CHAPTER 6

Dictionary-Based Data Compression: An Algorithmic Perspective

S. CENK SAHINALP
NASIR M. RAJPOOT

6.1 INTRODUCTION

Dictionary-based methods are among the most popular forms of lossless data compression. Given an input string of characters from a constant size alphabet A (e.g., ascii), a dictionary-based compression algorithm partitions the input string S into non-overlapping substrings which are called *phrases* and replaces each phrase with corresponding bit strings which are called *codewords*; the *dictionary* is this function which maps phrases to codewords. The fundamental task of a dictionary-based method is to partition the input S into phrases whose corresponding codewords have the smallest total bit-length.

Dictionary-based compression is sometimes referred to as *compression with textual substitution*. Textual substitution methods can naturally be classified into four categories according to whether they allow *fixed-* or *variable-length* phrases and *fixed-* and *variable-length* codewords. For us, dictionary-based compression will mean only textual substitution with variable-length phrases; the codewords on the other hand can be fixed or variable length. Note that in compression methods with fixed-length phrases (such as Huffman coding or some of the context sensitive methods), partitioning ceases to be an issue and dictionary construction becomes the main concern; better compression is achieved by replacing more frequent phrases with shorter codewords.

A dictionary-based compression method comprises two relatively independent stages. The first stage is the *dictionary construction*: the set of (potential) substrings of the input are determined as phrases and are given unique codewords. The second stage is *parsing*: the input string is partitioned into phrases which are then replaced with corresponding codewords.

Similar to other compression schemes, dictionary-based compression methods need to satisfy the following two constraints (see Section 3.1 in [22] for a similar set of constraints):

1. **Generality.** Every input string (of interest to the application in question) should be "compressible"; i.e., it should be possible for the algorithm to process every input string. It could the case that the compressed representation is bit-wise longer than the original input.
2. **Unique decompressibility.** Every compressed string should be uniquely decompressible; i.e., the compression algorithm needs to have a corresponding decompression algorithm which can uniquely obtain the original representation of any input string from its compressed representation.

Thus the first goal of a dictionary compression algorithm is to maximize the compression achieved while satisfying the above constraints. The achieved compression can be measured in terms of the *compression ratio*: the ratio of the number of bits in the original representation of the input and the number of bits in its compressed representation. (An alternative measure is the *compression rate* which is defined as $1 - \textit{compression ratio}$.) A secondary goal is to be as efficient as possible in terms of time and space utilization. Clearly, the compression ratio is a function of both the dictionary construction method and the parsing method employed. The overall efficiency of a compression algorithm is usually determined by its efficiency in searching for suitable phrases in the dictionary and hence is related to the *data structure* employed for representing the dictionary. In the following sections, we elaborate on these three main issues, namely, dictionary construction, parsing, and the data structure employed. We also discuss additional constraints on compressed representations of strings such as fault tolerance, searchability, and indexibility. Finally, we provide efficiency/hardness results on select compression/parsing methods and discuss possible uses of multiple dictionaries, as well as compression methods inspired by dictionary methods such as anti-dictionaries. We also present several open problems during our discussions.

6.2 DICTIONARY CONSTRUCTION: STATIC VERSUS DYNAMIC

Dictionary construction methods are usually classified into three basic families: *static, semi-dynamic*, and *dynamic* (some texts prefer to use the term *adaptive* instead of dynamic).

A static dictionary method uses the same dictionary for all input strings; thus such dictionaries are used only in specific applications where the input strings of interest contain many common phrases. A semidynamic dictionary method builds a "static" dictionary by inspecting the input string as a whole; the compression (i.e., parsing) is performed only after the dictionary is constructed in full. For more general applications, dynamic methods are typically used. A dynamic dictionary method updates its dictionary as it parses and compresses the input string; thus parsing and dictionary construction steps usually interleave.

In the following sections we will first motivate dictionary compression by analyzing the properties of static dictionaries, then we will discuss parsing issues for static dictionaries as well as dynamic dictionaries, and finally we present some of the well-known approaches to dynamic dictionary construction.

6.2.1 Static Dictionary Methods

As we indicated earlier, a static dictionary is simply a set of substrings from the input alphabet with corresponding codewords. Ideally the dictionary should consist of phrases common to input strings which are typically encountered in the application domain. Clearly the dictionary used needs to be available to both the compression algorithm and its corresponding decompression algorithm, and, as other dictionary methods, should satisfy the generality and the unique decompressiblity constraints.

The generality constraint can be computationally verified for static dictionaries via the following lemma.

Lemma 6.1. *Any dictionary D consisting of phrases from an input alphabet A satisfies the generality constraint if and only if for any given string S of characters from A there exists a prefix S' of S which is a phrase in D.*

Proof. If there exists a string S for which no prefix is a phrase in the dictionary, clearly S is not compressible by D. If for any string S there exists at least one prefix S' which is a phrase in D, then S must be compressible by D: Let $S = S'S''$; by the initial assumption, the string S'' must have a prefix which is a phrase in D, hence the proof follows. ∎

The above condition which is equivalent to the generality constraint can be verified in $O(|D|)$ time and space, where $|D|$ denotes the total number of characters in all phrases of D.

Unfortunately we are not aware of any similar condition that can efficiently check whether a dictionary satisfies the unique decompressibility constraint. Thus our first open problem follows:

Open Problem 6.1. *Analyze computationally verifiable conditions that are necessary and sufficient for a dictionary to satisfy the generality and the unique decompressibility constraints.*

Nevertheless it is easy to construct dictionaries satisfying the unique decompressibility constraint by the help of the following sufficiency lemma.

