On the optimality of parsing in dynamic dictionary based data compression

(preliminary version)

Yossi Matias * Süleyman Cenk Şahinalp †

May 15, 1998

Since the introduction of dynamic dictionary based data compression by Ziv and Lempel two decades ago, many dictionary construction schemes have been proposed and implemented. This paper considers the following question: once a (dynamic) dictionary construction scheme is selected, is there an efficient dynamic parsing method that results with the smallest number of phrases possible for the selected scheme, for all input strings. It is shown that greedy parsing, a method used in almost all dictionary based algorithms (including UNIX `compress`, `gif` image compression, and fax and modem standards), can be far from optimal for certain input strings. On the positive side, a simple parsing method is introduced which, for any selected dictionary scheme which has the prefix property (this includes virtually all the variations on Lempel-Ziv algorithms), is optimal with respect to the selected scheme for any input string. Also introduced is a simple data structure that enables an efficient dynamic implementation of the parsing method, in $O(1)$ time per character and optimal space requirement.

*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. Partly supported by Alon Fellowship.

†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. Partly supported by ESPRIT LTR Project no. 20244 - ALCOM IT. Part of this research was done when the author was with Bell Labs, Murray Hill, NJ, USA.

1 Introduction

Given a text string T, a compression algorithm $\mathcal{C}$ computes an output string, T', whose representation is smaller than that of T, and such that a corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$ can take T' as input and compute T. The most common compression algorithms used in practice are the dictionary schemes (a.k.a. parsing schemes [BCW90], or textual substitution schemes [Sto88]). These 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, that are called *codewords*. The task of partitioning the text into phrases is called *parsing*.

The dictionary can be constructed in static or dynamic fashion. In *static* schemes, the whole dictionary is constructed before the input is compressed. All 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. Most dynamic schemes (e.g., [ZL77, ZL78, Wel84, Yok92]) satisfy the *prefix property*: for any given phrase in the dictionary, all its prefixes are also phrases in the dictionary. Some of them (e.g., [ZL77, Yok92]) satisfy the *suffix property*: for any given phrase in the dictionary, all its suffixes 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]. The LZW scheme is the basis for UNIX `compress`, `gif` image compression format, and most popular fax and modem standards. The LZ77 scheme is the basis for all `zip` variants, including the UNIX `gzip`. All three algorithms are known to be asymptotically optimal for certain sources, in the sense that their compression ratio approaches to the bit entropy of the input source. Recently, it was shown in [JS95, LS95, Sav97] that the compression ratio of LZ78 approaches the asymptotic optimality faster than that of LZ77 (see appendix A.1 for details). 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.

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 text with small redundancy. Unfortunately, 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. In this paper we do not consider the question of dictionary construction scheme. Rather, we focus on the parsing method that is to be used once a particular dictionary construction scheme is selected.

In particular, this paper considers the following question: *Given a dictionary construction scheme, is there an efficient dynamic parsing method that achieves optimality with respect to this schemes on all input strings?* Note that unlike for the dictionary construction scheme, the question on optimality of the parsing scheme is well defined. We concentrate here on dictionaries in which all codewords are represented with the same number of bits. Hence, given a particular dictionary construction scheme, an optimal parsing scheme would compute the compressed version of any input string with the smallest number of phrases possible.

Given a dictionary construction scheme and a given input text, it is possible to obtain an optimal parsing via a dynamic programming algorithm. However, such an algorithm uses time and the space quadratic with the input size, and requires all the input to be available before the compression process starts. See [SS82] for a study of optimal parsing schemes; see [LH87, Sto88, BCW89, BCW90] for a detailed coverage of parsing strategies and dictionary compression algorithms. For the static case only, a recent work [FM95] describes a linear time optimal algorithm for dictionaries satisfying the prefix property. As this algorithm requires the entire input to be available before the compression process starts, it is not suitable for on-line settings or dynamic applications.

Almost all dictionary based algorithms in the literature (e.g., [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. (See [BW94] for a study of greedy parsing with static dictionaries.) Greedy parsing is fast and can usually be applied on-line, and is hence very suitable for communications applications. However, it was shown in [GSS85] that for static dictionaries greedy parsing can be quite far from optimal: there are strings that can be parsed to m phrases using a given (static) dictionary, for which greedy parsing with the same dictionary obtains $\Omega(m^{3/2})$ phrases. For the dynamic case, it has been an open question how well greedy parsing compares to optimal parsing. This question is particularly important for several practical algorithms including LZ78, LZW, and some implementations of LZ77, where all codewords in the dictionary $\mathcal{D}$ are represented by the same number of bits.

We present the following results, concentrating on dictionaries which satisfy the prefix property:

- *Greedy parsing can be far from optimal for dynamic dictionary construction schemes*: there is a dictionary scheme with the prefix property, so that for any sufficiently large integer m, there exists a string T which can be parsed to $O(m)$ phrases, whereas the greedy parsing results with $\Omega(m^{3/2})$ phrases.
- *Given any dictionary construction scheme with the prefix property, there exists an efficient dynamic parsing method that achieves optimality on all input strings with respect to this scheme*: interestingly, the parsing method is a rather simple one, consisting on greedy steps with a single step lookahead. We call this method *flexible parsing*, or $\mathcal{FP}$.

The negative result for greedy parsing is given for LZ78 and LZW dictionary construction methods, and may therefore have considerable practical implications. For the positive result, we emphasize that the notion of optimality is with respect to a specified dictionary construction. For instance, the algorithm using the LZ78 dictionary together with flexible parsing inserts to the dictionary the exact same phrases as would the original LZ78 algorithm with greedy parsing. Thus, it entirely circumvents the question regarding the choice of the dictionary construction scheme. On the other hand, it is relevant to any such scheme that is selected. We present a novel data structure to implement the flexible parsing method efficiently, in time and space competitive with greedy parsing: For LZ78 or LZW schemes, it runs in amortized $O(1)$ time per character, and requires space proportional to the number of phrases in the dictionary.

In order to consider the issue of optimality, it is imperative to have a formal model that can represent dynamic dictionary compression algorithms. We introduce such model, $\mathcal{M}$, in which for every compression algorithm $\mathcal{C}$ described by $\mathcal{M}$, there is a corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$ that is guaranteed to correctly decompress any string previously compressed by $\mathcal{C}$. An important property of $\mathcal{C}^{\leftarrow}$ is that it depends only on the dictionary construction method, and not the parsing scheme. Hence, for two algorithms which have identical dictionary construction rules yet different parsing methods, the decompression algorithms suggested by the model $\mathcal{M}$ are identical. Based on this model, we also provide a formal definition of *latency* in dictionary compression algorithms, and specify conditions for considering an algorithm *on-line*. An interesting aspect of the model $\mathcal{M}$ is that it can be used to succinctly describe substantially all known dictionary compression algorithms in the literature, and it may be useful to describe and study alternative dictionary construction schemes. It also turns out that while the model specification is quite detailed and elaborated, it allows for rather concise proofs regarding the optimality of the parsing methods.

