ERROR-RESILIENT OPTIMAL DATA COMPRESSION*

JAMES A. STORER† AND JOHN H. REIF‡

Abstract. The problem of communication and computation in the presence of errors is difficult, and general solutions can be time consuming and inflexible (particularly when implemented with a prescribed error detection/correction). A reasonable approach is to investigate reliable communication in carefully selected areas of fundamental interest where specific solutions may be more practical than general purpose techniques.

In this paper, we study the problem of error-resilient communication and computation in a particularly challenging area, *adaptive lossless data compression*, where the devastating effect of error propagation is a long-standing open problem that was posed in the papers of Lempel and Ziv in the late 1970s. In fact, the non-error resilience of adaptive data compression has been a practical drawback of its use in many applications. Protocols that require the receiver to request retransmission from the sender when an error is detected can be impractical for many applications where such two-way communication is not possible or is self-defeating (e.g., with data compression, retransmission may be tantamount to losing the data that could have been transmitted in the mean time). In addition, bits of encoded data that are corrupted while data is in storage will in general not be recoverable and may corrupt the entire decompressed file. By *error resilience*, we mean that even though errors may not be detected, there are strong guarantees that their effects will not propagate.

Our main result is a provable error-resilient adaptive lossless data-compression algorithm which nevertheless maintains optimal compression over the usual input distributions (e.g., stationary ergodic sources). We state our result in the context of a more general model that we call *dynamic dictionary communication*, where a sender and receiver work in a "lock-step" cooperation to maintain identical copies of a *dictionary* D that is constantly changing. For lossless data compression, the dictionary stores a set of strings that have been seen in the past and data is compressed by sending only indices of strings over the channel. Other applications of our model include robotics (e.g., remote terrain mapping) and computational learning theory.

Key words. data compression, adaptive algorithm, communication channel, error propagation

AMS subject classification. 68Q25

PII. S0097539792240789

1. Introduction.

1.1. Dynamic dictionary communication. We use the term *dynamic dictionary* communication to refer to the process of transmitting data over a communication channel that is encoded/decoded with respect to dynamically changing dictionary of data. As depicted in Figure 1, a sender generates an *input stream* of characters drawn from a fixed finite-input alphabet, which are encoded, transmitted over a communication channel (or saved on a storage device), decoded, and received as the *output stream*. The encoder and decoder cooperate to maintain identical copies of a *dictionary* D that is constantly changing. The encoder reads characters from the input stream that form an entry of D, transmits the index of this entry, and updates D with some method that depends only on the current contents of D and the current match. Similarly, the decoder repeatedly receives an index, retrieves the corresponding entry of D to write to the output stream, and then performs the same algorithm as the encoder to update D. Note that although both the encoding and decoding algorithms refer to their dictionary as D, errors on the communication channel or storage medium

†Computer Science Department, Brandeis University, Waltham, MA 02254 (storer@cs.brandeis.edu).

‡Computer Science Department, Duke University, Durham, NC 27707 (reif@cs.duke.edu).

Preprint of paper appearing in SIAM Journal of Computing (SICOMP), Vol 26, Num 4, July 1997, pp 934-939.

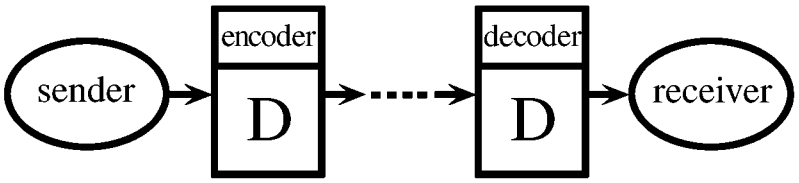

FIG. 1.1. *Dynamic dictionary communication.*

may cause the decoder's dictionary to differ from that of the encoder. Note also that D contains a set of entries (so when we talk about adding a new entry to D, we always mean to add it if it is not already present).

1.2. Applications of dynamic dictionary communication. The major application of dynamic dictionary communication that motivates this work is adaptive lossless data compression by textual substitution (Storer and Szymanski [1978]), where the dictionary stores a set of strings that have been seen in the past and data is compressed by sending only indices of strings over the channel; such compression algorithms are often called "LZ algorithms" after the work of Lempel and Ziv [1976] and Ziv and Lempel [1977, 1978]. (See Storer [1988] for an introduction to the subject and references to the literature.) Other possible applications of dynamic dictionary communication include computational learning theory and robotics (e.g., reporting of data by an autonomous remote robot that is mapping unexplored terrain and transmitting coordinates as displacements from previous locations).

1.3. Error resilience. A potential drawback of dynamic dictionary communication is *error propagation*; that is, a single error on the communication channel (e.g., insertion, deletion, or change of an index sent on the channel) could cause the dictionaries of the encoder and decoder to differ, which in the worst case could corrupt all data to follow. That is, since decoding is guaranteed to be correct only when the dictionary remains the same as the one used to encode the data, there is no way to bound the effects of a single error in the worst case. With *one-way* communication channels, where the decoder cannot send messages to the encoder, the problem becomes even more critical. In addition, with many existing communication systems where the full bandwidth of the channel is consumed, even if a two-way channel is available to let the encoder know that an error has occurred, retransmission of data can often be tantamount to losing new data that could have been transmitted during the time used to retransmit the old data. Retransmission can also introduce further error propagation problems. In addition, a data-storage device where data may be corrupted during storage can be viewed as a one-way communication channel.

In some sense, dynamic dictionary communication can be viewed as "raising the stakes" for the effects of errors on a one-way channel; even if the chance of an error is made very small via a standard error-detection/correction protocol, if an error does occur on a single data item, it can have the catastrophic effect of corrupting an unbounded number of additional data items. We present a technique for "error resilience," where no attempt is made to detect or correct errors but strong guarantees are provided that errors do not propagate. This technique can be combined with standard error-detection/correction protocols to yield one-way dynamic dictionary communication with a low rate of errors that do not propagate.