Lemma 6.2. *Consider a dictionary D consisting of phrases with corresponding binary substrings as codewords; if no codeword in D is the prefix of another, then D satisfies the unique decompressibility property.*

Proof. If no codeword of D is a prefix of another, then given a binary string S obtained by a compression algorithm that uses D, there is a unique prefix which can be extracted from S and replaced by a corresponding phrase regardless of the parsing method used in the compression algorithm. The proof follows by iteratively applying this argument. ∎

6.2.2 Parsing Issues

As mentioned earlier the ultimate goal of parsing is to partition the input string into phrases whose corresponding codewords have the smallest possible total bitwise length. Although most parsing methods can be employed by both static and dynamic dictionary methods, it is more informative to introduce them in the context of static dictionaries for which optimality results in terms of compression and performance can be given. We will only get to the specific details of parsing for dynamic dictionaries during our discussion on these methods.

For static dictionary methods, the optimal parsing of any input string can achieved by a two-pass dynamic programming algorithm, which first reads the input from left to right to determine the "best" parsing, and then from right to left to replace the phrases obtained with respective codewords, as follows.

Let $D = \{d_1, \ldots, d_k\}$ be the dictionary and its phrases and let c_i be the codeword of phrase d_i; the lengths of d_i and c_i are respectively represented as $|d_i|$ and $|c_i|$. Let the input string be $S = s_1, \ldots, s_n$. The algorithm constructs an array $A[1 : n]$ where $A[i]$ is the length of the optimal compression (i.e., its shortest possible representation by using D) of $s_1, \ldots, s_i$. It also constructs another array $B[1 : n]$, where $B[i]$ is the starting location of the last phrase replaced by a codeword in the optimal compression of $s_1, \ldots, s_i$.

Set $A[0] = 0$ and $A[i] = \infty$ for all $i \neq 0$. The algorithm iteratively computes $A[i]$ by using $A[1, \ldots, i-1]$ as follows. For each i, it iteratively checks for $j = 1, \ldots, k$ whether the phrase

d_j matches $s_{i-|d_j|+1}, s_i$ and replaces the current value of $A[i]$ with $A[i - |d_j|] + |c_j|$ and $B[i]$ with $i - |d_j|$ if the latter is smaller. Once $A[n]$ is computed, all that is needed is to follow the pointers of B iteratively, starting from $B[n]$, while replacing the phrases with their corresponding codewords.

Lemma 6.3. *The above dynamic programming routine obtains the shortest possible representation of any given string S when it is compressed with static dictionary D.*

Proof. Assume the contrary; then there should exist at least one $A[i]$, which does not represent the size of the shortest compressed representation of $s_1, \ldots, s_i$. Consider the smallest such i (i.e., the leftmost position). The dynamic programming method assigns to $A[i]$ the smallest possible $A[i - |d_j| + 1] + |d_j|$ among all phrases d_j which match a suffix of $s_1, \ldots, s_i$. Thus if $A[i]$ is incorrectly computed, then there should exist at least one $h < i$ for which $A[h]$ is incorrectly computed, contradicting the assumption that $A[i]$ is the leftmost incorrectly computed entry of A. ∎

The naive implementation of the above dynamic programming routine runs in $O(n.|D|)$ time (where $|D|$ is the total size of all phrases in the dictionary). There is a more efficient $O(n.\max_i |d_i|)$ time algorithm, where $\max_i |d_i|$ is the size of the longest phrase in the dictionary by the use of a simple trie data structure, which we will discuss later in the chapter. Nevertheless, because many (especially communication related) applications require an on-line processing of the input string and thus are not suitable for two-pass parsing and because there is an ever-increasing demand to compress the input faster, such a dynamic programming approach is often not used in practice. Rather, available methods usually employ specialized dictionaries for which there are on-line parsing methods that achieve optimality or near optimality in a single pass. Such schemes make parsing decisions based on local information about the input string and cannot afford to look ahead or look back for more than a few steps.

Among on-line parsing methods, by far the most popular one (in both static and dynamic methods) is the greedy strategy: While compressing the input from left to right, the greedy method simply parses (and replaces with the corresponding codeword) the longest phrase that matches a prefix of the uncompressed portion of the input. This approach requires no more than a single pass over the input string and uses no information about the "past" or "distant future" and thus the associated delay is limited. Moreover, it enables one to compress a string of size n in optimal $O(n)$ time (optimality follows from the fact that each one of the n characters needs to be read and processed) by the use of a trie data structure to represent the dictionary. The space requirement of this data structure is $O(|D|)$, which is again optimal. Greedy strategy parses any input string to the minimum number of codewords possible (and hence achieves the best possible compression when all codewords are of same length) if the dictionary is suffix complete [3]: A given dictionary D is suffix complete if for every phrase in D, all its suffixes are also phrases in D.

Suffix-complete dictionaries are not very common in practice; one exception is the Lempel–Ziv 1977 method [30], which will be described in more detail in the discussion on dynamic dictionaries. Most practical dictionaries (which are generally dynamic) are prefix complete; i.e., for each phrase in D all its prefixes are also phrases in D. For such dictionaries, compression with greedy parsing can be far from optimal on certain input strings. However, by a slight modification to the greedy rule, it is possible to obtain optimality for prefix-complete dictionaries: *Greedy parsing with one-step lookahead* achieves the smallest number of codewords on any input when used with a prefix-complete dictionary. This variant of greedy parsing is dubbed flexible greedy parsing in [17, 29], which provides optimality results for dynamic dictionaries; optimality results for static dictionaries were provided earlier in [11].

Flexible greedy parsing (similar to the standard greedy parsing) reads the input characters from left to right. In each step, it finds the two phrases whose concatenation matches the longest prefix

of the uncompressed portion of the input. It then replaces the first phrase's occurrence in the input with the corresponding codeword and iterates.