We generalize the optimality result of flexible parsing by showing that k-step greedy parsing (which is greedy parsing with $k > 0$ lookaheads) is optimal for dictionaries $\mathcal{D}$ satisfying the $(k+1)$-*overlap property*, in which for any given string R which can be obtained by concatenating $k+1$ phrases from $\mathcal{D}$ with overlaps, there also exist at most $k+1$ phrases whose concatenation without overlaps gives R. One interesting observation is that for any static dictionary scheme with the $(k+1)$-overlap property, optimal parsing implies certain *robustness*: the difference in the number of phrases obtained by an optimal parser for two strings with edit distance e is $O(e)$.

Finally, we show that greedy parsing is optimal for dictionaries satisfying the suffix property. This immediately implies that for any input string, LZ77 algorithm obtains the minimum number of phrases (though not necessarily minimum number of bits) possible by any dictionary compression method, partially resolving an open problem regarding the general case [Ziv97].

2 A model for on-line dictionary compression algorithms

In order to have a formal study of on-line dictionary compression algorithms, and in particular consider questions about the optimality in the parsing method, it is required to specify precisely the algorithmic framework. A recent paper [SR97] describes a (rather informal) framework for dictionary compression as well. This framework is not suitable to our needs, since it cannot express some of the popular compression algorithms; moreover, algorithms that can be expressed by this framework do not necessarily have a corresponding decompression algorithm.

In this section, we present a concrete model $\mathcal{M}$ that enables one to describe all known dictionary compression algorithms in the literature, including LZ78, LZW, LZ77, Miller-Wegman and Yokoo schemes. The model will be used in the following sections to present and analyze the flexible parsing and the k-greedy parsing scheme.

In the proposed model $\mathcal{M}$, a dictionary compression algorithm $\mathcal{C}$ takes as input an alphabet Σ, and a finite string T which consists of characters from Σ. Denote by $T[h]$ the h^{th} character of T, starting with $h = 0$; denote by $T[h : j]$, the substring of T which starts at the h^{th} character and extends until the j^{th} character. We denote the length of T by n, hence $T = T[0 : n-1]$.

The compression algorithm $\mathcal{C}$ uses and possibly maintains a set of substrings $\mathcal{D}$, denoted *dictionary*. The elements of $\mathcal{D}$ are called *phrases*, and for each phrase S in $\mathcal{D}$ there is a unique label $C(S)$, which is called the *codeword* that corresponds to S. The output of $\mathcal{C}$ is a sequence of codewords which is called the *compressed text* and denoted as $C(T)$. The decompression algorithm $\mathcal{C}^{\leftarrow}$ takes as input a compressed text $C(T)$, and maintains the same dictionary $\mathcal{D}$ as the compression algorithm $\mathcal{C}$. It replaces the codewords in $C(T)$ to reconstruct the text T. We describe below both the compression and decompression algorithms. Appendix A.2 provides the concrete realizations of several popular compression algorithms, using the model $\mathcal{M}$.

2.1 Incremental dictionary compression

The compression algorithm $\mathcal{C}$ can have one of the two types of dictionaries: static or dynamic. If $\mathcal{C}$ is a static dictionary algorithm, then both the substrings in $\mathcal{D}$ and their corresponding codewords do not change during the execution of $\mathcal{C}$. If $\mathcal{C}$ is a dynamic dictionary algorithm, then during its execution $\mathcal{C}$ inserts some of the substrings of T into $\mathcal{D}$; $\mathcal{C}$ may also update some of the codewords of existing phrases in $\mathcal{D}$.

The compression algorithm $\mathcal{C}$ parses the text T incrementally, starting with $T[0]$. $\mathcal{C}$ consists of n iterations. If $\mathcal{C}$ is dynamic, then in each iteration i it reads $T[i]$, (posslibly) updates $\mathcal{D}$, and then (possibly) parses T further for outputting codewords. We denote by $\mathcal{D}_i$ the state of the dictionary $\mathcal{D}$ at the end of iteration i. If $\mathcal{C}$ is static then $\mathcal{D}$ remains unchanged.

An important aspect of the model is that it distinguishes between the parsing process for constructing the dictionary, and the parsing process for output computation. We call the first process *dictionary parser* and denote it as $\mathcal{P}_d$; we call the second process *output parser* and denote it as $\mathcal{P}_o$. A schematic figure with the different components of the compression algorithm $\mathcal{C}$ is given in Figure 1.

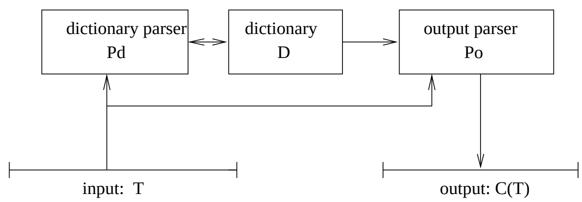

Figure 1: Model $\mathcal{M}$ for incremental dictionary compression algorithm $\mathcal{C}$.

The purpose of the model is to give a general framework for describing incremental dictionary compression algorithms by specifying what $\mathcal{P}_d$ and $\mathcal{P}_o$ do in each iteration. We first describe the general operations of $\mathcal{P}_d$ and $\mathcal{P}_o$ at iteration i.

The dictionary parser $\mathcal{P}_d$ has three steps:

(1) $\mathcal{P}_d$ determines some characters $T[j]$, $0 \leq j < i$, as starting positions of new dictionary phrases, and assigns them codewords; we call the dictionary phrases whose ending positions are yet to be determined the *developing* dictionary phrases.

(E.g. let $T = a, b, c, d, e, \ldots$; in iteration 4, $\mathcal{P}_d$ determines $T[2] = c$ as the starting position of a developing phrase and assigns it the codeword 8.)

(2) $\mathcal{P}_d$ determines some characters $T[k]$, $k \leq i$ as ending positions of some of the developing dictionary phrases, whose starting positions were determined in earlier iterations.

(E.g. (cont'd) in iteration 5, $\mathcal{P}_d$ determines $T[3] = d$ as the ending position of the phrase with codeword 8, resulting with $C(cd) = 8$.)

(E.g. (cont'd) the phrase $T[2:3] = c, d$ is in $\mathcal{D}(4 + i, 5 + j)$, for all $1 \leq i \leq j$.)

(3) $\mathcal{P}_d$ may update the codewords of existing dictionary phrases.

(E.g. (cont'd) in iteration 6, say, $\mathcal{P}_d$ may change the codeword of (cd) from 8 to 3.)

Given integers $l' \leq l$ we denote by $\mathcal{D}(l', l)$ the set of all dictionary phrases whose starting positions are determined before iteration l', and whose ending positions are determined before iteration l. Let $T[1 : c(i)]$ be the compressed prefix of T in iteration i of the algorithm.