1.4. Outline. The next section contains basic definitions; it formally defines the dynamic dictionary communication model, which assumes several axioms, reviews

how data compression by textual substitution fits into the model and shows that the axioms hold for key methods that are provably optimal for stationary ergodic sources (the standard assumption in data compression literature), presents a model for errors on the communication channel, and defines error-resilient communication. Section 3 then presents a scheme for protecting against error propagation from k errors for any constant k under any distribution of errors; we make no attempt to analyze how this scheme affects the amount of compression achieved (although it appears to be quite practical). In section 4, we employ randomized techniques to "expand" the k-error technique to give propagation protection with very high probability against a fixed uniformly and independently distributed error rate of probability $1/r$; in particular, for any $k \geq 1$, the probability of error propagation can be made less than $1/r^k$. For example, an error rate of $1/10^{12}$ (a relatively "clean" communication channel with a low-overhead error-correcting mechanism) can be effectively "damped" to $1/10^{48}$ by choosing $k = 4$. In addition, the generalized k-error protocol presented in section 4 has no asymptotic affect on the amount of compression achieved. Section 5 discusses practical considerations.

2. Basic definitions.

2.1. Dynamic dictionary communication. Throughout this paper, when discussing dynamic dictionary communication with respect to a dictionary D, we use the following notation:

- $|D|$ = the current number of entries in D.
- $|D_{\max}|$ = the maximum number of entries that D may contain.
- $|s|$ = the number of characters in the string s.
- $BITS(i)$ = the number of bits needed to write i in binary (i.e., $\text{BITS}(i) = \lfloor \log_2(i) \rfloor + 1$).

Generic encoding and decoding algorithms for dynamic dictionary communication are shown below. Indices of D run from 0 to $|D| - 1$ and are represented using exactly $\text{BITS}(|D| - 1)$ bits, although in practice it is often the case that for simplicity $\text{BITS}(|D_{\max}| - 1)$ bits are used for all indices. We refer to codes sent over the channel that represent indices as *pointers*.

GENERIC ENCODING ALGORITHM.

1. Initialize the local dictionary D with one entry for each character of the input alphabet.
2. **repeat forever**
   - A. {Get the current match pointer p:}
     Use a *match method* to read a string s from the input.
     p = the index of s in D.
     Transmit p using $\text{BITS}(|D| - 1)$ bits.
   - B. {Update D:}
     Add each of the strings specified by an *update method* to D.

GENERIC DECODING ALGORITHM.

1. Initialize the local dictionary D by performing step 1 of the encoding algorithm.
2. **repeat forever**
   - A. {Get the current match string s:}
     Receive $\text{BITS}(|D| - 1)$ bits for the current match pointer p.
     s = the string in D corresponding to p.
     Output s.
   - B. {Update D:}
     Perform step 2B of the encoding algorithm.

The *match heuristic* reads from the input stream a string that is in the dictionary. (It must always be possible to read such a string since all strings of length 1 are always in the dictionary.) After the match has been encoded by the encoder or decoded by the decoder, "learning" consists of modifying the dictionary by adding one or more new strings with an update heuristic. Although it is possible to have the update heuristic employ a deletion heuristic that removes old entries to make room for new ones (see Storer [1988]), we restrict our attention here to update heuristics that do nothing once the dictionary is full so that encoding and decoding continue indefinitely (or until the system is restarted) without any further modification of the dictionary.

The exact choice of the match and update heuristics is not important for the work described here as long as the following axioms are satisfied. (We shall show shortly that these axioms hold for a variety of optimal adaptive lossless data compression methods.)

Succinctness. Except for the dictionary initialization (where entries for each of the characters of the input alphabet are formed), the string (or strings) added by the update heuristic at a given step depend only on the current match and the previous match.

Robustness.

A. There is a constant $1 \leq \rho$ such if the update heuristic adds a new match based on a particular successive pair of matches, it will do so after this successive pair has occurred at most ρ times.

B. There is a constant $1 \leq \alpha$, called the *learning constant*, such that the encoder dictionary reaches size $|D_{\max}|$ after at most $\alpha\rho|D_{\max}|$ entries have been transmitted.

The succinctness axiom addresses how dictionary entries may be formed. The robustness axiom allows the system to be "conservative" and to not form a new entry until it has "proved" itself by appearing a number of times, but not too conservative because this number of times must be bounded by a constant. In addition, the learning constant ensures that progress is made toward filling up the dictionary as data is encoded (and we don't waste too much effort relearning entries that have already been added earlier). Note that condition B of the robustness axiom is somewhat pessimistic because for all of the data-compression methods that we shall consider, if ρ, $\alpha > 1$, then the encoder dictionary reaches size $|D_{\max}|$ after at most $(\alpha + \rho)|D_{\max}|$ entries have been transmitted.

For technical reasons to be discussed shortly, our results will also work for a variation of the succinctness axiom.

Succinctness, version 2. Except for the entries that contain the characters of the input alphabet, the string (or strings) added by the update heuristic at a given step depend only on the current match and the next character.

The second version of the succinctness axiom is similar to the first version except that it is "one step ahead"; it models certain update heuristics that are important for proofs of optimality in applications involving data compression. Both versions have the property that except for the characters of the alphabet, each entry of the dictionary can be constructed from a pair of pointers to two other entries; this is all that will really be needed in our construction (and, as we shall see shortly, suffices for practical data-compression algorithms).