It is possible to implement the flexible greedy parsing in optimal $O(n)$ time via a trie–reverse trie pair in space proportional to the size of the compressed version of the string [17, 29], which is again optimal. It may be possible to improve compression rates while maintaining $O(n)$ time performance by employing dictionaries that can utilize greedy parsing methods with more than a single-step lookahead. In fact the general dynamic programming approach described earlier can be seen as a greedy method with an unlimited number of lookaheads. Unfortunately we are not aware of any compression algorithms in the literature that exploit such an approach.

6.2.3 Semidynamic and Dynamic Dictionary Methods

A static dictionary may also be precomputed by inspecting the input string to be compressed and inserting in the dictionary some of its frequently occurring substrings as phrases. In this context, the dictionary itself (which can also be compressed via several means) should be considered as part of the compressed version of the input string. This *semidynamic* approach, thus, should aim to construct the "best" static dictionary suitable for one or more documents to be compressed. Once the dictionary is constructed via a "training" input string, it is possible to use it later for input strings from the same application domain.

A rigorous treatment by Storer [22] classifies dynamic dictionary methods into four basic *macro schemes* as follows.

All semidynamic dictionary methods can be investigated under the *external pointer macro scheme* (EPM). In this macro scheme, the compression of an input string S is achieved by constructing a "dictionary string" T and then replacing substrings of S with pointers to identical substrings in T. Thus the compressed representation of S is the tuple (T, S'), where S' is the string of pointers to substrings of T, and characters from the original alphabet. Note that it is possible to compress T as well by replacing substrings of T with pointers to other occurrences of the same substring within T.

In a *compressed pointer macro scheme* (CPM), the compressed representation S' of input string S can be decompressed by considering the tuple (S', S') as one obtained by an EPM scheme and thus treating S' as its own dictionary.

In an *original pointer macro scheme* (OPM), the compressed representation S' of input S can be decompressed by replacing in S' each pointer of any given length k, by k pointers of length 1, and then treating it as a CPM scheme.

In an *original external pointer macro scheme* (OEPM), the compressed form of input S consists of a tuple (T', S') where T' is the OPM compressed form of a dictionary T, which is used to EPM compress S.

Storer gives several sufficient conditions for specific implementations of these schemes so that they satisfy the generality and unique decompressibility constraints. He also provides hardness results related to the computational complexity of obtaining optimal compression under specific schemes, in particular the CPM scheme, which is especially important for practical purposes. We will come back to some of these hardness results later in this chapter.

6.2.3.1 Dynamic Dictionary Methods

Almost all practical dynamic dictionary methods are specific implementations of the CPM. At the intuitive level, a dynamic dictionary method aims to "learn" and adaptively insert in the dictionary substrings that are frequently observed in the input. It is possible to insert in the dictionary certain substrings which are not observed in the input; we will give examples of such schemes later in our discussions (on approximate dictionaries, anti-dictionaries, etc.). For now, we focus only on

dictionaries which consist of substrings that are present in the input string. In such schemes, a codeword that replaces a substring of the input can be seen as a *pointer* to another occurrence of the same substring. Thus dynamic dictionary methods can naturally be classified into two categories: *unidirectional*, in which all pointers have the same direction (e.g., they always point left), and *bidirectional*, in which pointers can point both left and right. We discuss unidirectional and bidirectional algorithms separately in the following sections.

6.2.3.2 Unidirectional Methods

Most practical dictionary methods are unidirectional. As mentioned above, these methods use codewords that all point to the same direction (without loss of generality we will assume that this direction is *left*). Thus such methods usually enable one to compress a given input string iteratively, in a *single pass*: Suppose that a prefix of the input is already compressed; then all that is needed to do is to find the "best" prefix of the uncompressed portion of the input which can be replaced by a pointer to an occurrence in the compressed portion of the input. One can readily observe that all *unidirectional* methods enable decompression in a single pass, which is a desired feature for applications in which computational power and memory of the decompressing party (such as in mobile communications) are much lower than those of the sending party. Thus all unidirectional methods that we are aware of are actually single-pass methods in terms of both compression and decompression.

Single-pass methods are especially popular due to the fact that they usually have *on-line* implementations where the goal is to limit the delay between reading a character and outputting a codeword representing the character (within a phrase). Such implementations are crucial in communication applications, which probably explains why pretty much all single-pass methods are on-line implementable. Because all unidirectional methods in the literature are single pass, and all single-pass methods have on-line methods, these terms are sometimes used in place of the others.

In the next few pages we discuss some of the best known unidirectional/single-pass/on-line methods by describing their dictionary construction and parsing methods and the synchronization between these two methods. We also provide some compressibility results, asymptotic and nonasymptotic.

6.2.3.2.1 Some History The general idea of compression by textual substitution is usually attributed to a 1967 paper by White [28], which suggests a generic unidirectional scheme as a concluding remark. This suggestion was realized in a powerful method by Ziv and Lempel in 1977 [30], which we describe in detail later in this section. Their seminal paper proves that their algorithm is asymptotically optimal for input strings which are generated by an *ergodic* source (a very general class of probabilistic sources). The first algorithmic realization of this method required a running time of $O(n^2)$, where n is the number of characters in the input. One year later, Ziv and Lempel [31] came up with an alternative method with similar asymptotic properties with a very straightforward $O(n)$ time algorithmic realization. In 1981, Rodeh *et al.* [19] demonstrated that it was possible to realize the first method in $O(n)$ time as well. Later it was shown independently in [12, 14] and in [20] that the second algorithm gets closer to asymptotic "optimality" faster than the first algorithm.

The Ziv and Lempel methods were popularized by a version of their second algorithm obtained by Welch [26, 27]. Interestingly, it was Welch who obtained the first patent for the Lempel–Ziv-based approaches. This version became the basis of many popular programs including the GIF image compression format, UNIX `compress` and `compact` programs and became part of the popular `V42.bis` modem compression standard. (A newer modem compression standard, V44, also uses an algorithm that can be thought as a combination of the two Ziv–Lempel methods.)