The output parser $\mathcal{P}_o$ has three steps:

(1) $\mathcal{P}_o$ reads the character $T[i]$.

(2) $\mathcal{P}_o$ identifies a (possibly empty) substring that we denote by T_i, where $T_i = T[c(i-1)+1 : c(i)]$, for some increasing function $c()$ for which $c(i-1) \leq c(i) \leq i$ (if the substring is empty then $c(i-1) = c(i)$). If T_i is a non-empty string then $\mathcal{P}_o$ partitions it into some $g(i)$ substrings, $T_i^0, T_i^1, \ldots, T_i^{g(i)-1}$. Each substring $T_i^\ell = T[c_\ell(i)+1 : c_{\ell+1}(i)]$, for some $c_0(i) \leq c_1(i) \leq \cdots \leq c_{g(i)-1}(i) \leq c_{g(i)}$, where $c_0(i) = c(i-1)$ and $c_{g(i)}(i) = c(i)$. The partition is under the constraint that for $0 \leq \ell \leq g(i)$, T_i^ℓ is in $\mathcal{D}(c_{\ell-1}(i)+1, c_\ell(i))$.
(3) $\mathcal{P}_o$ outputs the codewords $C_i^1, C_i^2 \ldots, C_i^{g(i)}$, where C_i^ℓ corresponds to the substring T_i^ℓ in $\mathcal{D}(c_{\ell-1}(i) + 1, c_\ell(i))$.
(E.g. (cont'd) in iteration 6, $\mathcal{P}_o$ may identify $T[2:3] = c, d$, and without further partitioning may directly output its corresponding codeword 8.)

Greedy parsing: In greedy parsing (with no lookaheads), the objective is to maximize at each iteration the length of the prefix of T which is compressed until that point. Specifically, $\mathcal{P}_o$ selects $c(i)$ to be $i-1$ ($i > 0$), when $T[c(i-1)+1 : i-1]$ is a phrase in $\mathcal{D}_{i-1}$, but $T[c(i-1)+1 : i-1]$ is not a phrase in $\mathcal{D}_i$, and always assigns $g(i) = 1$.

One feature of the model is that any compression algorithm $\mathcal{C}$ that can be described by the above model, has a corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$. Given the compressed version $C(T)$ of any string T, $\mathcal{C}^{\leftarrow}$ is guaranteed to correctly compute T. We describe in the next subsection how $\mathcal{C}^{\leftarrow}$ works.

2.2 Incremental dictionary decompression

The input $C(T)$ to the decompression algorithm $\mathcal{C}^{\leftarrow}$ is a sequence of codewords output by $\mathcal{C}$: $C_1^1, C_1^2 \ldots C_1^{g(1)} \ldots C_i^1, C_i^2 \ldots C_i^{g(i)} \ldots$. After reading each codeword C_i^l, $\mathcal{C}^{\leftarrow}$ replaces it with its corresponding phrase T_i^l, while building the exact same dictionary $\mathcal{D}$ that $\mathcal{C}$ builds for T. A schematic figure with the different components of the decompression algorithm $\mathcal{C}^{\leftarrow}$ is given in Figure 2.

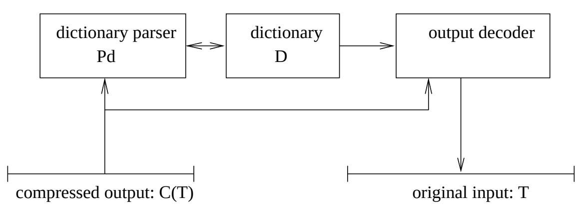

Figure 2: The incremental dictionary decompression $\mathcal{C}^{\leftarrow}$.

The decompression algorithm $\mathcal{C}^{\leftarrow}$ maintains the invariant that after the codeword C_i^ℓ from the input is decoded, the prefix of the text already decompressed is $T[0 : c_\ell(i)]$. The operation of $\mathcal{C}^{\leftarrow}$ works as follows. By definition, T_i^ℓ is the phrase that corresponds to C_i^ℓ. Let $T[j : k]$ be an identical phrase to T_i^ℓ in $\mathcal{D}$. If $T[j : k]$ is in $\mathcal{D}_{c_{\ell-1}(i)}$, then clearly $\mathcal{C}^{\leftarrow}$ can immediately replace C_i^ℓ with T_i^ℓ. Otherwise, by the description of the model $\mathcal{M}$, $T[j : k]$ should be in $\mathcal{D}(c_{\ell-1}(i)+1, c_\ell(i))$. Therefore the value of j could be computed once $T[0 : c_{\ell-1}(i)]$ is completely available, which is the case by the invariant above. In this case, starting from $T[j]$, $\mathcal{C}^{\leftarrow}$ concatenates the characters $T[j+1], T[j+2], \ldots$ one by one (without knowing the value of k) to the end of the already decompressed prefix of T. Before $\mathcal{C}^{\leftarrow}$ reaches $T[k]$, it is guaranteed to be able to deduce the value of k, hence guaranteeing the correct decompression of any string compressed by $\mathcal{C}$.

Lemma 2.1 (correct decompression) *Given a dictionary compression algorithm $\mathcal{C}$ that can be described by the model $\mathcal{M}$, its corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$ as described above is guaranteed to decompress the compressed version of any input string T.*

Proof. We show that in the procedure described above $\mathcal{C}^{\leftarrow}$ correctly computes the value of k before reaching $T[k]$. Indeed, in the model $\mathcal{M}$ we insist that a substring $T[c_{\ell-1}(i)+1 : c_\ell(i)]$ is parsed and replaced by its codeword $C_\ell(i)$ (in the case of $\mathcal{M} - T[j:k]$) only if it is in $\mathcal{D}(c_{l-1}(i)+1, c_l(i))$. By definition, $T[0 : c_l(i)]$ is identical to the concatenation of $T[0 : c_{l-1}(i)]$ with $T[j : k]$, hence once $\mathcal{C}^{\leftarrow}$ adds all characters of $T[j : k]$ to the end of $T[0 : c_{l-1}(i)]$ (the already decompressed prefix of T), $\mathcal{D}(c_{l-1}(i)+1, c_l(i))$ would be completely computed. □

From Lemma 2.1 we obtain:

Corollary 2.2 (identical decompression) *The decompression algorithm that corresponds to a given compression algorithm only depends on its dictionary construction scheme, and is independent from its output parser. Therefore, two compression algorithms which have the same dictionary construction scheme have the same corresponding decompression algorithm.*

Given the compression algorithm $\mathcal{C}$ and the input $T[0 : n-1]$, let the string $T[c(i)+1 : i]$, the uncompressed portion of T in iteration i, be denoted as the *buffered string* in iteration i, and $\phi(i) = i - c(i)$, the size of the buffered string, be denoted as the *delay* of $\mathcal{C}$ in iteration i. We denote by ϕ, the maximum delay of $\mathcal{C}$ over all iterations on input T. We call an algorithm *on-line* only if for any input T and at any iteration i, the buffered string $T[c(i)+1 : i]$ is later parsed to at most some λ phrases, where λ is a constant independent of the length of T. On-line performance is particularly interesting in the context of communications applications.

