffer from pointer jumps in a multi-layer memory hierarchy. Are there alternative data structures to obtain more efficient implementations of $\mathcal{FP}$ - in particular can we employ hashing to support dictionary lookups without all the pointer jumps?

3. What are the sizes of random data on which the better average case asymptotic properties of the LZ78 over LZ77 start to show up?

4. Does the worst case optimality of $\mathcal{FP}$ translate into improvement over greedy parsing on the average case?

5. Do better asymptotic properties of LZW in comparison to LZ-77 and $\mathcal{FP}$ in comparison to LZW show up in any practical domain of importance? Specifically how well does $\mathcal{FP}$ perform on DNA/protein sequences and medical images?

We address each one of these issues as follows:

1. We consider a data compression algorithm based on $\mathcal{FP}$, which constructs the dictionary by inserting it the concatenation of each of the substrings parsed with the character following them

- as in the case of LZW algorithm. We will refer this algorithm as the $\mathcal{FP}$-based-alternative-dictionary-LZW algorithm, or FPA. The dictionary built by FPA on any input still satisfies the prefix property in every execution step of the algorithm. In our experiments we consider the implementation of FPA as well as the implementation of the compression algorithm which builds the same dictionary as LZW, but uses $\mathcal{FP}$ for output generation which we refer as LZW-$\mathcal{FP}$. We compare the compression ratios obtained by LZW-$\mathcal{FP}$ and FPA with that of UNIX `compress` and `gzip`.

2. We present an on-line data structure based on Karp-Rabin fingerprints [KR87], which implements both LZW-$\mathcal{FP}$ and FPA in expected $O(1)$ time per character, by using space proportional to the size of the codewords in the output. We are still in the process of improving the efficiency of our implementations; we leave to report our timing results to the full version of this paper. We note, however, that our algorithms run about $3-5$ times slower than `compress` which is the fastest among all algorithms, both during compression and decompression. We also note that all the software, documentation, and detailed experimental results available on the WWW [Sou]. The readers are encouraged to check updates to the web site and try our software package.

3. We first demonstrate our tests on pseudorandom (non-uniform) i.i.d. bit strings with a number of bit probabilities. We observe that the redundancy in the output of each of the four programs we consider approach to the expected asymptotic behavior very fast - requiring less than 1KB for each of the different distributions, and better asymptotic properties of LZW in comparison to LZ77 can be very visible. For files of size $> 1MB$, `compress` can improve over `gzip` up to 20% in compression achieved. A next step in our experiments will involve pseudo-random sources of limited markovian order.

4. We report on our experimens with several "real-life" data files as well; those include DNA/protein sequences, medical images, and files from the Calgary corpus and Canterbury corpus benchmark suites. These results suggest that both LZW-$\mathcal{FP}$ and FPA are superior to LZW (UNIX `compress`) in compression attained, up to 20%. We also observe that both LZW-$\mathcal{FP}$ and FPA are 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-$\mathcal{FP}$.

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 each of the compression algorithms.

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.

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 iteration $m+1$. 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 $T[\alpha - 1]$.

We demonstrate how the execution of the LZW and LZW-$\mathcal{FP}$ algorithms differ in the figure below.

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 $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$.

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 $T[b_m : f(b_m) + 1]$, 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 $T[b_m : f(b_m) + 1]$ 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 $\mathcal{D}$.

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 $T[b_{m+1} - 1]$.

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 guarantees a worst case linear running time for both algorithms as described in [MS98]). The new data structure based on *fingerprints* [KR87], is randomized, and guarantees expected linear running time for any input.

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 $\mathcal{D}$ and return a pointer. 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 on a path of 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 a character 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 u 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 u with corresponding character a; hence extend operation takes $O(1)$ time. Similarly the contract operation takes $O(1)$ time by going from u to u', the node representing reverse of S in $\mathcal{T}^r$, and checking if the parent of u' 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 ϵ 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 $O(1)$ expected time: suppose the hash value of a given string S is h_S, and the label of S is ℓ. To extend S with character a, we first compute from h_S, the hash value h_{Sa} of the string Sa. 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 $h_{S'}$ of the string $S' = S[2:|S|]$. Among the phrases whose hash value is $h_{S'}$, the one whose label matches the suffix label of S gives the result of the extend 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 and all its prefixes takes $O(|R|)$ expected 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 inductively 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 $S' = T[b_m : f(b_m) + 1]$ 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 $b_m + i > f(b_m)$. 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 testing 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 experimented with two versions of FPA, one whose dictionary was limited to 2^{16} phrases, and the other with 2^{24} phrases.

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 test 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 in uncompressed `pgm` format in ASCII (*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, which is the most popular benchmark suite for lossless compression. It includes a bibliography file (*bib*), two complete books (*book1, book2*), two binary files (*geo, pic*), source codes in `c, lisp, pascal` (*progc, progl, progp*), and the transcript of a login session (*trans*). The third set also 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 FPA implementation 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. Also the theoretical expectations for the convergence rate in the redundancy of the output for i.i.d. data were consistent with the test results. 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 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 were consistent our theoretical expectations: the asymptotic properties were observed even in strings of a few KB size. In general, all LZW based schemes performed 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 may provide 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.