6.2.3.2.2 Lempel–Ziv-77 Method Among the popular data compression methods, the Lempel–Ziv-77 method (denoted LZ77) deserves special attention. Although it is not the first dynamic dictionary method proposed, it clearly is the one which popularized the general approach.

Before presenting the original version of the LZ77 algorithm, we describe a simpler variant by Storer and Szymanski [21] (denoted LZSS). The parsing method employed by the LZSS is greedy (with no lookahead). The dictionary, at each step, simply consists of (1) all single-character substrings and (2) all substrings of the input which start in the already compressed prefix of the input and finish before the end of the phrase to be parsed. Each codeword consists of a bit flag that denotes whether the codeword is of type (1) or (2). Type (1) codewords (of single-character phrases) simply are composed of the respective character; type (2) codewords are composed of two entries: (i) a pointer to the start location of the last occurrence of the phrase and (ii) the length of the phrase. Both the pointer and the length information can be represented by a fixed number of bits or can be represented through Elias (universal) coding. (For brevity we omit some details of the LZSS coding method.) (Universal coding is described in Chapter 3.)

In the original LZ77 method, all codewords are identical in composition. They are composed of the following three entries: (i) a pointer to the start location of the last occurrence of the phrase (set to 0 if no such occurrence exists); (ii) the length of the phrase (set to 0 if no such occurrence exists); and (iii) the first character of the uncompressed portion of the input; the next step of parsing is performed after skipping this character.

A first glance at the LZ77 algorithm may give the impression that the richness of the dictionary may degenerate the compression ratio achieved and running time of the algorithm; two papers show that this is not the case: (i) Ziv and Lempel [30] were able to demonstrate that if an *ergodic* source generates the input string, as the input gets larger, the compression ratio achieved by the LZ77 algorithm gets arbitrarily close to the entropy of the source. Thus, for ergodic sources, the LZ77 algorithm is *asymptotically optimal*. Our focus in this presentation is more on the combinatorial issues; thus we do not provide the details of this result here. (ii) Rodeh *et al.* [19] later showed that an on-line version of the *suffix-tree* data structure [16] enables one to implement the LZ77 algorithm in time and space linear with the input size. We will give further details of this implementation in the Data Structures in Dictionary Compression section.

One final observation is that the LZ77 dictionary construction method is suffix complete (all suffixes of a phrase are by definition phrases) and thus when it is used for compressing a given input, the greedy parsing should achieve the minimum number of phrases possible. Thus, if the lengths of the codewords are all equal, the greedy parsing achieves the best compression possible by the LZ77 dictionary construction. We are not aware of any on-line or off-line parsing scheme that achieves optimality when the LZ77 dictionary is in use under any constraint on the codewords other than being of equal length.

6.2.3.2.3 LZ77 Variants There are many variants of the LZ77 algorithm; for example, a sliding window version does not allow pointers of size more than a user-defined bound. There are also mixed schemes which normally allow only fixed-size windows but may also tolerate longer matches when opportunities for large replacements arise [15]. Other versions try to compress again the sequence of pointer and/or phrase length entries of the codewords separately, by other means of compression (see, for example, [1]), such as Huffman coding (see [2]) and arithmetic coding. Such variants, as well as the original algorithm, have been used in many applications and provided the basis for many popular programs such as all the `zip` variants (e.g., `gzip`, `winzip`, `pkzip`), `PNG` (an image compression format), `arj` (an archive compression format), and `LHarc` (another archive compression program).

6.2.3.2.4 Lempel–Ziv-78 Method The second algorithm by Ziv and Lempel (denoted LZ78) was popularized by a variant reported by Welch [27] and is known as the LZW algorithm. The LZW algorithm is simpler and thus is described first: It uses greedy parsing as per the LZ77 algorithm. The difference is in dictionary construction. The dictionary initially consists only of single-character substrings. After each parsing step, the phrase parsed is concatenated with the first character of the uncompressed portion of the input and is inserted in the dictionary as a new phrase. This phrase is assigned $|D| + 1$ as a codeword (where $|D|$ is the number of phrases in the dictionary). At any given step of the algorithm each phrase is represented by $\lceil \log_2 |D| \rceil$ bits.

The original LZ78 algorithm again uses a variant of greedy parsing. At each step of the algorithm, the longest prefix of the uncompressed portion of the input is parsed and replaced with a codeword; the following character is skipped uncompressed to iteratively continue the compression process. The dictionary at a given step includes all parsed phrases concatenated with the single characters following them, which are left uncompressed.

In their seminal paper, Ziv and Lempel [31] demonstrated that this algorithm also achieves asymptotic optimality when the input is generated by an ergodic source. The advantages of this algorithm are simplicity and the newly discovered property that the compression ratio achieved by the LZ78 algorithm and the LZW variant approaches the *entropy* (a measure of compressibility, which translates into the best asymptotic compression possible by any algorithm) of an ergodic source generating the input faster than that of the LZ77 (see [12, 14, 20]).

6.2.3.2.5 LZ78 Variants The most popular variants of the LZ78 scheme include those used in the UNIX `compress` program, the GIF image compression format, and the `V42.bis` modem standard, which all simply use the LZW method with minor modifications on the dictionary maintenance (see discussion below). Other variants focus on modifying the dictionary construction and parsing method employed.

6.2.3.2.6 Modifications in Dictionary Construction There are many ways to update the dictionary in a way similar to the LZ78 scheme. One immediate modification [9] considers extending the longest parsed phrase by some fixed k characters as well as $k - 1, \ldots, 1$ characters. This is similar to an earlier modification by Miller and Wegman [18] which concatenates the longest parsed phrase with the following longest phrase parsable and inserts it in the dictionary. A combination of these two ideas was considered by Storer so that all prefixes of each phrase inserted in the dictionary by the Miller–Wegman method are inserted in the dictionary as well (see [22]). Later, Horspool considered updating the dictionary by parsing the longest string greedily at each step and inserting in the dictionary with a single-character extension as in the case of LZ78, but before outputting the corresponding codeword, checking out if one can reach further if parsing restarts not at the immediate next character but a few characters back [10]. Because the LZ78 dictionary is prefix complete, one can rather replace with its codeword the prefix which gives the furthest reach in the next step.

