Joint Source–Channel LZ'77 Coding[*]

Stefano Lonardi
Dept. Computer Science & Engineering
University of California
Riverside, CA 92521
stelo@cs.ucr.edu

Wojciech Szpankowski
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
spa@cs.purdue.edu

Abstract

Limited memory and bounded communication resources require powerful data compression techniques, but at the same time noisy tetherless channels and/or corrupted file systems need error correction capabilities. Joint source-channel coding has emerged as a viable solution to this problem. We present here the first practical joint source-channel coding algorithm capable of correcting errors in the popular Lempel-Ziv'77 scheme without practically losing any compression power. This is possible since the LZ'77 encoder does not completely remove all redundancy. The inherent additional redundancy left by LZ'77 encoder is used succinctly by a channel coder (e.g., Reed-Solomon coder) to protect against a limited number of errors. In addition to this, the scheme proposed here is perfectly backward-compatible, that is, a file compressed with our error-resilient LZ'77 can be still decompressed by a common LZ'77 decoder. In this preliminary report, we present our algorithm, collect some experimental data supporting our claims, and provide some theoretical justifications.

1 Introduction

Error-resilient adaptive lossless data compression is a particularly challenging problem because of two opposing "forces". *Source coding* tries to decorrelate as much as possible the input sequence (i.e., by removing redundant information), while *channel coding* introduces additional correlation (i.e., by adding redundant information) in order to protect against errors. The devastating effect of errors in adaptive data compression is a long-standing open problem posed in '70s by Lempel and Ziv. In fact, the non-resilience of adaptive data compression has been a practical drawback of its use in many applications. Joint source-channel coding has emerged as a viable solution to this problem.

In this paper we deal with the best known adaptive data compression scheme, namely that of Lempel–Ziv'77 [16]. This popular compression scheme works on-line and replaces the longest prefix of non-yet-compressed file with a *pointer* and *length* of its copy in the already compressed file. The lack of error-resilience of LZ'77 is a well-recognized problem. In a recent posting on the `comp.compression` newsgroup, we read [6]:

[*]This work was partially supported by NSF Grant CCR-0208709 and by Bourns College of Engineering, University of California, Riverside

... I'm a casualty of corrupt tar'd gzipped files on Solaris 8. (`gzip` 1.3) ... Is there a reason why there are no compression utilities that allow controlled amounts of redundancy for error correction? ... How much overhead would be needed to correct these?

Indeed, how much overhead is needed in LZ'77 to correct errors? We shall argue that practically there is no need for additional overhead in order to correct errors in LZ'77. This seemingly impossible goal is achieved in practice due to the fact that the LZ'77 encoder is unable to decorrelate completely the input sequence. Some implicit redundancy is still present in the compressed stream and can be exploited by the encoder. The additional redundancy derives from the encoding of phrases for which one has a choice among more than one possible pointer. We observed experimentally that in a significant proportion of phrases, there is more than one copy of the longest prefix in the compressed file (see, e.g., Table 1 that shows number of redundant bytes that can be used for error correction). We also prove that on average there are $O(1)$ copies (Theorem 1). In practice, if there are q copies of the longest prefix, we recover $\lfloor \log_2 q \rfloor$ redundant bits by choosing one of the q pointers (see Figure 1). In order to avoid the loss inherent to the ceiling operator, one could also "batch" the bits for several phrases together.

Once the redundant bits of LZ'77 have been identified, one can devise the method to exploit them for channel coding. We choose $RS(255, 255-2e)$ Reed-Solomon codes, where 255 is the size of the buffer containing the data and the parity bits, and e is the maximum number of errors that the code can correct. The actual payload is $255-2e$, because Reed-Solomon codes need $2e$ parity bits. We should point out that if e is large then we may not have *always* enough redundant bits to embed the parity bits. For the purposes of this paper, we fixed a value of e small enough such that there are always enough bits. We are currently working on a scheme in which e is changed adaptively with the availability of redundancy bits in the input stream.

To the best of our knowledge, the scheme described here is the first joint source-channel LZ'77 algorithm. In [12], Storer and Reif address the issue of *error propagation* but not error recovery (cf. see [8] for an analysis of the Storer and Reif algorithm). There are, however, joint source-channel coding algorithms for arithmetic coding and other variable length codes (see, e.g., [11]).