3 Optimal parsing in prefix dictionaries

Consider a dynamic dictionary compression algorithm $\mathcal{C}$, described in the model $\mathcal{M}$, with a dictionary parser $\mathcal{P}_d$ that satisfies the prefix property.

Theorem 3.1 *Let m be a sufficiently large integer. If the output parser $\mathcal{P}_o$ is a greedy parser then there exists a string T which can be parsed to $O(m)$ phrases, whereas the greedy parsing results with $\Omega(m^{3/2})$ phrases.*

Theorem 3.2 *There exists a dynamic parsing method $\mathcal{P}_o$ that computes the optimal number of phrases with respect to $\mathcal{P}_d$. It runs in amortized $O(1)$ time per character, and requires space proportional to the number of phrases in the dictionary.*

Theorem 3.1 is proved in Section 3.1. Theorem 3.2 is proved in Section 3.2 and Section 3.3.

3.1 Limits of greedy parsing

The following lemma shows that for LZW dictionary construction scheme, greedy parsing can be far from the optimal.

Lemma 3.3 *There are input strings which can be parsed to some $O(\ell)$ phrases, and be represented by $O(\ell \log \ell)$ bits by using the LZW dictionary, for which LZW scheme with greedy parser obtains $\Omega(\ell^{3/2})$ phrases and outputs $\Omega(\ell^{3/2} \log \ell)$ bits.*

Proof. We first construct such a string T which uses an arbitrarily large dictionary $\Sigma = \{0, 1, \ldots, k, k+1, k+2, \ldots, k+\sqrt{k}\}$, where k is a prime number. Let R be the substring $1, \ldots, k$, and let R_i denote the concatenation of i copies of the string R. Let S be the substring $1, 2, 1, 2, 3, \ldots, 1, 2, \ldots, k$, and let T be the concatenation of $0, S, k+1, 0, R_1, 1, k+2, 0, R_2, 1, \ldots, k+\sqrt{k}, 0, R_{\sqrt{k-1}}, 1$.

The LZW dictionary scheme first processes the substring S, and inserts the substrings $(01), (12), (21), (123), (31), \ldots, (12\ldots k)$ in $\mathcal{D}$ with respective codewords $k+\sqrt{k}+1, k+\sqrt{k}+2, \ldots, 3k-2$. Then it processes the substrings $(k+i, 0, R_i, 1)$ for $i = 1, \ldots, \sqrt{k-1}$: for each such substring it first inserts in $\mathcal{D}$, $(1(k+i))$, then inserts $((k+i)01\ldots i+1)$, and because k is prime, then inserts all substrings of R_i of size $i+1$. Altogether there will be $k+2$ insertions to $\mathcal{D}$.

Notice that initially the size of the dictionary $k+\sqrt{k}+1$, and hence each possible codeword requires more than $\log k$ bits for a unique representation. The maximum size of the dictionary is $k^{3/2} + O(k)$, hence no more than $(3/2)\log k + O(1)$ bits are required to represent a codeword at any iteration.

For each substring inserted in $\mathcal{D}$, LZW outputs one codeword, hence the total number of codewords output by LZW for T is at least $k^{3/2}$. This implies that the total number of bits it outputs is at least $k^{3/2} \log k$.

An optimal parser still obtains $2k-1$ phrases for S; however it obtains only one phrase for every occurrence of R in T. Hence the number of phrases it outputs for each R_i is no more than $i+2$, and the total number of phrases it outputs for T is no more than $3k$, and the total number of bits it outputs is no more than $(9/2)k\log k + O(k)$.

The proof for binary strings (hence for any constant size Σ) follows after replacing all characters $0, 1, \ldots, k+\sqrt{k}$ with their binary representations, and adding to the beginning of T a training sequence which would insert all binary substrings of size $\log(k+\sqrt{k})$ to $\mathcal{D}$. □

Lemma 3.4 *There are input strings which can optimally be parsed to $O(\ell)$ phrases, and be represented by $O(\ell \log \ell)$ bits, for which LZ78 obtains $\Omega(\ell^{3/2})$ phrases and outputs $\Omega(\ell^{3/2} \log \ell)$ bits.*

Proof. The proof is similar to that of lemma 3.3, once T is set to be the concatenation of $S, 0, R_0, 0, R_1, \ldots, 0, R_{\sqrt{k}}$. □

3.2 Optimality results for flexible parsing

A parsing scheme of particular interest is the 1-step greedy parsing, which we call the flexible parsing, $\mathcal{FP}$. Rather than greedily parsing the longest advancing prefix of the uncompressed portion of the text, $\mathcal{FP}$ uses the flexibility of choosing the prefix which results in the longest advancement in the next iteration. We demonstrate in Table A.3 and Figure 3 how the flexible parser can be used with the dictionary parser $\mathcal{P}_d$ of the LZW compression algorithm on the same input used in Table 7. The execution of the corresponding decompression algorithm on the output of the compression algorithm is demonstrated in Table A.3.

A natural question is therefore whether $\mathcal{FP}$ is merely a first level improvement towards better parsing schemes. Surprisingly, it turns out that $\mathcal{FP}$ obtains phrase optimality, and cannot be further improved with respect to dictionary construction schemes with the prefix property.

Lemma 3.5 (optimality for $\mathcal{FP}$) *For any dictionary construction scheme $\mathcal{P}_d$ within the model $\mathcal{M}$ which builds a dictionary that satisfies the prefix property at all iterations, flexible parsing obtains the minimum number of phrases out of any input string T.*

Proof. Let $\mathcal{P}_d$ be a dictionary construction scheme that builds a dictionary satisfying the prefix property at all iterations, and let $\mathcal{C}$ and $\mathcal{C}'$ be the compression algorithms that respectively use $\mathcal{FP}$, and any other parsing scheme $\mathcal{P}_o$, together with $\mathcal{P}_d$. Our claim is that the number of codewords output by $\mathcal{C}$ on any given input T is at most that output by $\mathcal{C}'$.

We show by induction on i that $e_1(c_{\mathcal{FP}}(i)) \geq c_{\mathcal{P}_o}(i)$. The induction step for $i+1$ is proven as follows: Since $T[c_{\mathcal{FP}}(i)+1 : e_1(c_{\mathcal{FP}}(i))]$ is in $\mathcal{D}[c_{\mathcal{FP}}(i)+1 : e_1(c_{\mathcal{FP}}(i))]$, the prefix property implies that $T[c_{\mathcal{FP}}(i)+1 : c_{\mathcal{P}_o}(i)]$ is in $\mathcal{D}[c_{\mathcal{FP}}(i)+1 : e_1(c_{\mathcal{FP}}(i))]$. Therefore by definition of the flexible parsing $e_1(c_{\mathcal{FP}}(i+1)) \geq c_{\mathcal{P}_o}(i+1)$. □

From the above lemma we can immediately infer the following.