6.2.3.2.7 Modifications in Parsing It was shown [17, 29] that flexible greedy parsing, when performed independent of the dictionary construction (which, on any string, constructs the exact same dictionary as that obtained by the LZ78 method), achieves the minimum number of codewords possible by the dynamic dictionary constructed through the LZ78 method. In fact, this optimality result is valid for any dynamic dictionary construction method which maintains the prefix completeness property at all times. It is also shown that one-step lookahead greedy parsing can be implemented within the same time and space bounds of the original LZ78 algorithm and

that the corresponding decompression algorithm also correctly decompresses data compressed by the original LZ78 algorithm.

6.2.3.2.8 *Dictionary Maintenance in Dynamic Methods*

Another important issue in implementations of Lempel–Ziv methods and other dictionary-based algorithms is how to handle memory limitations on the dictionary. One simple solution is to put a bound on the size of the dictionary and start building the dictionary again once it gets filled up. Other methods for cleaning up the dictionary once it is full include deleting the least recently used (LRU), least frequently used (LFU), swap, or freeze heuristics (see, for example, [6], and check [22] for details).

The freeze heuristic simply stops inserting new phrases to the dictionary once it is full and compresses the rest of the input by using the dictionary without any changes.

The LRU heuristic deletes the least recently used phrase/codeword pair from the dictionary. The codeword that belonged to the deleted phrase could be used for a new phrase inserted in the next step. To implement this heuristic, one needs to keep time stamps for every phrase which should be updated every time it is accessed. The time stamps could be maintained in a priority queue, which can efficiently provide which phrase is to be deleted once the dictionary is full.[1] One obvious drawback of the LRU heuristic is the need for extra memory for storing the priority queue.

The LFU heuristic deletes the least frequently used phrase/codeword pair. It is very hard to implement the LFU heuristic if the frequency of a phrase is defined to be the number of times it is accessed, normalized by the number of accesses to the whole dictionary since its insertion; this may require updating the frequency information of every phrase at each step. Another version, in which the normalization is done by the total number of accesses to the dictionary since the compression started, can be implemented through priority queues within the space and time bounds for the LRU heuristic.

A natural heuristic is to simply delete the dictionary when it gets full and start building it from scratch. An alternative for avoiding "forgetting the past" altogether is provided by the swap heuristic, which constructs a primary dictionary while constructing an auxiliary dictionary simultaneously. Only the primary dictionary is used for compression purposes. When the primary dictionary is full, the algorithm deletes the primary dictionary and swaps it with the auxiliary one.

6.2.3.3 *Bidirectional Methods*

Because bidirectional methods allow codewords that point to "future" occurrences of phrases, they are off-line by nature and are usually slower than their on-line siblings. The benefit is usually improved compression via more careful parsing.

A very important issue in bidirectional parsing is decompressibility: It should be possible to start out with the uncompressed portions of the compressed string and deduce the compressed portions without ambiguity. One can impose a number of restrictions on a bidirectional method to ensure decompressibility (see Section 5.1 in [22] for a similar set of restrictions). One natural restriction is allowing no circular dependencies of size 2; under this restriction, there cannot exist two pointers each of which points to (portions of) substrings represented by the other. This can be generalized to a restriction on circular dependencies of size k, under which one cannot start from a pointer and reach one of its substrings by following a chain of k or more pointers.

[1] For n phrases, such a priority queue can be implemented in $O(n)$ space, $\Omega(\log n)$ time for at least one of the following operations: (i) updating a phrase; (ii) deleting the least recently used phrase; or (iii) inserting a new phrase.

6.3 EXTENSIONS OF DICTIONARY METHODS FOR COMPRESSING BIOMOLECULAR SEQUENCES

Recently, there has been a dramatic increase in interest in processing biomolecular sequences such as DNA, RNA, and protein sequences. Along with this surge in interest comes a massive amount of biomolecular information that needs to be stored and transmitted across the networks efficiently, and thus arises the crucial need for compression. These sequences may typically contain information about proteins or genes or the complete genetic information about a living organism. The human genome, for example, contains the sequence information of all 23 pairs of human chromosomes and spreads across approximately three billion characters from the alphabet {A, C, G, T} representing the four nucleotides. The protein sequences, on the other hand, are much shorter (with an average length of approximately 300) and are composed of symbols from the 20-character amino acid alphabet. Also of interest are structured and semistructured Web documents related to the health sciences in HTML, XML, and the newly emerging BIOXML formats.

The first studies aimed toward understanding the composition of the human genome seemed to suggest that the distribution of nucleotides and amino acids was uniformly random, which would imply that extracting some non-negligible compression out of (portions of) the genome would be a very difficult task. Moreover, the well-known dictionary compression programs (such as `gzip` and `compress`) usually achieve negative compression rates on such sequences. Here we describe two dictionary-based compression methods which have been shown to produce not only positive compression rates but also reasonable amounts of compression: *biocompress* and *GenCompress*.

6.3.1 The *Biocompress* Program

Grumbach and Tahi [7] exploit the fact that the human genome consists of many exact *tandem* repeats and *complementary palindromes*. A tandem repeat is a concatenation of many copies of a short sequence. A complementary palindrome, on the other hand, is a sequence of length n where the ith nucleotide is complementary to the $(n - i)$th nucleotide (nucleotide pairs A–T and G–C are complementary) and thus lead to *hairpin* structures. These structures make dictionary-based compression possible, especially when reversals and complements of phrases are allowed. The *biocompress* and *biocompress-2* programs attempt to achieve this via an LZ77-based approach by maintaining all observed sequences of size smaller than a user-determined bound on a complete 4-ary tree. The preprocessed tree enables one to search for reversed complements of phrases in addition to standard exact searches. The *biocompress-2* program attempts to achieve further compression via a second-order arithmetic coder to encode areas of the input that contain no repetitions. On tandem repeat- and reverse palindrome-rich regions of the human genome, these programs are reported to achieve compression rates of up to 32%.