This paper is organized as follows. In the next section we describe how to obtain redundant bits in the LZ'77 scheme. We prove in Section 3 that there are $O(1)$ copies of the longest prefix in LZ'77. Finally, in Section 4 we explain how to embed Reed-Solomon codes in LZ'77 and we report experimental results in Section 5.

2 Redundant Information in LZ'77

Let T be a text of length n over a finite alphabet $\mathcal{A}$. We write $T_{[i]}$, $1 \le i \le n$ to indicate the i-th symbol in T. We use $T_{[i,j]}$ shorthand for the substring $T_{[i]}T_{[i+1]}\dots T_{[j]}$ where $1 \le i \le j \le n$, with the convention that $T_{[i,i]} = T_{[i]}$. Substrings in the form $T_{[1,j]}$ corresponds to the prefixes of T, and substrings in the form $T_{[i,n]}$ to the suffixes of T.

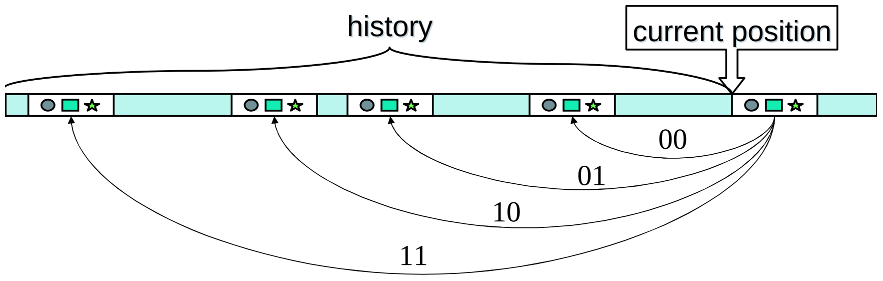


Figure 1: The multiplicity of the next phrase is four ($q = 4$). Choosing one of the four possible pointers recovers two redundant bits.

The LZ'77 algorithm [16] processes the data *on-line* as it is read, i.e., it parses the file sequentially *left to right* and looks into the sequence of past symbols to find a match with the longest prefix of the string starting at the current position. The longest prefix is substituted with a *pointer*, which is a triple composed of *(position, length, symbol)*. Several variations on LZ'77 have been proposed (see, e.g., [2] and references therein), but the basic principle remains the same.

Let us suppose that the first $i-1$ symbols of the string T have been already parsed in $h-1$ phrases, i.e., $T_{[1,i-1]} = y_1 y_2 \ldots y_{h-1}$. To identify the h-th phrase, LZ'77 looks for the *longest* prefix of $T_{[i,n]}$ that matches a substring of $T_{[1,i-1]}$. If $T_{[j,j+l-1]}$, $j < i$ is the substring that matches the longest prefix, then the next phrase is $y_h = T_{[i,i+l-1]}$. The algorithm issues the pointer $(j, l, T_{[i+l]})$ and updates the current position i to $i + l + 1$. The reason we need $T_{[i+l]}$ is to be able to advance when $l = 0$, which is common in the very beginning of this process.

In order to recover additional bits to be used for channel coding, we slightly modify the LZ'77 encoding. The resulting algorithm, called LZS'77, allows one to embed some bits of another text M over the alphabet $\{0, 1\}$. We define a position i corresponding to the beginning of a phrase to have *multiplicity* q if there exists exactly q matches for the longest prefix that starts at position i in T. The positions with multiplicity $q > 1$ are the places where we can embed some of the bits of M. Specifically, the next $\lfloor \log_2(q) \rfloor$ bits will drive the selection of one particular pointer out of the q choices (see Figure 1). These additional bits can be used for various purposes such as authentication [1] or error correction as described next.