Corollary 3.6 (compression optimality) *If a given $\mathcal{P}_d$ builds a dictionary which at every iteration (1) satisfies the prefix property (2) represents each codeword with the same number of bits, then the output of the compression algorithm $\mathcal{C}'$, which uses $\mathcal{P}_d$ and $\mathcal{FP}$ consists of the minimum number of bits achievable by any compression algorithm $\mathcal{C}$ that uses $\mathcal{P}_d$.*

Corollary 3.7 (compression optimality w.r.t. LZ78/LZW) *Among the compression algorithms using the LZ78 or LZW dictionary construction schemes, the ones which use $\mathcal{FP}$, outputs the minimum number of bits on any input string.*

We note that parsing methods based on lookaheads were considered in several contexts in compiler theory and recently by Horspool [Hor95], for LZ78 compression. However, the algorithm in [Hor95] builds a different dictionary from LZ78 and hence can perform worse than the LZ78 method (the number of codewords output by this algorithm can be quadratic with that of the LZ78 algorithm). Also, it runs in time quadratic with the input size.

3.3 An efficient data structure for flexible parsing

It is possible to implement $\mathcal{FP}$ using the suffix trees data structure (see [RPE81] for the use of suffix trees in on-line compression algorithms). However, the space requirements of a suffix tree is substantial: it is proportional to $|T| + |\mathcal{D}|$, where $|T|$ is the number of characters in the input string, and $|\mathcal{D}|$ is the number of phrases in $\mathcal{D}$. In contrast, the space requirement of our data structure (as well as the standard trie used for greedy parsing) is proportional to the number of dictionary entries. In the case of LZW or LZ78, $|\mathcal{D}| = O(|T|/\log|T|)$ and $|\mathcal{D}| = \Omega(\sqrt{|T|})$ [Sto88].

In this section we describe a more efficient data structure to implement the $\mathcal{FP}$ technique. Similar to suffix trees, our data structure enables two basic operations: (1) insert a phrase (2) search for a phrase. The running time for performing each of the operations is identical in both the suffix tree and our data structure. However, the space requirement of our data structure is only $O(|\mathcal{D}|)$, much smaller than that of the suffix tree.

The standard data structure used in many compression algorithms is a compressed trie $\mathcal{T}$ which 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.

Our data structure still 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 dictionary: Given a string $A = a_1, a_2, \ldots, a_n$, its reverse A^r is defined to be the string $a_n, a_{n-1}, \ldots, a_2, a_1$. Therefore for each node v in $\mathcal{T}$, there will be a corresponding node v^r in $\mathcal{T}^r$ which represents the reverse of the phrase represented by v.

The data structure described above is used by $\mathcal{FP}$ in the following context. Given a string $T[k:l]$, where $l = e_1(k)$ the maximal 1-extension of position k, this data structure can efficiently compute the maximum position $k < i \leq l$, for which $e_1(i) \geq e_1(m)$ for all $k < m \leq l$. We use the following lemma to compute i efficiently.

Lemma 3.8 *For any given phrase A in $\mathcal{D}$, the parent of the node for A^r in $\mathcal{T}^r$ represents the longest suffix, B, of A that is in $\mathcal{D}$.*

Proof. If a prefix of A^r is represented in $\mathcal{T}^r$ as a node, then it should be an ancestor of A^r by the construction of $\mathcal{T}^r$. Also each ancestor of the node for A^r represents a prefix of A which is in the dictionary, by definition. Clearly the parent of the node for A^r is the longest one among them. □

Given $T[k:l]$, the computation of i is performed in iterations. In the first iteration i is assigned to k; we also note that $e_1(i) = l$ by definition. By using the above lemma, we compute in constant time, the longest suffix $T[i' : e_1(i)]$ of $T[i : e_1(i)]$, which is in $\mathcal{D}$, for which $i' \leq l$. We make the assignment $i = i'$, and proceed with the next iteration. The algorithm terminates if at any iteration no such i' exists.

Lemma 3.9 *The algorithm described above correctly computes the value of i.*

Proof. The value of i computed by the algorithm above clearly satisfies $k < i \leq l$. Moreover: (1) there can not exist a j, such that $k < i < j \leq l$, and $e_1(j) > e_1(i)$, as the algorithm would not have terminated at this value of i, and (2) there can not exist a j, such that $k < j < i \leq l$, and $e_1(j) > e_1(i)$, as the algorithm would have terminated before reaching i. A contradiction. □

Running time: Note that each character of the input is processed by the algorithm only once, hence the running time of the algorithm will be $O(1)$ per character for performing the searches. The time for each insertion is proportional with the number of characters in the inserted phrase. Hence the total running time of the algorithm would be $O(|T| + |\mathcal{D}|)$.

Space considerations: The space requirement of our data structure is competitive with the original trie approach. For efficient representations of the substrings A^r of the edges in the reverse trie $\mathcal{T}^r$ we keep pointers at each such edge to the beginning and ending nodes of the substring A in the original trie $\mathcal{T}$. Hence the space required for our data structure is $O(|\mathcal{D}|)$.

4 Optimality of greedy parsing and k-step greedy parsing

In this section we identify dictionaries (which may not have the prefix property) that can be parsed optimally by k-step greedy parsing. This generalized the optimality result of flexible parsing for prefix dictionary. We also show that (standard) greedy parsing provides optimal parsing for suffix dictionaries, which have some interesting implications for the LZ77 scheme.

4.1 Optimality results for greedy parsing

The theorem below describes under which conditions greedy parsing with no lookaheads, $\mathcal{GR}$, achieves optimality.

Theorem 4.1 (optimality for $\mathcal{GR}$) *For any dictionary construction scheme $\mathcal{P}_d$ which builds a dictionary that satisfies the suffix property at all iterations, $\mathcal{GR}$ obtains the minimum number of phrases out of any input string T.*

Proof. Let $\mathcal{P}_d$ be a dictionary scheme with the suffix property, and let $\mathcal{C}$, and $\mathcal{C}$' be the compression algorithms that respectively use $\mathcal{GR}$ and some other parsing scheme $\mathcal{P}_o$, together with $\mathcal{P}_d$. We show that the number of codewords output by $\mathcal{C}$ on any given input T is at most that by $\mathcal{C}$'.