2.2. Applications to data compression. As mentioned earlier, the major application that motivates this work is data compression by textual substitution. In this section, we review a number of textual-substitution algorithms and show that

they can be modified to satisfy the robustness axiom without sacrificing compression. Our definition of "optimal" compression is *optimal in the information-theoretic sense*; that is, for stationary ergodic sources of entropy H, after processing n characters, the method in question sends an average of $H+\varepsilon(n)$ bits per character, where $\varepsilon(n)$ goes to 0 as n goes to ∞. (See Cover and Thomas [1991] or Gallager [1968] for definitions of entropy, etc.) This asymptotic measure of performance for stationary ergodic sources is standard throughout the literature. Furthermore, the methods we discuss here work well in practice for virtually any source. (See Storer [1988] for a detailed presentation of these methods as well as experimental results.)

Typically, the match heuristic used is the *greedy heuristic*; that is, read the longest match possible. Although other heuristics can be used and the greedy match heuristic does not guarantee the best possible compression on finite strings for any of the update heuristics to be discussed below (Storer [1988]), it does perform well in practice and is used in all of the methods that are provably optimal in the information-theoretic sense. Examples of update heuristics that might be used for data compression are as follows.

Uncompressed character (UC). Add the current match concatenated with the next character of the input; the next character of the input is sent along in uncompressed form as part of the current pointer (so that the next match starts after the character following the current match). This heuristic does not fit exactly into the generic encoding and decoding algorithms, but they can easily be modified to accommodate it by not allowing a match to a single character until it has been sent to the decoder. (The cost of dictionary entries "wasted" by characters that have yet to be seen becomes arbitrarily low as the dictionary size increases.) For finite-length strings and dictionaries of bounded size, UC typically does not perform as well in practice as the NC or FC heuristics discussed below, but we include it because it reflects the construction of Ziv and Lempel [1978] that is provably optimal.

Next character (NC). Add the current match concatenated with the next character of the input. This can be viewed as a more practical implementation of UC. It also requires modifications of the generic encoding and decoding algorithms because the next character of the input stream cannot be deduced by the decoder until the following match is received. (For the special "glitch" where the string matched at the current step is the one formed at the previous step, the decoder can deduce that the first and last character of the current match are the same as the first character of the previous match.) This is the heuristic used by Welch [1984] (upon which UNIX "compress" command is based) and Miller and Wegman [1985].

First character (FC). Add the last match concatenated with the first character of the current match. This heuristic performs similarly to NC in practice, but fits cleanly into the generic encoding and decoding algorithms (and does not have the decoding "glitch" mentioned above); it is discussed in Storer [1988].

Current match (CM). Add the last match concatenated with the current match. This heuristic is discussed by Miller and Wegman [1985] and Seery and Ziv [1977, 1978]. A variant of this method is employed by Reif and Storer [1990, 1992] and Royals et al. [1993] in a massively parallel systolic custom VLSI design.

All prefixes (AP). Add the set of strings consisting of the last match concatenated with each of the prefixes of the current match. This heuristic has the fast-growing characteristics of CM but like FC, NC, and UC maintains a dictionary with the prefix property (if a string is in the dictionary, then so are all of its prefixes); it is discussed in Storer [1988].

It is easy to see that the FC, CM, and AP update heuristics satisfy the succinctness axiom and that the UC and NC update heuristics satisfy version 2 of the succinctness axiom.

For part A of the robustness axiom, observe that the learning constant is 1 for the UC and NC heuristics since the greedy match heuristic ensures that the current match together with the next character of the input stream cannot already be in the dictionary. The FC heuristic has a learning constant of ≤ 2 since the greedy match heuristic insures that the only way the last match together with the first character of the current match can already be in the dictionary is when this is the same as the current match, which can only happen once. Similarly, it can be argued that the learning constant for CM and AP is ≤ 2.

Part B of the robustness axiom is satisfied by all of the heuristics listed above for $\rho = 1$. In this paper, we will use $\rho > 1$ in our constructions for error resilience (so that a successive pair of matches may be seen several times before learning takes place). We shall not address how this modification affects the amount of compression achieved by the CM and AP heuristics since they are not provably optimal to start with (Lempel and Ziv [1990]), although they work well in practice, and restrict our attention to the FC, NC, and UC heuristics. The UC heuristic is exactly the algorithm shown optimal in Ziv and Lempel [1978]; a nice proof that it is optimal in the information-theoretic sense appears in Cover and Thomas [1991]. The essence of this proof is to show that any method that parses the input stream into distinct phrases (and sends a number of bits equal to $\varepsilon(n)$ of the current number of phrases to the decoder) must be optimal. Furthermore, as outlined in the appendix, this proof can be modified to show that a *constant-redundant parsing* (one where there is a constant that bounds how many times any phrase may appear in the parsing) is also optimal. Hence a "redundant" version of UC is optimal. In addition, we also show in the appendix that redundant versions of FC and NC are optimal.

2.3. Bounded-size dictionaries. Theoretical proofs of optimality such as presented in Lempel and Ziv [1976] and Ziv and Lempel [1977, 1978] simply assume that the dictionary grows indefinitely as the infinitely long input stream is processed. However, a practical strength of the compression methods outlined earlier is that relatively small bounds on the size of the dictionary (e.g., 2^{12} to 2^{16} entries) provide good performance in practice on virtually all types of data. As mentioned earlier, we take the same approach here and simply assume that once the dictionary fills, it remains fixed for the remainder of the data to be processed (so that the update heuristic is defined to add nothing once the dictionary is full). Although this strategy typically works well on individual files, in practice some method for changing the dictionary over time after it has filled is usually incorporated into the algorithm. (See Storer [1988] for discussion of various strategies.) For example, the UNIX compress command monitors compression and restarts the dictionary-growing process if compression falls off after the dictionary is full. Another simple strategy is a "swapping" arrangement with two dictionaries, where after the dictionary fills for the first time, continue using it to compress data but at the same time start forming new entries in a second dictionary; once the second dictionary is full, it can be used for compression and the first dictionary reset to be empty, and so on. Since restarting or changing of dictionaries occurs infrequently, very secure protocols (e.g., several hundred bits to encode a single bit) can be used to periodically restart the process or to send a swap bit. Such a protocol can also be used to agree that the dictionaries are now "frozen" and will not be modified further. We shall not address these issues further in this

paper.