Suppose again that the initial portion of T, say $T_{[1,i-1]}$, has been already parsed. Let $\{(p_0, l, T_{[i+l]}),\ (p_1, l, T_{[i+l]}),\ \ldots,\ (p_{q-1}, l, T_{[i+l]})\}$, $q \geq 1$, be the set of feasible pointers for the longest prefix of $T_{[i,n]}$, where $l > 1$, and $1 \leq p_l \leq i$ for all $0 \leq l \leq q-1$. If $q = 1$ we skip to the next phrase, and no extra bits are embedded. When $q > 1$, we use the next $d = \lfloor \log_2(q) \rfloor$ bits of M to choose one of the q pointers. If we assume that the first $r-1$ bits of M have been already embedded in previous phrases, we select the pointer whose name matches the next d bits of M, that is $(p_{M_{[r,r+d]}}, l, T_{[i+l]})$. We move the current position to $i + l + 1$, and we increment r by d. The complete algorithm is summarized in Figure 2.

```
LZS'77_ENCODER $(T, M)$
1  let $i, r, n, m, P \leftarrow 0, 0, |T|, |M|, []$
2  while $i < n$ do
3      let $T_{[i,i+l-1]} \leftarrow$ the longest prefix of $T_{[i,t]}$ that matches a substring in $T_{[1,i-1]}$
4      let $R \leftarrow \{(p_0, l, T_{[i+l]}), \ldots, (p_{q-1}, l, T_{[i+l]})\}$ be the set of feasible pointers for $T_{[i,i+l-1]}$
5      if $q > 1$ then
6          let $d \leftarrow \lfloor \log_2(q) \rfloor$
7          append $(p_{M_{[r,r+d]}}, l, T_{[i+l]})$ to $P$
8          let $r \leftarrow r + d$
9      else
10         append $(p_{q-1}, l, T_{[i+l]})$ to $P$
11     let $i \leftarrow i + l + 1$
12 return $P$
```

```
LZS'77_DECODER $(P)$
1  let $D, M \leftarrow$ empty string, empty string
2  for each $(p, l, c) \in P$ do
3      let $R \leftarrow \{p_0, \ldots, p_{q-1}\}$ be the set of occurrences of $D_{[p,p+l-1]}$
4      let $i$ the index such that $p_i = p$
5      append $\lfloor \log_2(q) \rfloor$ bits of $i$ to $M$
6      append $D_{[p,p+l-1]}c$ to $D$
7  return $(D, M)$
```

Figure 2: Recovering redundant bits M in LZ'77. T is the text, M represents the redundant bits, P is the compressed stream of pointers, D is the decompressed text.

We want to stress that these changes do not affect the internal structure of LZ'77 encoding, other than a possible re-shuffling of the pointers. A file compressed with LZS'77 can still be decompressed by a standard LZ'77 algorithm. The fact that LZS'77 is "backward-compatible" makes it possible to deploy it gradually over the existing LZ'77, without disrupting service.

3 Average Case Analysis

Given the limited space allowed for the paper, we only present preliminary analytical results regarding the size of redundant bits. More precisely, we formulate and sketch a proof of a result that will allow us to estimate the average number of copies of the longest prefix (i.e., its multiplicity) when the text is generated by a Markov source.

Let $T_{[1,n]}$ be the first n symbols generated by the source and let L_n be the random variable associated with the length of the longest prefix (phrase) of $T_{[n+1,\infty]}$ which have an occurrence in $T_{[1,n]}$. The random variable L_n describes the length of the phrases of LZ'77. It is known (see [4, 13, 15]) that $L_n \sim \frac{\log_2 n}{H} + O(1)$ in probability and on average, where H is the entropy of the source that generates T. In the following we write L instead of L_n to simplify the notation. Finally, we define the number of longest prefix occurrences as W_n, that is,

$$W_n = \sum_{i=1}^{n-L} \mathbf{1}(T_{[i,i+l-1]} = T_{[n+1,n+L]}).$$

Theorem 1 *Let $T_{[1,n]}$ be generated by a Markov source. Then*

$$\mathbf{E}[W_n] = O(1), \tag{1}$$

that is, on average there are constant number of pointers for n large.