We prove by induction on i that $c_{\mathcal{GR}}(i) \geq c_{\mathcal{P}_o}(i)$. The induction step for $i+1$ is proven as follows: Since $T[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$ is in $\mathcal{D}[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$, the induction hypothesis and the suffix property imply that $T[c_{\mathcal{GR}}(i)+1 : c_{\mathcal{P}_o}(i+1)]$ is in $\mathcal{D}[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$, and hence in $\mathcal{D}[c_{\mathcal{GR}}(i)+1 : c_{\mathcal{P}_o}(i+1)]$. Therefore by definition of the greedy parsing $c_{\mathcal{GR}}(i+1) \geq c_{\mathcal{P}_o}(i+1)$. □

Corollary 4.2 *Greedy parsing is optimal for the dictionary construction schemes of LZ77 and the Yokoo.*

Lemma 4.3 (similarity of $\mathcal{FP}$ and $\mathcal{GR}$) *Given a dictionary scheme $\mathcal{P}_d$ which builds a dictionary that satisfy the suffix property at all iterations, the compression algorithms $\mathcal{C}$ and $\mathcal{C}$', which respectively use $\mathcal{GR}$ and $\mathcal{FP}$ together with $\mathcal{P}_d$, parse any input string T identically.*

Proof. We prove by induction on i that $c_{\mathcal{GR}}(i) = c_{\mathcal{FP}}(i)$. The induction step for $i+1$ is proven as follows: Because of the suffix property and the definition of greedy parsing, the set $L(i+1)$ of candidate indices consists of $l \in \{c_{\mathcal{GR}}(i), \ldots, c_{\mathcal{GR}}(i+1)\}$, and the maximum extension of $c_{\mathcal{GR}}(i+1)$ is greater than or equal to that of any other candidate index in $L(i+1)$. □

Corollary 4.4 *$\mathcal{FP}$ is optimal for dictionaries with the suffix property.*

Theorem 4.5 *The LZ77 algorithm (with greedy parsing) parses any given input string to the minimum number of phrases possible by any dynamic dictionary scheme.*

Proof. Let $\mathcal{C}$ be an alternative dictionary algorithm which uses the parsing scheme $\mathcal{P}_o$ and the dictionary scheme $\mathcal{P}_d$. We prove by induction on i that $c_{\mathrm{LZ77}}(i) \geq c_{\mathcal{P}_o}(i)$. The induction step for $i+1$ is proven as follows: Since $T[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$ is in $\mathcal{D}[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$, the induction hypothesis and the suffix property imply that $T[c_{\mathrm{LZ77}}(i)+1 : c_{\mathcal{P}_o}(i+1)]$ is in $\mathcal{D}_{\mathcal{P}_o}[c_{\mathcal{P}_o}(i)+1 : c_{\mathcal{P}_o}(i+1)]$, and hence (as the dictionary of LZ77 includes all substrings of $T[0 : c_{\mathrm{LZ77}}(i)])$ in $\mathcal{D}_{LZ77}[c_{\mathrm{LZ77}}(i)+1 : c_{\mathcal{P}_o}(i+1)]$. Therefore by definition of the greedy parsing $c_{\mathrm{LZ77}}(i+1) \geq c_{\mathcal{P}_o}(i+1)$. □

4.2 Greedy parsing with k steps lookahead

Recall that greedy parsing maximizes at each iteration the length of the compressed prefix of the input . Greedy parsing with k lookaheads, or in short k-step greedy parsing, is based on the premise that by being non-greedy in iteration i, one may get better progress in some later iteration $i + k$.

To define k-step greedy parsing formally, we first give the definition of what we call the *maximal k-extension* of a character, In order to provide some better intuition, we first define this notion in the context of static dictionaries, and then generalize it for the dynamic dictionaries.

Maximal k-extension of a character for static dictionaries: Given a static dictionary $\mathcal{D}$, and an input string T, the maximal k-extension of a character $T[i]$, is defined to be the character $T[e_k(i)]$ ($e_k(i) \geq i$) such that $T[i : e_k(i)]$ is the longest prefix of $T[i : n-1]$ which could be parsed to at most k phrases from $\mathcal{D}$.

Maximal k-extension of a character for dynamic dictionaries: Given a character $T[j]$, $0 \leq j \leq n-1$, its maximal k-extension is the character $T[e_k(j)]$, for which $T[j : e_k(j)]$ is the longest prefix of $T[j : n-1]$ which can be parsed into at most k phrases $T[j_0 : j_1 - 1], T[j_1 : j_2 - 1], \ldots, T[j_{k-1} : j_k - 1]$, where $j_0 = j$, $j_k - 1 = e(j)$, and for all $0 \leq i \leq k$, $T[j_i : j_{i+1} - 1]$ is in $\mathcal{D}(j_i, j_{i+1} - 1)$.

k-step greedy parsing: Given a suffix $T[i : n-1]$, k-step greedy parsing parses its following prefix: Consider the substring $T[i + 1 : e_1(i)]$, where $T[e_1(i)]$ is the maximal 1-extension of $T[i]$. Among the characters of $T[i + 1 : e_1(i)]$, consider the ones $T[m]$, for which the substring $T[i : m - 1]$ is a phrase in $\mathcal{D}$. Suppose $T[j]$ is the rightmost character among all such $T[m]$, whose maximal k-extension, $T[e_k(j)]$, is the furthest, i.e., $e_k(j)$ is maximum among k-extensions of all such characters $T[m]$ in $T[i + 1 : e_1(i)]$ for which $T[i : m - 1]$ is a phrase in $\mathcal{D}$. Then, k-step greedy parsing chooses to parse $T[i : j - 1]$.

4.3 Properties for k-step greedy parsing

In this section we show our most general optimality result for parsing, that k-step greedy parsing is optimal for dictionary schemes that satisfy the $(k + 1)$-overlap property. We then concentrate on robustness of optimal parsers to small changes in the input.

Informally a dictionary $\mathcal{D}$ is said to satisfy the *ℓ-overlap property* if for any given string R that can be obtained by concatenating ℓ phrases with overlaps, there also exists at most ℓ phrases whose concatenation without overlaps gives R. We provide the formal definition of this property below.

ℓ-overlap property: A dictionary $\mathcal{D}$ is said to have the ℓ-overlap property if there exists phrases $Q_1, \ldots, Q_\ell$ in $\mathcal{D}$ such that $Q_j = P_j, S_j, P_{j+1}$ for $1 \leq j < \ell$, and $Q_\ell = P_\ell, S_\ell$, for some possibly empty strings $P_1, \ldots, P_\ell$, then there should exist phrases $Q'_1, \ldots, Q'_m$ in $\mathcal{D}$, for which $m \leq \ell$ and $Q'_1, \ldots, Q'_m = R = P_1, S_1, \ldots, P_{\ell+1}, S_{\ell+1}$.

In the theorem below, we show that the $\ell = (k + 1)$-overlap property is crucial to a dictionary for which k-step greedy parsing is optimal.