References

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

[FM95] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. In *ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[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.

File	*Size* (KB)	*gzip* (KB)	*compress* (KB)	*LZW-FP* $\uparrow_g$ (%)	*LZW-FP* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
bib	109	34	45	-26.01	5.04	-19.69	9.80	-19.69	9.80
book1	751	306	324	-3.81	2.03	-2.48	3.29	3.94	9.34
book2	597	202	245	-16.61	3.89	-12.32	7.42	-7.47	11.42
geo	100	67	76	-11.90	1.46	-11.66	1.67	-11.66	1.67
pic	501	55	61	-6.64	3.25	-5.31	4.47	-5.31	4.47
progc	39	13	19	-37.25	4.85	-33.40	7.52	-33.40	7.52
progl	70	16	26	-56.83	5.99	-49.29	10.51	-49.29	10.51
progp	48	11	19	-59.70	6.54	-53.75	10.02	-53.75	10.02
trans	91	18	37	-87.12	7.12	-73.96	13.65	-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*, and *FPA-24* over *gzip* and *compress*

File	*Size* (KB)	*gzip* (KB)	*compress* (KB)	*LZW-FP* $\uparrow_g$ (%)	*LZW-FP* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
E.coli	4530	1310	1226	6.91	0.56	6.43	0.05	8.48	2.24
bible.txt	3953	1163	1369	-12.87	4.11	-7.79	8.42	4.68	19.01
world192.txt	2415	708	964	-31.70	3.32	-20.36	11.64	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*, and *FPA-24* over *gzip* and *compress*

[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.

[MS98] Y. Matias and S. C. Sahinalp. On optimality of parsing in dynamic dictionary based data compression. Unpublished Manuscript, 1998.

[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.

File	*Size* (KB)	*gzip* (KB)	*compress* (KB)	*LZW-FP* $\uparrow_g$ (%)	*LZW-FP* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
dna1	3096	978	938	5.59	1.54	5.75	1.70	8.91	5.00
dna2	2877	846	813	4.64	0.75	4.33	0.43	5.89	2.05
mr.pgm	260	26	29	-7.23	3.60	6.38	15.84	6.38	15.84
ct.pgm	1039	110	110	4.10	3.61	14.56	14.12	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*, and *FPA-24* over *gzip* and *compress*

File	*Size* (KB)	*gzip* (B)	*compress* (B)	*LZW-$\mathcal{FP}$* $\uparrow_g$ (%)	*LZW-$\mathcal{FP}$* $\uparrow_c$ (%)	*FPA* $\uparrow_g$ (%)	*FPA* $\uparrow_c$ (%)	*FPA-24* $\uparrow_g$ (%)	*FPA-24* $\uparrow_c$ (%)
P(0)=0.7	1	212	217	2.83	5.07	2.83	5.07	2.83	5.07
P(1)=0.3	10	1644	1554	7.48	2.12	9.85	4.63	9.85	4.63
	100	15748	13558	15.23	1.54	17.58	4.27	17.58	4.27
	1024	**160011**	**132278**	**18.35**	**1.23**	**20.40**	**3.71**	**20.60**	**3.95**
P(0)=0.9	1	142	143	5.63	6.29	6.34	6.99	6.34	6.99
P(1)=0.1	10	1024	924	12.79	3.35	16.80	7.79	16.80	7.79
	100	9839	7781	23.11	2.78	26.06	6.50	26.06	6.50
	1024	**99853**	**74075**	**27.48**	**2.25**	**30.21**	**5.92**	**30.21**	**5.92**
P(0)=0.97	1	99	107	3.03	10.28	4.04	11.21	4.04	11.21
P(1)=0.03	10	508	503	5.91	4.97	10.24	9.34	10.24	9.34
	100	4625	3754	22.42	4.42	28.06	11.37	28.06	11.37
	1024	**45957**	**33420**	**29.76**	**3.41**	**34.64**	**10.12**	**34.64**	**10.12**

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*, and *FPA-24* over *gzip* and *compress*; random data generated by drand48().

[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.

[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.