Proof. We follow Wyner [15]. Let for any sequence z

$$Z_n(z) = \min\{k \geq 1 : \ P(T_{[1,k]} = z_{[1,k]}) \leq 1/n\}.$$

Observe that for $k = Z_n(z)$

$$\mathbf{E}[W(z)] = (n-k)/n \sim 1,$$

hence when the prefix $z_{[1,k]}$ of $T_{[n,\infty]}$ is of length $k = Z_n(z)$, there is about one occurrence of this prefix. Wyner [15] proved that the longest prefix is very well approximated by $Z_n(z)$. More precisely, let

$$L_n = Z_n(T) + C.$$

Then there are constants $\alpha, \beta < 1$ such that

$$\begin{aligned} P(C \geq j) &\leq \alpha^j & j > 0, \\ P(C \leq j) &\leq \beta^{-j} & j < 0. \end{aligned}$$

Furthermore, one finds that for $\ell = Z_n(T) + j$ we have $P(T_{[n,n+\ell]}) \leq \frac{1}{n}\alpha^j$ for $j > 0$ and $P(T_{[n,n+\ell]}) \leq \frac{1}{n}\beta^{-j}$ for $j < 0$. With this in mind, we have

$$\mathbf{E}[W_n(T_{[n,n+Z(T)+C]})] \leq \sum_{j \leq 0} \beta^{-j} + \sum_{j \geq 0} \alpha^j = O(1).$$

This completes the proof. ∎

The experimental results in Section 5 seem to confirm that the multiplicity of the pointers converges asymptotically to a constant. Actually, we conjecture that W_n is Poisson distributed with mean $\lambda = O(1)$. In passing, we should add that we can obtained more redundant bits, if necessary, by considering not the longest prefix, but say the k-th longest (where $k = o(\log \log n)$). We will not, however, explore this line of study in this paper.

4 Error-Resilience in LZ'77

We now describe how to use the extra redundant bits to achieve error-resilience. Since we are protecting the stream of pointers, i.e., a sequence of bytes, we choose Reed-Solomon (RS) codes [7]. Reed-Solomon codes are block-based error correcting codes widely used in digital communications and storage.

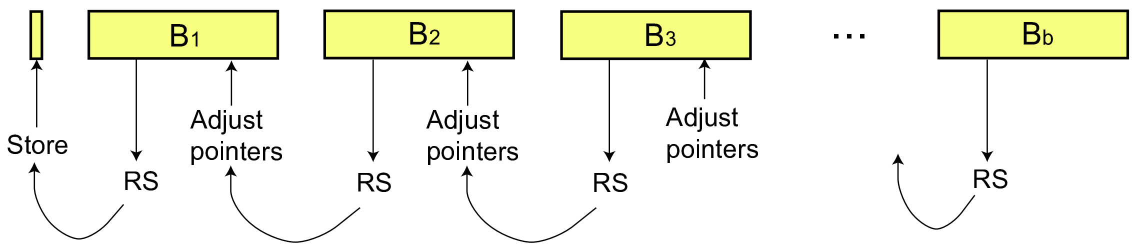


Figure 3: The right-to-left sequence of operations on the blocks for the encoder

Reed-Solomon codes belongs to the family of BCH codes (see, e.g., [10]). A Reed-Solomon code is specified as $\mathrm{RS}(a, b)$, where a is the size of the block and b is the size of the payload. The datum is a symbol drawn from an alphabet of cardinality 2^s. The encoder collects b symbols and adds $a-b$ parity symbols to make a block of length a. A Reed-Solomon decoder can correct up to e errors in a block, where $e = (a-b)/2$. One symbol error occurs if one or more of the bits of the symbol (up to s) is wrong.

Given a symbol size s, the maximum block length a for a Reed-Solomon code is $a = 2^s - 1$. For example, the maximum length of a code with 8-bit symbols ($s = 8$) is 255 bytes. The family of Reed-Solomon codes for $s = 8$ is therefore $\mathrm{RS}(255, 255 - 2e)$. Each block contains 255 bytes, of which $255 - 2e$ are data and $2e$ are parity. Errors up to e bytes anywhere in the block can be automatically detected and corrected.

We can use the extra redundancy bits of LZS'77 to embed $2e$ extra bytes, as described in the following. The encoder, called LZRS'77, first compresses T using standard LZ'77. The data is broken into blocks of size $255 - 2e$. Then, blocks are processed in reverse order, beginning with the very last. When processing block i, the encoder computes first the Reed-Solomon parity bits for the block $i + 1$ and then it embeds the extra bits in the pointers of block i using the method described in Section 2. The order in which the operations are performed in the encoder is illustrated in Figure 3. The parity bits of the first block are not embedded, but saved at the beginning of the compressed file.