2.4. A model for error-resilient communication. We use the term *communication channel* to refer to any medium or device over which data is transmitted and received. We assume that pointers sent from the encoder to the decoder over a communication channel are subject to the following errors:

- *add:* An extra pointer, chosen at random, is inserted into the communication stream.
- *delete:* A pointer is deleted from the communication stream.
- *change:* A new pointer, chosen at random from the space of all pointers, replaces a pointer.

We consider the following classes of error distributions:

- *uniform:* Errors occur randomly.
- *arbitrary:* Errors may be arbitrarily correlated.

Note that our results do not apply to the case where an individual bit is inserted or deleted but do allow individual bits to be changed because this is just a special case of a change error. (In fact we are only charging a cost of 1 no matter how many bits of a pointer are changed.)

The goal is to provide guarantees that there is *perfect protection* against error propagation due to a specified number of channel errors; that is, there is no further corruption of the data beyond what is directly due to the channel errors.

3. Protection against k errors. In this section, we present an encoding scheme which, for any fixed integer constant $k \geq 0$, guarantees that k or fewer errors on the channel will not cause *any* error propagation. We will not address how this protocol affects the amount of compression (although it appears reasonable in practice); rather, this protocol will be an important building block for a more general construction, presented in the next section, that does guarantee that compression is not affected asymptotically.

To simplify our presentation, we shall assume that the update heuristic adds at most one new match (e.g., UC, NC, FC, and CM have this property but AP does not); at the end of this section, we describe how this encoding scheme can be generalized to update heuristics that may add more than one new entry. Also, similar modifications as for the generic encoding and decoding algorithms are discussed for the UC and NC heuristics.

The two key ideas are as follows:

- Hashing is used to compute where a new entry is to be placed in the dictionary. For simplicity, throughout this paper, we assume that a truly random hash function is used; in practice, a 2-universal hash function (see Carter and Wegman [1979]) can be used. Hashing has the effect of eliminating the dependence between addresses that is normally present in dynamic dictionary communication so that if a given index is not used right away, it will have no effect on what indices are used in the future.
- Counts are maintained for all pointer pairs seen thus far and a pair is used by the match heuristic only if it "warms up" to be a clear winner over pairs that hash to the same address.

DEFINITION. *Suppose that counts (initially 0) are maintained in the encoder for all pointer pairs sent thus far (i.e., each time a pointer is sent/received, the count of the pair of pointers it represents is incremented) and in the decoder for all pointer pairs received thus far. For a sequence S of pointers sent by the encoder,* $\mathrm{LAG}(S)$ *is the maximum amount that any count in the decoder is incorrectly increased due to errors on the channel plus the maximum amount any count fails to be increased due*

to errors on the channel. For any integer $k \geq 0$, *the* warming value *for* k, $w(k)$, *is the smallest integer such that* $\mathrm{LAG}(S) \leq w(k)$ *over all sequences of pointers with at most* k *errors.*

LAG THEOREM. *For any* $k \geq 1, w(k) \leq 3k$.

Proof. Let

$S = U_0 C_1 U_1 \ldots C_m U_m$ be the sequence of pointers *sent* by the encoder and

$R = U_0 I_1 U_1 \ldots I_m U_m$ be the sequence of pointers *received* by the decoder

where

$U_i, 0 < i < m, |U_i| \geq 1$ are blocks of pointers,

$I_i, 1 \leq i \leq m, |I_i| \geq 0$ are blocks of pointers, and

$|\prod_{i=1}^{m} I_m| \leq k$ (the product denotes string concatenation).

That is, S and R are partitions (which are not necessarily unique) of the input and output streams into blocks of pointers defined as follows. The U's are blocks of pointers that went across the channel *unchanged* and the C's are blocks of pointers that were *correctly* sent to the communication channel but due to errors on the channel were received *incorrectly* by the decoder as the I blocks, where all pointers in a given I block are incorrect.

Now consider a particular block C_i that is received as I_i. Let

$x =$ the number of pointers that are changed,

$y =$ the number of pointers that are deleted, and

$z =$ the number of pointers that are added

and write

$$C_i = q_1 \cdots q_{x+y},$$
$$I_i = r_1 \cdots r_{x+z}.$$

Any way of choosing x, y, and z that corresponds to a way to convert C_i to I_i suffices for our construction. Let p be the last pointer of U_{i-1} and s be the first pointer of U_{i+1}. The only counts that might not be incremented (but should have been) are due to the loss of

$$pq_1, \quad q_1 q_2 \cdots q_{x+y-1} q_{x+y}, \quad q_{x+y} s,$$

which in the worst case can increase the lag by $x + y + 1$. The only counts that might be incorrectly incremented are due to the addition of

$$pr_1, \quad r_1 r_2 \cdots r_{x+z-1} r_{x+z}, \quad r_{x+z} s,$$

which in the worst case can increase the lag by $x + z + 1$. Note that if $i = 1$ and/or $i = m$, then p and/or s may not exist, and this can only lower the number of counts that might not be incremented or might be incorrectly incremented. Thus the total change in lag for the transformation of C_i to I_i is at most $2x + y + z + 2$. In fact, for the case where $y = z = 0$ (there are only change errors), this upper bound can be reduced by observing that if

$$pq_1 = q_1 q_2 = \cdots q_{x+y-1} q_{x+y} = q_{x+y} s,$$

then it follows that $p = q_1 \cdots q_{x+y} = s$, and so it cannot be that $pr_1 = r_1 r_2$ (or that $pr_1 = r_1 s$) or r_1 would not be an incorrect pointer, so the lag can be at most $2x + 1$. Hence we have

$$\mathrm{LAG}(C_i) \leq \left\{ \begin{array}{ll} 2x + 1 & \text{if } (y + z) = 0, \\ 2x + y + z + 2 & \text{if } (y + z) > 0 \end{array} \right\}.$$