6.3.2 The *GenCompress* Program

A more recent work by Chen *et al.* [4] makes use of the fact that the human genome is rich with sequence duplications within a small percentage of edit differences. These differences are usually single nucleotide replacements but can also be in the form of omissions or additions of nucleotides. The *GenCompress* program is thus based on finding approximate matches of the uncompressed portions of the input in the already compressed part of the genome sequence of interest in the LZ77 fashion. It finds the longest prefix of the uncompressed portion of the input which occurs in the compressed portion with a single-edit difference and replaces it with an appropriate codeword

that encodes not only the location and the length of the match but also the position and character value of the difference. The *GenCompress* program achieves compression rates of up to 44% on sequences mentioned in the context of *biocompress*.

6.4 DATA STRUCTURES IN DICTIONARY COMPRESSION

Data structures and their efficient implementations play a key role in determining the time and space complexity of a data compression algorithm. A data structure for implementing a dictionary should be able to store common phrases in a space-efficient way. It should also allow the efficient performance of the following operations: *insertion* of a new phrase into the dictionary, *searching* for a given substring and returning its corresponding codeword if it is present in the dictionary as a phrase, and occasional *deletion* of an already existing phrase from the dictionary. In some dictionary-based compression methods, it is also required that the data structure supports two additional operations, namely, *extend* and *contract* (see, for example, [29]). Given a phrase S in the dictionary D, the operation *extend*(S, a) finds out whether the concatenation of S and a is a phrase in D, whereas the operation *contract*(S) finds out whether the suffix $S[2 : |S|]$ already exists in the dictionary. We will discuss a number of fundamental data structures commonly employed in the dictionary-based data compression methods we described. We start with the description of the most basic trie data structure followed by suffix trees [16], trie–reverse trie pairs [17], and hash tables particularly for efficient implementations of Karp–Rabin fingerprints [13].

6.4.1 Tries and Compact Tries

A *trie* T is a labeled tree in which (i) each edge of a trie is labeled with a single character from the input alphabet Σ; (ii) sibling edges have different labels; (iii) each phrase P is represented by a node n_P in the trie; the phrase P itself can be obtained by concatenating the labels of the edges on the path from the root to node n_P. Clearly each prefix $P[1 : i]$ of any given phrase P corresponds to the ith internal node on the path from the root to n_P. Both the insertion of a substring P in a trie T and the searching of P in T can be trivially performed in $O(|P|)$ time when $|\Sigma|$ is a small constant. For larger alphabets a slightly more involved implementation can perform searches and insertions in $O(|S| \log |\Sigma|)$ time. A *compact trie* improves the space efficiency of a trie by allowing each edge to be labeled by more than a single character while making sure that the first characters of the labels of sibling edges are different.

6.4.2 Suffix Trees

The *suffix tree* of a given string S is the compact trie $T(S)$ of all the suffixes of S. Each leaf in $T(S)$ thus represents a suffix and so a position in S. The simple observation that makes the suffix tree a powerful data structure for a string search is that any substring of S is a prefix of a suffix of S. Thus to find *all* occurrences of (i.e., substrings that are identical to) a given pattern P in S, one needs to start at the root of $T(S)$ and follow the path labeled with characters of P to reach a node r. The leaves (each of which corresponds to a position in S) in the subtree of r give the positions of the occurrences of P in S.

Suffix trees were introduced by Weiner [25]. Since then various algorithms have been presented in the literature to construct a suffix tree for a given string S in time and space linear with $|S|$. These algorithms include [5, 16, 24] as well as [8, 23], which were originally designed for parallel machine models but provide optimal serial implementations. The original linear time algorithm

of McCreight [16] was not designed for an on-line construction of the suffix tree. Rodeh *et al.* [19] described an on-line version of the McCreight method for using suffix trees for an efficient implementation of the LZ77 method. With this approach it is possible to find the longest prefix match of the uncompressed portion of the input S in its already compressed portion in time proportional to the size of the prefix matched. Because the construction of the suffix tree takes an overall time of $O(|S|)$, the total time for executing the LZ77 algorithm on S is optimal $O(|S|)$.

6.4.3 Trie–Reverse Trie Pairs

For one-pass dictionary-based data compression algorithms employing greedy parsing, tries provide a simple and powerful mechanism for finding the longest prefix matches. If one wants to incorporate one or more steps of lookahead, however, tries do not necessarily provide the most efficient method for dictionary maintenance. In particular, greedy parsing with one-step lookahead requires finding a prefix match which provides the farthest reach in the next step. This is possible if one can perform not only prefix extensions but also suffix contractions during the search.

A trie–reverse trie pair [17, 29] is composed of the (compressed) trie T of all phrases in the dictionary, a separate trie T^R of *reverses* of all phrases in the dictionary (the reverse of string $S[1], S[2], \ldots, S[k]$ is $S[k], S[k-1], \ldots, S[1]$), and pointers between the pair of nodes r and r^R that correspond to the phrase P in T and its reverse P^R in T^R, respectively.

As in the case of a simple trie, the insertion of a phrase P into this data structure takes $O(|S|)$ time. Given a dictionary phrase P, and the node r which represents P in T, one can find out whether the substring obtained by extending P with any character a is in D, by checking out if there is an edge from r with corresponding character a. Thus such an *extension* operation can be done in $O(1)$ time. To find out whether the suffix $P[2 : |P|]$ of P is a phrase in the dictionary, one needs to simply follow the pointer from r to r^R (which will effectively reverse the phrase P) and move one character up in T^R and check whether the reverse of the substring that corresponds to this node ($P[2 : |P|]$) is a phrase in D. Clearly this *contraction* operation can be performed in $O(1)$ time.