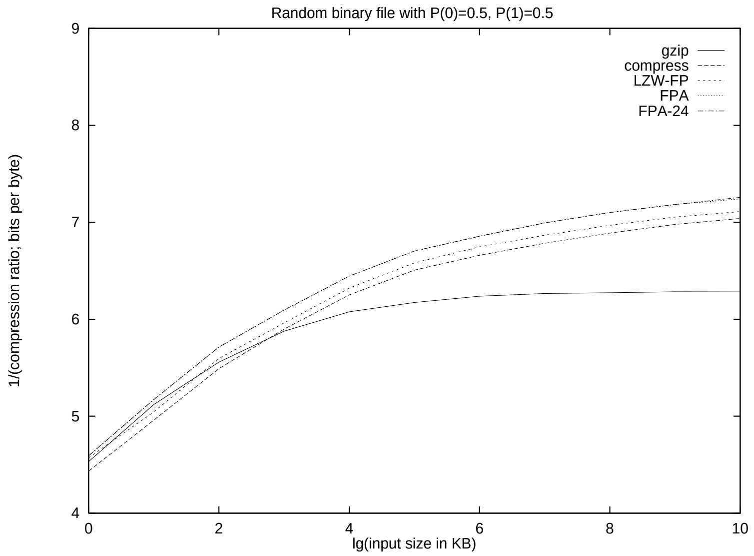

Figure 2: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = P(1) = .5$.

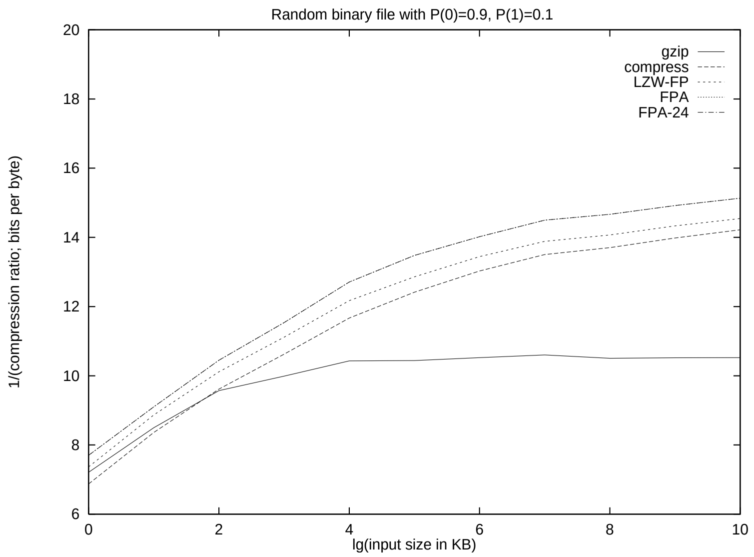

Figure 3: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = .9$ and $P(1) = .1$.

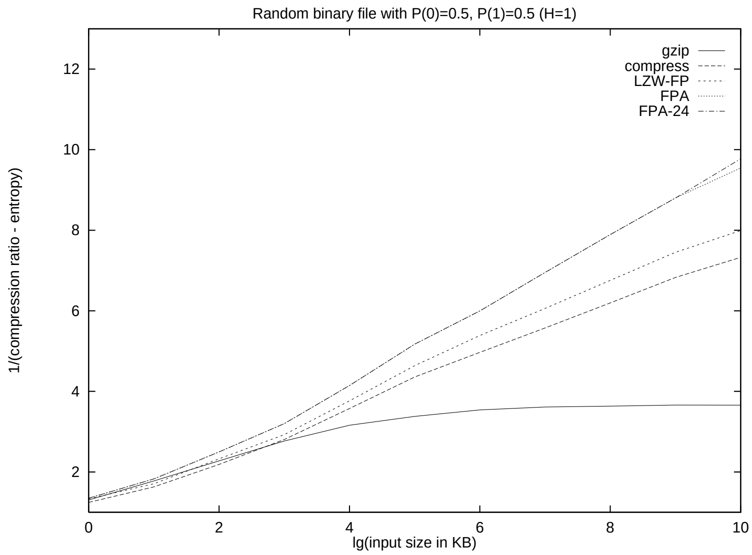

Figure 4: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = P(1) = .5$.

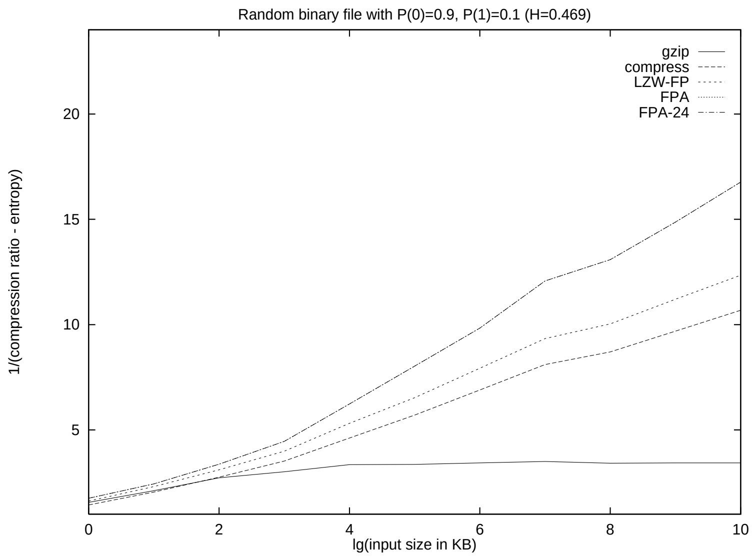

Figure 5: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = .9$ and $P(1) = .1$.