Theorem 4.6 (optimality for k-step greedy parsing) *Consider a dictionary construction scheme $\mathcal{P}_d$ such that for all $0 \leq i \leq j \leq n - 1$, the dictionary $\mathcal{D}(i, j)$ satisfies the $(k + 1)$-overlap property. Then for any string T, the k-step greedy parsing method $\mathcal{P}_o$ obtains an optimal parsing with respect to $\mathcal{P}_d$.*

Proof. Let $\mathcal{P}_d$ be a dictionary scheme with the $(k + 1)$-overlap property and let $\mathcal{C}$ be the compression algorithm that uses $\mathcal{P}_d$ and k-step greedy parsing. Let $\mathcal{C}$' be a compression algorithm that uses $\mathcal{P}_d$ with an updated flexible parsing $\mathcal{FP}$', which is defined as follows: given a suffix $T[i:n-1]$, $\mathcal{FP}$' parses the longest prefix $T[i:j-1]$ such that $e_1(j)$ is the maximum among all $e_1(m)$ for which $T[m]$ is in $T[i+1:e_1(i)]$, regardless of whether $T[i:j-1]$ is in $\mathcal{D}$ or not. Then $\mathcal{C}$' represents $T[i:j-1]$ with the 2-tuple $C(T[i:e_1(i)]), (e_1(i)-j)$.

Notice that due to Theorem 3.5, m, the total number of phrases obtained by $\mathcal{C}$' on T, is at most that obtained by an optimal parser for $\mathcal{D}$ on T. Also notice that m is equal to the number of phrases obtained by $\mathcal{C}$ on T due to the $(k+1)$-overlap property. Hence, the number of phrases obtained by $\mathcal{C}$ on T is optimal. $\square$

Robustness: Given two strings T, and T', the *edit distance* between them is defined to be the minimum number of characters that are needed to insert in, delete from, and replace in T, in order to obtain T'. The following lemma states that the compression algorithm $\mathcal{C}$ that uses a given static dictionary $\mathcal{D}$ which satisfies the $(k+1)$-overlap property, and the k-step greedy parser is robust to small changes in input text.

Lemma 4.7 (phrase robustness) *Let $\mathcal{D}$ be a static dictionary satisfying the $(k+1)$-overlap property for some k. Let $\mathcal{C}$ be the compression algorithm which uses $\mathcal{D}$ and k-step greedy parser. The difference in the number of codewords output by $\mathcal{C}$ for any two input strings T and T' whose edit distance is e is $O(e)$.*

Proof. If the edit distance between T and T' is e, then we can write $T = S_1, c_1, S_2, c_2, \ldots, c_e, S_{e+1}$, and $T' = S_1, c'_1, S_2, c'_2, \ldots, c'_e, S_{e+1}$, such that for each k, S_k is a possibly empty substring and c_k and c'_k are different characters, one of which can be null. The proof follows from the fact that for any given substring S_k, if $S_k[i:j]$ is a phrase parsed by $\mathcal{C}$ in T, there can not exist phrase $S_k[i':j']$ parsed by $\mathcal{C}$ in T' for which $i' < i$ and $j < j'$; i.e. $S_k[i:j]$ is a proper substring of $S_k[i':j']$. $\square$

5 Conclusions

We considered the question of optimality in parsing, which is relevant to any dictionary scheme used in dynamic dictionary compression algorithms. Our main result focus on dictionaries that satisfy the prefix property. We show that the standard greedy parsing can be far from optimal in general, and in particular for the important schemes of LZ78 and LZW. On the positive size, we provide a scheme, $\mathcal{FP}$, that is optimal for any given dictionary scheme that has the prefix property. We also present a data structure that can support an implementation that is competitive with that of greedy parsing.

We show, on the other hand, that greedy parsing is optimal for dictionaries with the suffix property, and in particular for the LZ77 scheme. From a practical point of view, the main contribution of the $\mathcal{FP}$ scheme is for inputs for which the LZ78 or LZW schemes provide better compressibility than to LZ77. Candidate classes of inputs include sequences of random (biased) bits, and an archive of DNA sequences. In an experimental research in progress [MRS98] we consider such classes. Our preliminary results indicate that for the pseudo-random sequences, $\mathcal{FP}$ with the LZW dictionary scheme results with about 17% improvement in compressibility over greedy parsing with the same dictionary (UNIX `compress`), and about 33% over LZ77 (UNIX `gzip`). For DNA sequences the respective improvements are about 5% and about 10%. (We expect more extensive results to be available shortly at URL `http://www.math.tau.ac.il/~matias/fp`.)

We identified classes of dictionaries that can be parsed optimally by k-step greedy parsing. It remains as an open problem to find universal parsing schemes for other types of dictionaries. It is interesting to note, however, that any dictionary $\mathcal{D}$ implicitly defines a dictionary $\mathcal{D}$' which consists of all the codewords of $\mathcal{D}$ and all their prefixes (i.e., $\mathcal{D}$' is a prefix dictionary). A codeword in $\mathcal{D}$' can be represented by a pair $\langle \alpha, \ell \rangle$, where α is a codeword from $\mathcal{D}$ and ℓ is the length of the prefix of α. This representation is quite efficient for all for dictionaries in which the number of codewords is significantly larger than the lengths of the codewords. Clearly, the number of phrases in an optimal parsing for $\mathcal{D}$' (which can be obtained using $\mathcal{FP}$) would be at most the number of phrases in an optimal parsing for $\mathcal{D}$.

Acknowledgements

We would like to acknowledge our valuable discussions with S. Even, M. Farach, S. Kannan, S. R. Kosaraju, A. Lempel, G. Manzini, M. Muthukrishnan, M. S. Paterson, S. Savari, J. S. Vitter, and J. Ziv.

References

[BCW89] T. Bell, T. Cleary, and I. Witten. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, December 1989.

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

[BW94] T. Bell and I. Witten. The relationship between greedy parsing and symbolwise text compression. *Journal of the ACM*, 41(4):708–724, July 1994.

[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.

[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.

[KV94] P. Krishnan and J. S. Vitter. Optimal prefetching in the worst case. In *ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[LH87] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19:261–296, 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.

[MRS98] Y. Matias, N. M. Rajpoot, and S. C. Sahinalp. Experimental evaluation of flexible parsing compression. in preparation, 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.

[SR97] J. A. Storer and J. H. Reif. Error resilient optimal data compression. *SIAM Journal of Computing*, 26(3):934–939, August 1997.

[SS82] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

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

[VK91] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *IEEE Symposium on Foundations of Computer Science*, 1991.

[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.

[Ziv97] J. Ziv. Personal communication. 1997.

[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

A.1 Properties of LZ78, LZW and LZ77 algorithms

Lempel Ziv algorithms are the most common lossless compression methods. The LZ78 (or more accurately LZW) scheme is the basis for UNIX `compress` utility and is used in the most popular fax and modem standards. LZ77 algorithm is the basis for all `zip` variants including MS-Windows `win/pk-zip`, and UNIX `gnu-zip`. Both LZ77 and LZ78 algorithms use *greedy parsing* and dynamic dictionary construction methods.