Let $e = x + y + z$. For the case where $(y + z) = 0$, assuming that $k > 0$ (otherwise, $\mathrm{LAG}(C_i) = 0$), $2x + 1 \leq 2e + 1 \leq 3e$. For the case where $(y + z) > 0$:

$$\begin{aligned} 2x + y + z + 2 &= 2(e - (y + z)) + (y + z) + 2 \\ &= 2e - (y + z) + 2 \\ &\leq 2e + 1 \\ &\leq 3e. \end{aligned}$$

Hence the warming value for each block of e errors is $\leq 3e$, and the theorem follows.

ENCODING ALGORITHM WITH k-ERROR PROTOCOL.

1. Initialize the *local dictionary* D to have one entry for each character of the input alphabet and to have an empty hash table that is capable of storing pointer pairs; let h denote a hash function that maps pointer pairs to the range $0 \ldots (|D| - 1)$.
2. **repeat forever**
   A. {Get the current match string s and compute the current match pointer p:}
      Read a string s for which there exists pointer pair qr such that
      - s = the string corresponding to the pointer pair qr,
      - $\mathrm{count}(qr) > \mathrm{count}(uv) + w(k)$ for all uv such that $h(uv) = h(qr)$.

      if s does not exist
      then p = the index in D of the next input character
      else begin
      $p = h(qr)$
      $\mathrm{count}(qr) = \mathrm{count}(qr) + 1$
      end
      Transmit p using $\mathrm{BITS}(|D| - 1)$ bits.
   B. {Update D:}
      for each pair xy produced by the update method **do**
      if xy is not already in the dictionary **then** $\mathrm{count}(xy) = 0$

DECODING ALGORITHM WITH k-ERROR PROTOCOL.

1. Initialize the local dictionary D by performing step 1 of the encoding algorithm.
2. **repeat forever**
   A. {Get the current match pointer p and compute the current match string s:}
      Receive $\mathrm{BITS}(|D| - 1)$ bits for the current match pointer p.
      if p represents a single character
      then s = the single character corresponding to p
      else if there is a pointer pair qr such that
      $h(qr) = p$ and $\mathrm{count}(qr)$ is largest among all pairs that hash to p
      then begin
      s = the string corresponding to qr
      $\mathrm{count}(qr) = \mathrm{count}(qr) + 1$
      end
      else s = the empty string
      Output s.
   B. {Update D:}
      Perform step 2B of the encoding algorithm.

Given the lag theorem, the generic encoding and decoding algorithms can be modified, as shown above, to employ the *k-error protocol* to insure perfect protection against any k errors; that is, no bytes are corrupted beyond those corrupted by channel errors.

For correctness of the protocol, observe that for each pointer p in a sequence S of pointers sent by the encoder, the pointer pair qr that p represents is a clear "winner" among all pointer pairs that hash to p; that is, the count of qr is greater by at least $\mathrm{LAG}(S)$ than the count of any other pair uv that hashes to p. Hence when the decoder receives an uncorrupted pointer p, the pointer pair qr with the largest count that hashes to p must be the correct pair for p because k errors cannot cause some other pair uv that hashes to p to have a count equal or greater than that of qr.

As mentioned at the beginning of this section, the encoding and decoding algorithms with the k-error protocol can be adapted to the UC and NC heuristics, and to heuristics such as AP that may add more than one entry. For the UC and NC heuristics, the same modifications as for the generic encoding and decoding algorithms can be used. For the AP heuristic, or any heuristic that may construct more than one match from the last match and the current match, we can hash on triples consisting of the two pointers in question together with an integer that indicates which of the strings corresponding constructed from this pointer pair is being referenced; this integer can be provided by the match heuristic. The lag theorem can still be used to verify the protocol because it is still the case that at most one count is incremented when a pointer is sent by the encoder or received by the decoder.

We leave as an open problem the class of sources for which the k-error protocol produces a constant-redundant parsing for update heuristics that satisfy the succinctness and robustness axioms. All that has been shown here is that k errors will not propagate when using the k-error protocol. Because hash conflicts could in theory cause the counts of two entries that hash to the same location to "race" (so that neither count sufficiently exceeded the other), it is not clear what effect, if any, the protocol has asymptotically on the amount of compression obtained with a given update heuristic (as compared with the same heuristic without the protocol). We conjecture that in practice, performance will not be significantly affected for small values of k and reasonably size dictionaries (e.g., 2^{16} or more entries). In the next section, we make use of the k-error protocol in a way that avoids hash conflicts and insures constant redundancy.

4. High-probability protection against an error rate. In this section, we employ probabilistic analysis to examine more carefully the proof of the lag theorem of the last section. The idea is to show that the k-error protocol actually gives, with high probability, strong protection against a fixed error rate during the period when the dictionary is changing and is vulnerable to error propagation. In addition, by encoding pointers to avoid hash conflicts with high probability, we can guarantee no asymptotic loss of compression.