One can perform successive extension and contraction operations to find the two consecutive prefixes that reach the farthest as follows. Let P^0 be the longest prefix of S^U, the uncompressed portion of S, which is a phrase. For $i = 1, 2, \ldots, |P^0|$, one can find P^i, the longest prefix of $S^U[i+1 : |S^U|] = P^i$ which is a phrase, iteratively as follows. Start from the node r that was reached in the previous iteration (it should represent P^{i-1} or some earlier P^j). Contract by exactly one character through the reverse trie and extend by as many characters as possible through the trie to reach a new node q. If an extension is not possible, assign $q = r$ and make a note through which P^j the node r was reached. Iterate.

The prefix P^j which is noted in the final iteration (at which $i = |P^0|$) is the phrase to be parsed by the one-step lookahead greedy parsing.

6.4.4 Karp–Rabin Fingerprints

Karp and Rabin [13] introduced fingerprints for simple and efficient string search via hashing, enabling one to perform both expand and contract operations as per the trie–reverse trie pair.

In the context of dictionary-based compression one can employ a hash table H of size p for a sufficiently large random prime number p to store all phrases in D. This is done by treating each phrase P as a number in base $|\Sigma|$ and using the hash function $h(P) = P \pmod{p}$. Clearly inserting and deleting a phrase P in the hash table takes optimal $O(|P|)$ time. It is also possible to implement contract and expand operations in $O(1)$ expected time each [29] in the spirit of the

trie–reverse trie pair as follows. Given string Q it is possible to compute the hash value of string Qa, the concatenation of Q with an arbitrary character a, from the hash value $h(Q)$ in $O(1)$ time by using the fact that $h(Qa) = [h(Q) \cdot |\Sigma| + a] \pmod{p}$. Similarly one can compute the hash value of string $Q[2 : |Q|]$ in $O(1)$ time via the fact that $h(Q[2 : |Q|]) = [h(Q) - |Q| \cdot |\Sigma| \cdot a] \pmod{p}$.

6.5 BENCHMARK PROGRAMS AND STANDARDS

Many variants of the LZ77 and LZ78 methods have become benchmarks for dictionary-based compression. Almost all the archiving programs such as `zip`, `PKZip`, `LHArc`, `ARJ`, and `gzip` make use of the LZ77 algorithm (or its LZSS variant) after the files are merged together in a reversible fashion. Below we describe some of the better known benchmark compression programs and some compression standards based on the dictionary compression paradigm.

6.5.1 The `gzip` Program

The `gzip` compression program was distributed by the Free Software Foundation as a utility for the GNU operating system. It provides a modification of the LZSS compression method, often giving good compression results in a reasonable amount of time. The main innovation in `gzip` lies in searching for the "best" match in the uncompressed portion of the text. For this purpose `gzip` builds a hash table to store three-character phrases observed in the compressed part of the input. The hash table stores (the location of) every occurrence of each three-character phrase in a linked list where the order of the occurrences is determined in a move-to-front fashion (i.e., the more recently an occurrence has been accessed, the closer to the top of the link list it is located). This approach improves the search time for cases where more recently accessed three-character phrases (and characters following them in the input string) have a higher chance of being accessed again. It is possible to limit the length of the linked lists corresponding to each hash value so as to put an upper bound on the time to perform a search with a given three-character phrase. This, however, may result in a decrease in compression performance.

The `gzip` program is also capable of performing greedy parsing with one-step lookahead and return not the longest phrase but one which is shorter if the next phrase obtained is composed of a single character. Note that for the general implementation of LZ77, such a lookahead cannot provide any improvement in compression; this becomes an issue in `gzip` only because it does not maintain all the dictionary entries (i.e., all substrings of the compressed portion of the input) implicitly suggested by LZ77. The `gzip` program further tries to improve the compression performance of LZ77 by Huffman coding the *offset* and *length* entries of each codeword separately.

6.5.2 The `compress` Program

Developed as a UNIX operating system utility, the `compress` program is essentially based on the LZW compression method. The dictionary is initialized with 256 entries corresponding to the possible combinations of 8 bits (that is, that this program processes the file to be compressed in a byte-wise fashion). The maximum size of the dictionary is variable, so it could be set to a value in the range $2^9, 2^{10}, \ldots, 2^{16}$. At any step of the execution of the algorithm each codeword is encoded by $b = \lceil \log_2 (|D| + 1) \rceil$ bits, where $|D|$ is the size of the dictionary. Once the dictionary size reaches the upper limit, a statistical check is performed at regular intervals so as to determine whether reasonable compression is being achieved without any alterations in the existing state of the dictionary. If the compression ratio falls below a preset threshold, the dictionary is reset to the initial state.

6.5.3 The GIF Image Compression Standard

Although it was developed more than a decade ago by CompuServe, the GIF compression standard still remains the most widely accepted lossless image compression standard and is used extensively in many applications and on the Internet. Based mainly on the LZW compression method, it employs the compress-like variable bit encoding of codewords while enforcing a maximum dictionary size of 4096. The dictionary is initialized with 2^r entries corresponding to all possible combinations of r bits, where r is the minimum number of bits required to represent any pixel in the image. The next two entries contain the *clear code* and the *end of image* codewords. Thus the number of bits b required to encode a codeword is set initially to $r + 1$. As the compression proceeds and new entries are continuously added to the dictionary, the value of b is recomputed to ensure that no more than sufficient bits are being used to encode the codewords. The dictionary remains static once it becomes full unless a clear code is sent to the decoder to signal the resetting of the dictionary and the initialization of all the relevant parameters. This compression method is particularly useful in lossless compression of synthetic (i.e., computer-generated) images which consist mainly of smooth regions of same color intensity values. Since the image is scanned in a horizontal direction such that two vertically oriented neighboring pixels in an image are set apart by the number of columns of the image, this compression scheme exploits only horizontal redundancies and not the vertical redundancies.