Both the LZ77 and LZ78 (in addition to LZW) algorithms are (1) asymptotically optimal in the information theoretic sense, (2) are very fast, with $O(1)$ processing time per input character, (3) require a single pass over the input, and (4) can be applied on-line. The LZ78 (and the LZW) can be implemented by the use of simple trie data structure whose space complexity is proportional to the size of the output. In contrast LZ77 builds a more complex suffix tree in an on-line fashion, whose space complexity is proportional to the size of the input text.

Although it is generally accepted that LZ77 is usually better for compressing excerpts of natural or computer languages, LZ78 (or LZW) performs better in many other domains. In fact a number of recent theoretical results show that LZ78 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 (and stronger) result for more general, unifilar, sources has been obtained by Savari [Sav97]. For the LZ77 algorithm, this redundancy is as much as $O(\log\log n/\log n)$ [Wyn95]. Another recent result by Kosaraju and Manzini [KM97] states that 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. The LZ78 and LZW algorithms are also of interest in several other contexts, including data prefetching [VK91, KV94], and DNA sequence classification [FNS+95].

We note that there are quite a few algorithms that were proposed to improve the compression ratio attained by the LZ algorithms, and were presented as "on-line" (often not according to our definition). All of these algorithms, including the ones by Miller and Wegman [MW85], and by Yokoo [Yok92], focus on the effect of alternative dictionary construction schemes (see Section 2 for a detailed description of their dictionary construction schemes), and use greedy parsing for output generation.

A.2 Concrete realizations of popular compression algorithms

We provide several concrete realizations of $\mathcal{P}_d$ and $\mathcal{P}_o$ in some well known dictionary compression algorithms. For all these algorithms, $\mathcal{P}_o$ is the greedy parsing. We therefore need only to describe the dictionary parser $\mathcal{P}_d$.

A.2.1 LZW algorithm

In LZW, $\mathcal{D}_0$ consists of all possible single character substrings. The codeword of a single character substring β is $j-1$, where j is the lexicographic order of β in the alphabet Σ. In a given iteration $i+1$, $\mathcal{P}_d$ checks if $T[c(i-1)+1:i]$ is in $\mathcal{D}_i$. Only if it is not the case, then $\mathcal{P}_d$ inserts $T[c(i-1)+1:i]$ in the dictionary with the codeword $|\mathcal{D}_i|$. Otherwise no action is taken.

We note that LZW is an on-line algorithm according to our definition as $\lambda = 1$.

We demonstrate how the LZW algorithm works on an example in Table 7, and Figure 3.

A.2.2 LZ78 algorithm

In the LZ-78 algorithm $\mathcal{D}_0$ again consists of all possible single character substrings. However, the codewords of those substrings are a bit different. Given a phrase, $T[k:l]$, its codeword is a pair consisting of (1) a so-called *enumeration* of $T[k:l-1]$, and (2) the lexicographic rank of $T[l]$ in Σ.

Initially only the null string has an enumeration, which is 0. Therefore the codeword for single character substring β is $(0,j)$, where j is the lexicographic rank of β in Σ. In a given iteration $i+1$, $\mathcal{P}_d$ checks if $T[c(i-1)+1:i]$ appears in $\mathcal{D}_i$. Only if it is not the case, then $\mathcal{P}_d$ enumerates $T[c(i-1)+1:i-1]$, with

$e = \textit{maximum enumeration} + 1$. Finally, for each $\beta \in \Sigma$, $\mathcal{P}_d$ inserts the substring $T[c(i-1)+1 : i-1]\beta$ in $\mathcal{D}$, assigning it the codeword (e, j), where j is the lexicographic rank of β in Σ.

Notice that LZ-78 is an on-line algorithm as $\lambda = 1$.

A.2.3 LZ-77 algorithm

In the LZ-77 algorithm $\mathcal{D}_0$ consists of all characters $\beta \in \Sigma$. The codeword for the character β is the 2-tuple $(|\Sigma| - j, 1)$, where j is the lexicographic order of β in Σ. In a given iteration $i+1$, $\mathcal{P}_d$ identifies the position $T[i]$ as the starting point of $n-i+1$ developing phrases, and assigns them the 2-tuple $(0, 1)$, where the first entry denotes the *relative location* and the second entry denotes the *length* of the phrase. It then identifies the position $T[i]$ as the ending position of $i+1$ developing phrases, whose starting characters are $0, 1, \ldots, i$. Finally, it replaces the codeword (j, k) of each developing phrase in the dictionary with the codeword $(j+1, k+1)$ and each non-developing phrase in the dictionary with the codeword $(j+1, k)$.

A.2.4 Miller-Wegman algorithm

In the Miller Wegman algorithm, $\mathcal{D}_0$ again consists of all possible single character substrings. The codeword of a single character substring β is $j-1$, where j is the lexicographic order of β in the alphabet Σ. In a given iteration $i+1$, $\mathcal{P}_d$ checks if $T[c(i-1)+1 : i]$ is in $\mathcal{D}_i$. If it is not the case then it checks if there exists a $j < i$, such that $T[c(i-1)+1 : j]$ is in $\mathcal{D}_j$, $T[j+1 : i]$ is in $\mathcal{D}_i$, but $T[c(i-1)+1 : j+1]$ is not in $\mathcal{D}_{j+1}$. Only if it is the case, then $\mathcal{P}_d$ inserts $T[c(i-1)+1 : i]$ in the dictionary with the codeword $|\mathcal{D}_i|$. Otherwise no action is taken.

A.2.5 Yokoo algorithm

Yokoo describes several schemes in [Yok92]. In all such schemes, $\mathcal{D}_0$ consists of all possible single character substrings. Again, the codeword of a single character substring β is $j-1$, where j is the lexicographic order of β in the alphabet Σ. In a given iteration i, $\mathcal{P}_d$ checks if $T[c(i-1)+1 : i]$ is in $\mathcal{D}_i$. Only if it is not the case, then in the first scheme described in [Yok92], $\mathcal{P}_d$ inserts each substring $T[j : i]$ for $c(i-1) < j \leq i$, in the dictionary with the codeword $|\mathcal{D}_i| + j - c(i-1) - 1$, unless $T[j : i]$ is already a phrase in the dictionary – otherwise no action is taken. In the latter schemes, among the substrings $T[j : i]$ for $c(i-1) < j \leq i$, only a select few (such as the shortest one which is not in the dictionary) are inserted in the dictionary.

A.3 Examples

We give here some examples for LZW, and LZW-$\mathcal{FP}$ algorithms.

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 1: LZW compression algorithm for $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$. We have $\Sigma = \{a, b\}$, $\mathcal{D}_0 = \{a : 0, b : 0\}$, 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 2: 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 3: 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 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 $i = j = 0$, 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.