We employ the Chernoff bound (see Hagerup and Rub [1989]) on the number of heads X in a sequence of independent coin tossings with expected value μ ($e = 2.7182\ldots$ denotes the natural logarithm base defined by $\lim_{n\to\infty}(\frac{n}{n-1})^n$):

$$\text{For } x \geq 0, \quad \mathrm{Prob}(X \geq (1+h)\mu) < \left(\frac{e^h}{(1+h)^{(1+h)}}\right)^{\mu}.$$

To obtain the form of the bound that we shall use here, first we observe that the right-hand side is

$$= \frac{1}{e^{\mu}}\left(\frac{e}{1+h}\right)^{(1+h)u} = \frac{1}{e^{\mu}}\left(\frac{(1+h)\mu}{e\mu}\right)^{-(1+h)\mu}$$

and hence, since $e > 1$, it follows that

$$\text{For } z > e\mu, \quad \text{Prob}(X \geq z) < \frac{1}{e^{\mu}} \left(\frac{z}{e\mu}\right)^{-z} < \left(\frac{z}{e\mu}\right)^{-z}.$$

DAMPING THEOREM. *If a sufficiently large dictionary is employed with a method satisfying the succinctness and robustness axioms together with the k-error protocol of the last section (where $w(k)$ denotes the warming value for k and α denotes the learning constant) on a channel with a uniform independent error probability of $1/r$ (on the average, one error for every r pointers), then the probability that the system looses stability (i.e., the probability that any errors propagate) is*

$$< \frac{1}{\left(\frac{r}{4e\alpha}\right)^{\left(\frac{w(k)}{2}+1\right)}}.$$

In addition, with probability greater than $1 - 1/2^{|D|(1-\varepsilon(|D|))}$, the amount of compression achieved is within a factor of $(1+\varepsilon(|D|))$ of what would be achieved with the same method used on a perfect channel without the protocol, where $\varepsilon(|D|) \to 0$ as $|D| \to \infty$.

Proof. To simplify notation, we assume throughout this proof that all logarithms are base 2.

Although the k-error protocol may work in the presence of hash conflicts, it is difficult to estimate the effect of the hash conflicts on the performance of the protocol for the application in question; for example, for data compression, it is not clear how the compression achieved is affected in the worst case. We avoid this problem by employing a more complicated hashing scheme that has no conflicts with extremely high probability. Although this scheme may increase the number of bits in some pointers, it will have no asymptotic effect on the total number of bits sent.

To store n items in the hash table, we use a table of size $n \log(n)^2$. Since at any time the table contains at most n elements, each time an element is inserted into the table, the probability that it goes to a bucket that already contains an entry is at most $n/(n \log(n)^2) = 1/(\log(n)^2)$, and thus the expected number of hash conflicts after inserting all n elements is bounded above by $n/\log(n)^2$. Hence from the Chernoff bound, with $\mu = n/\log(n)^2$ and $z = n/\log(n)$, it follows that the number of hash conflicts is greater than $n/\log(n)$ with probability

$$\begin{aligned} &< \left(\frac{n/\log(n)}{en/(\log(n)^2)}\right)^{-n/(\log(n)^2)} \\ &= \left(\frac{\log(n)}{e}\right)^{-n/(\log(n)^2)} \\ &< \frac{1}{2^{n/(\log(n)^2)}}. \end{aligned}$$

Thus it follows that

$$\text{Prob}(\text{number of hash conflicts } < n/\log(n)) > 1 - \frac{1}{2^{n/(\log(n)^2)}}.$$

The $n/\log(n)$ conflicting entries can all be hashed again into a table of size $n\log(n)$, the remaining $n/(\log(n)^2)$ conflicting entries can all be hashed again into a table of size n, and so on. In general, each pointer is now being represented by a sequence of indices, each consisting of a number of bits equal to log of the size of the corresponding hash table followed by a bit that is 0 if the index is the last in the pointer or 1 if there is another to follow. By summing the bits sent for a sequence of n pointers, we see that all n pointers use $\log(n\log(n)^2)+1$ bits for the first index, at most $n/\log(n)$ pointers have an additional $\log(n\log(n))+1$ bits for the second index, and so on. The first term is bounded by $(1+\varepsilon(n))n\log(n)$, where $\varepsilon(n)\to 0$ as $n\to\infty$, the second term is $O(n)$, and the remaining terms go down geometrically by a factor of $\log(n)$. Hence the total number of bits sent is asymptotically arbitrarily close to the $n\log(n)$ bits that are sent in any case.

Given that we can assume that there are no hash conflicts for the encoder, the k-error protocol takes care of conflicts at the decoding end. That is, if a pointer is received whose bits are the prefix of the bits of two or more pointers, the one with the largest count wins.

We now bound the average value of the following two quantities:

X_{ij} = the number of times that the count of pair i,j is incorrectly increased by the decoder due to errors on the channel, before the dictionary of the encoder is full;

Y_{ij} = the number of times that the count of a pair i,j fails to be increased when it would otherwise have been increased if there had been no errors on the channel, before the dictionary of the encoder becomes full.

Let

$E(D)$ = the expected number of pointers transmitted by the encoder before its dictionary is full.