6.5.4 Modem Compression Standards: v.42bis and v.44

It is a well-known fact that the effective speed of signal transmission through a modem depends largely on the data compression scheme used therein. Compression methods based on dictionaries have been at the heart of two modem compression standards: the older and widely accepted v.42bis and the newer v.44. The telecommunications standardization division of the International Telecommunication Union (formerly CCITT) has approved a recommendation v.44 for data compression procedures over telephone lines recently (in November 2000). Where the v.42bis standard was based on the LZW compression scheme patented by the Unisys Corp., the new v.44 recommendation is based on the lesser known Lempel–Ziv–Jeff–Heath (LZJH) compression method patented by the Hughes Network Systems. The LZJH method can basically be regarded as an LZ78 variant with the provision of extending a phrase by its first few symbols. The v.44 standard appears to be applicable to a variety of communication applications and is claimed to be particularly efficient for typical Internet downloads such as HTML files and images.

6.6 REFERENCES

1. Bell, T. C., 1986. Better opm/l text compression. *IEEE Transactions on Communications*, Vol. COM-34, pp. 1176–1182.
2. Brent, R. P., 1987. A linear algorithm for data compression. *Australian Computing Journal*, Vol. 19, pp. 64–68.
3. Cohn, M., and R. Khazan, 1996. Parsing with suffix and prefix dictionaries. In *IEEE Data Compression Conference*.
4. Chen, X., S. Kwong, and M. Li, 2000. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB-00), April 8–11 2000.* (R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, Eds.), p. 107, ACM Press, New York.
5. Farach, M., 1997. Alphabet independent suffix tree construction. In *IEEE Symposium on Foundations of Computer Science.*

6. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, pp. 490–505.
7. Grumbach, S., and F. Tahi, 1994. A new challenge for compression algorithms: Genetic sequences. *Inf. Proc. and Management*, Vol. 30, No. 6, pp. 875–886.
8. Hariharan, R., 1997. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, Vol. 55, No. 1, pp. 44–69, August 1997.
9. Horspool, R. N., 1991. Improving lzw. In *IEEE Data Compression Conference.*
10. Horspool, R. N., 1995. The effect of non-greedy parsing in Ziv–Lempel compression methods. In *IEEE Data Compression Conference.*
11. Hartman, A., and M. Rodeh, 1985. Optimal parsing of strings. *Combinatorial Algorithms on Words*, pp. 155–167.
12. Jacquet, P., and W. Szpankowski, 1995. Asymptotic behavior of the Lempel–Ziv parsing scheme and digital search trees. *Theoretical Computer Science*, Vol. 144, pp. 161–197.
13. Karp, R., and M. O. Rabin, 1987. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, Vol. 31, No. 2, pp. 249–260.
14. Louchard, G., and W. Szpankowski, 1995. Average profile and limiting distribution for a phrase size in the Lempel–Ziv parsing algorithm. *IEEE Transactions on Information Theory*, Vol. 41, No. 2, pp. 478–488, March 1995.
15. McIlroy, M. D., and J. L. Bentley, 1999. Data compression using long common strings. In *IEEE Data Compression Conference.*
16. McCreight, E. M, 1976. A space economical suffix tree construction algorithm. *Journal of the ACM*, Vol. 23, No. 2, pp. 262–272, April 1976.
17. Matias, Y., and S. C. Sahinalp, 1999. On optimality of parsing in dynamic dictionary based data compression. In *ACM-SIAM Symposium on Discrete Algorithms.*
18. Miller, V. S., and M. N. Wegman, 1985. Variations on a theme by Lempel and Ziv. *Combinatorial Algorithms on Words*, pp. 131–140.
19. Rodeh, M., V. Pratt, and S. Even, 1981. Linear algorithm for data compression via string matching. *Journal of the ACM*, Vol. 28, No. 1, pp. 16–24, January 1981.
20. Savari, S., 1997. Redundancy of the Lempel–Ziv incremental parsing rule. In *IEEE Data Compression Conference.*
21. Storer, J. A., and T. G. Szymanski, 1982. Data compression via textual substitution. *Journal of the ACM*, Vol. 29, No. 4, pp. 928–951, October 1982.
22. Storer, J. A., 1998. *Data Compression: Methods and Theory.* Computer Science Press, 1988.
23. Sahinalp, S. C., and U. Vishkin, 1994. Symmetry breaking for suffix tree construction. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing: Montreal, Quebec, Canada, May 23–25, 1994*, pp. 300–309, ACM Press, New York.
24. Ukkonen, E. 1995. On-line construction of suffix-trees. *Algorithmica*, Vol. 14, No. 3, pp. 249–260. [TR A-1993-1, Department of Computer Science, University of Helsinki, Finland]
25. Weiner, P., 1973. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pp. 1–11.
26. Welch, T. A., 1984. A technique for high-performance data compression. *IEEE Computer*, pp. 8–19, January 1984.
27. Welch, T. A., 1985. U.S. Patent 4,558,302, December 1985.
28. White, H. E., 1967. Printed English compression by dictionary encoding. *Proceedings of the IEEE*, Vol. 55, pp. 390–396.
29. Sahinalp, S. C., Y. Matias, and N. Rajpoot, 2001. The effect of flexible parsing for dynamic dictionary based data compression. *ACM Journal of Experimental Algorithmics.*
30. Ziv, J., and A. Lempel, 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. IT-23 No. 3, pp. 337–343, May 1977.
31. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. IT-24 No. 5, pp. 530–536, September 1978.