From the proof of the lag theorem, we know that each error can cause at most two counts to incorrectly increase, and hence for each of the $E(D)/r$ pointers that are corrupted, at most two counts, which are equally likely to be any of at least $|D_{\max}|$ pairs, can be incorrectly increased. So the expected value of X_{ij}, which we denote by μX_{ij}, is bounded by

$$\mu X_{ij} \le \frac{2E(D)}{r|D_{\max}|}.$$

By the robustness axiom, $E(D)\le \alpha w(k)|D_{\max}|$, and hence

$$\mu X_{ij} \le \frac{2\alpha w(k)}{r}.$$

Since each error can cause at most two counts to incorrectly fail to increase, similar to X_{ij}, we can compute

$$\mu Y_{ij} \le \frac{2\alpha w(k)}{r}.$$

Hence the probability that X_{ij} or Y_{ij} is $\geq (\frac{w(k)}{2}+1)$ is

$$> \left(\frac{\frac{w(k)}{2}+1}{\frac{2e\alpha w(k)}{r}}\right)^{-\left(\frac{w(k)}{2}+1\right)}$$

$$= \frac{1}{\left(\frac{r(w(k)+2)}{4e\alpha w(k)}\right)^{\frac{w(k)}{2}+1}}$$

$$< \frac{1}{\left(\frac{r}{4e\alpha}\right)^{\frac{w(k)}{2}+1}}.$$

From the above bound, the theorem follows since a LAG of more than $w(k)$ cannot occur if no values of X_{ij} or Y_{ij} are more than $w(k)/2$.

Let us now consider the amount of compression achieved. We have already verified that each of the $|D|$ pointers is encoded with only a factor of $1+\varepsilon(|D|)$ more bits, where $\varepsilon(|D|) \to 0$ as $|D| \to \infty$. Hence if the method in question is optimal in the information-theoretic sense, then by the robustness axioms, as $|D| \to \infty$, the same method with the k-error protocol must also be optimal in the information theoretic sense (since it is $w(k)$-redundant).

COROLLARY. *For $\alpha \leq 2$ (which is the case for all data-compression methods that we have considered) and $r \geq 10^{12}$ (a reasonable assumption in practice for a clean channel with a low-overhead error-correction mechanism), choosing $w(k) \geq (2.2509k-2)$ yields a probability that is*

$$< \frac{1}{r^k}.$$

For example, if $k=5$, then by using $w(k)=10$, an error rate of $1/10^{12}$ is effectively "damped" to $1/10^{60}$.

Proof. If we write $w(k) = 2xk-2$, we can solve for the minimum value of x by simplifying the expression above as follows:

$$\leq \frac{1}{r^k\left(\frac{r^{\frac{w(k)}{2}+1-k}}{(8e)^{\frac{w(k)}{2}+1}}\right)}$$

$$= \frac{1}{r^k\left(\frac{r^{x-1}}{(8e)^x}\right)^k}.$$

The expression $\frac{r^{x-1}}{(8e)^x}$, which is monotonically increasing in x, becomes ≥ 1 when $(r/8e)^x \geq r$, which is true when $x \geq \frac{\log(r)}{\log(r)-\log(8e)}$. Assuming $r \geq 10^{12}$, $w(k) \geq 2.2509k-2$ suffices.

5. Practical considerations. The previous section encoded pointers to make the probability of *any* hash conflicts arbitrarily small. Although the protocol may work even in the presence of hash conflicts, this eliminated the need to address the issue of what affect hash conflicts have on performance of the system (e.g., the amount

of compression achieved). We leave it as an open problem whether, in the case of data compression for stationary ergodic sources, performance is affected when hash conflicts are allowed. Here we mention another strategy that can be viewed as a compromise between allowing hash conflicts and rehashing to avoid them. With a small number of extra bits per pointer, we can make the number of hash conflicts small and then simply not use these entries of the dictionary (thus "wasting" a small fraction of the dictionary).

A small constant $c \geq 1$ extra bits are added to each pointer so that the $|D_{\max}|$ dictionary entries are hashed into a space of $|D_{\max}|2^c$ indices. Consider a particular index i that is used for the first time in a sequence of n pairs that are hashed into the dictionary. The probability that all other pairs do *not* hash to index i is

$$
\begin{aligned}
&\geq \left(1 - \frac{1}{n2^c}\right)^{n-1} \\
&> \left(1 - \frac{1}{n2^c}\right)^{n} \\
&= \left(\left(1 - \frac{1}{n2^c}\right)^{n2^c}\right)^{2^{-c}} \\
&\geq \left(\frac{1}{e(n)}\right)^{2^{-c}}, \quad \text{where } e(n) = \left(\frac{n}{n-1}\right)^{n}, \\
&= \frac{\left(\frac{1}{e}\right)^{2^{-c}}}{\left(\frac{e(n)}{e}\right)^{2^{-c}}} \\
&> \frac{1 - (2^{-c})}{\left(\frac{e(n)}{e}\right)^{2^{-c}}}, \quad e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \text{and for all } 0 < x < 1, \quad \frac{1}{e^x} > 1 - x.
\end{aligned}
$$

Hence the probability that two pairs hash to index i is

$$
\begin{aligned}
&\leq 1 - \frac{1 - (2^{-c})}{\left(\frac{e(n)}{e}\right)^{(2^{-c})}} \\
&< 2^{-c} + \left(1 - \left(\frac{e}{e(n)}\right)^{(2^{-c})}\right) \quad \text{since for all } n > 1,\ e(n) > e \\
&= 2^{-c} + \varepsilon(n) \quad \text{since } e = \lim_{n \to \infty} e(n),
\end{aligned}
$$

where $\varepsilon(n)$ goes to 0 as n goes to ∞. Thus the expected number of hash conflicts is less than

$$n(2^{-c} + \varepsilon(n)),$$

and by the Chernoff bound it follows that the probability that there are more than εn hash conflicts is

$$< \left(\frac{\varepsilon}{e(2^{-c} + \varepsilon(n))}\right)^{-\varepsilon n}.$$

For any constant $d > 1$, this expression can be made less than $d^{-\varepsilon n}$ if c is made sufficiently large (by choosing n sufficiently large).

Given that the probability of more than εn hash conflicts can be made arbitrarily low, a simple approach is the conservative strategy of "throwing out" all indices that correspond to a hash conflict (even though many or all of these entries may not cause any problems for the decoder). This construction adds c extra bits to each pointer, which can be made to have an insignificant effect on compression by using a very large dictionary (and hence a very large pointer size) but will most likely be significant for dictionary sizes that are used in practice. However, this effect may not be unacceptable. For example, if we take $|D_{\max}| = 2^{16}$ (a common and practical choice for data compression) and $\varepsilon = 0.1$ (again, a reasonable value in practice), then we must choose $c \geq 5$; taking $c \geq 5$ and computing the error term $\varepsilon(n)$, we see that the probability that there are more than $\varepsilon(n)$ hash conflicts is $< 2^{-1,500}$. However, the cost for this security against many hash conflicts is an extra five bits for every 16-bit pointer. If we consider a typical example for lossless compression using a dictionary of size 2^{16}, we might expect the compressed data to be 30 percent of the size of the original data, but now with the extra five bits per pointer, the compressed size will be $\frac{21}{16}30 \approx 39$ percent of the original size. Although this cost might be reasonable in practice, it would be nice to avoid it. We conjecture that this cost can be reduced by a tighter analysis that does not throw out all indices with hash conflicts but rather throws out only those that delay the dictionary-filling process due to "racing" and "thrashing" of counts. We leave such analysis (as well as the effect of hash conflicts on application-dependent performance issues such as compression ratio) as a subject for future research.

Appendix. Constant-redundant parsings are optimal. The UC heuristic is exactly the algorithm shown to be optimal in Ziv and Lempel [1978]; a nice proof appears in Cover and Thomas [1991]. The essence of this proof is to show that *any* method that parses the input stream into distinct phrases (and sends a number of bits equal to $\log_2$ of the current number of phrases to the decoder) must be optimal. This appendix notes how this notion can be generalized and still maintain optimality.

DEFINITION. *A* τ-redundant parsing *of the input stream is one with at most* τ *copies of any given phrase.*

LEMMA. τ*-redundant parsings give optimal data compression for the UC model.*

Proof. We refer to the presentation in section 12.10 of Cover and Thomas [1991] and describe how slight modifications can be made to the formulas. Note that here $c(n)$ denotes the total number of phrases (with repetition), whereas in the original presentation phrases are not repeated. Lemmas 12.10.1 and 12.10.2 do not change if the parsing is τ-redundant. Lemma 12.10.3 (Ziv's inequality) gets a factor of τ inside the log on the right side. (The proof is essentially unchanged except that the 1 in the log on the right side in equation (12.286) becomes τ.) Theorem 12.10.1 (the main theorem) still holds for a τ-redundant parsing; the proof is essentially the same, with the following small changes: A factor of τ goes in the denominator inside the log on the right side of equation (12.288), which is equivalent to adding the term $\tau \log(\rho)$ to the right side of equation (12.288). This term of $c \log(\tau)$ is added to the right side of equation (12.292) and subtracted from the right side of equation (12.293). Finally, the error term $\varepsilon_k(n)$ in equation (12.300) is changed to have $(c/n)\log(\tau)$ added to it; since c is $O(n/\log(n))$ by Lemma 12.10.1, the error term still goes to 0 as n goes to ∞.

COROLLARY. *The lemma holds for the FC and NC heuristics as well.*
Proof. These heuristics can use a phrase at most $\tau|\Sigma|$ times.

REFERENCES

J. L. CARTER AND M. N. WEGMAN [1979], *Universal classes of hash functions*, J. Comput. System Sci., 18, pp. 143–154.

T. M. COVER AND J. A. THOMAS [1991], *Elements of Information Theory*, John Wiley, New York.

R. G. GALLAGER [1968], *Information Theory and Reliable Communication*, John Wiley, New York.

T. HAGERUP AND C. RUB [1989], *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33, pp. 305–308.

A. LEMPEL AND J. ZIV [1976], *On the complexity of finite sequences*, IEEE Trans. Inform. Theory, 22, pp. 75–81.

A. LEMPEL AND J. ZIV [1990], private communication.

V. S. MILLER AND M. N. WEGMAN [1985], *Variations on a theme by Lempel and Ziv*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, pp. 131–140.

J. REIF AND J. A. STORER [1990], *A parallel architecture for high speed data compression*, in Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, IEEE Press, Los Alamitos, CA, pp. 238–243.

J. REIF AND J. A. STORER [1992], *A parallel architecture for high speed data compression*, J. Parallel Distrib. Comput., 13, pp. 222–227.

D. M. ROYALS, T. MARKAS, N. KANAPOULOS, J. H. REIF, AND J. A. STORER [1993], *On the design and implementation of a lossless data compression and decompression chip*, IEEE J. Solid-State Circuits, 28, pp. 948–953.

J. B. SEERY AND J. ZIV [1977], *A universal data compression algorithm: Description and preliminary results*, Technical Memorandum 77-1212-6, Bell Laboratories, Murray Hill, NJ.

J. B. SEERY AND J. ZIV [1978], *Further results on universal data compression*, Technical Memorandum 78-1212-8, Bell Laboratories, Murray Hill, NJ.

J. A. STORER [1988], *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD.

J. A. STORER AND T. G. SZYMANSKI [1978], *The macro model for data compression*, in Proc. 10th Annual ACM Symposium on the Theory of Computing, ACM, New York, pp. 30–39.

T. A. WELCH [1984], *A technique for high-performance data compression*, IEEE Comput., 17, pp. 8–19.

J. ZIV AND A. LEMPEL [1977], *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23, pp. 337–343.

J. ZIV AND A. LEMPEL [1978], *Compression of individual sequences via variable-rate coding*, IEEE Trans. Inform. Theory, 24, pp. 530–536.