The LZRS'77 decoder receives a sequence of pointers, preceded by the parity bits of the first block. It first breaks the input stream in blocks of size $255-2e$. Then, it uses the parity bits at the beginning of the compressed stream to check (and possibly correct) the first block. Once block B_1 is correct, it decompresses it using LZS'77. This not only reconstructs the initial portion of the original text, but it also recovers the bits stored in those particular choices for the pointers. These extra bits are collected, and they become the parity bits for the second block. The decoder can therefore correct possible errors in B_2. Block B_2 is then decompressed, and the parity bits for B_3 are recovered. This process continues until all blocks have been decompressed. A high-level description of the encoder and the decoder is shown in Figure 4.

The reason the encoder needs to process the blocks in reverse order should now be apparent. The encoder cannot compute the RS parity bits before the pointers are finalized. We embed the RS bits for the current block in the *previous* block, because the decoder needs to know the parity bits of a block *before* it attempts to decompress it. This has the unfortunate effect of making the encoder off-line, since it requires the encoder to keep the

```
LZRS'77_ENCODER ($T, e$)
1  let $b, j, n \leftarrow 1, 1, |T|$
2  while $j < n$ do
3      append LZ'77_COMPRESS($T_j$) to $B_b$   /* compress the next phrase */
4      if $|B_b| = 255 - 2e$ then
5          let $b \leftarrow b + 1$
6  for $i \leftarrow b, \ldots, 2$ do
7      let $RS_i \leftarrow$ REED_SOLOMON_ENCODER($B_i, e$)
8      embed in the block $B_{i-1}$ the bits $RS_i$ using LZS'77
9  let $RS_1 \leftarrow$ REED_SOLOMON_ENCODER($B_1, e$)
10 return $RS_1, B_1, B_2, \ldots, B_b$

LZRS'77_DECODER ($RS_1, B_1, B_2, \ldots, B_b, e$)
1  $D \leftarrow$ empty string
2  if REED_SOLOMON_DECODER($B_1 + RS_1, e$) = errors then correct $B_1$
3  append LZ'77_DECOMPRESS($B_i$) to $D$
4  recover $RS_2$ from the pointers used in $B_1$ using LZS'77
5  for $i \leftarrow 2, \ldots, b$ do
6      if REED_SOLOMON_DECODER($B_i + RS_i, e$) = errors then correct $B_i$
7      append LZ'77_DECOMPRESS($B_i$) to $D$   /* decompress all pointers in $B_i$ */
8      recover $RS_{i+1}$ from the pointers used in $B_i$ using LZS'77
9  return $D$
```

Figure 4: Error-resilient LZ'77 algorithm. T is the text, e is the maximum number of errors that can be corrected in each block of $255 - 2e$ bytes.

entire set of buffers in primary memory. The problem can be alleviated by breaking up large inputs in chunks of a size that could be easily stored and processed in main memory.

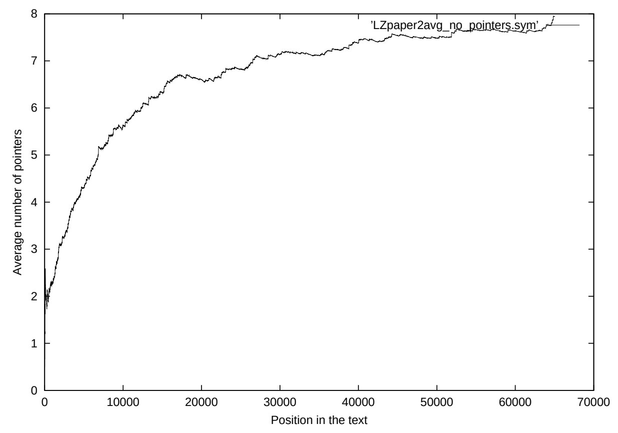
5 Experimental Results

In order to validate our theoretical studies and test the correctness of our scheme, we instrumented several implementations.

In the first one, we designed an implementation of LZ'77 based on suffix trees [9], and we kept track of the multiplicity q for each phrase of the LZ'77 parsing, when the length of the phrase is greater than two. The average value of q is shown in Figure 5, for increasing lengths of the prefixes. Note that for both graphs, the average for q appears to converge asymptotically to some constant, as the analysis in Section 3 suggests.

In the second, we modified the code of `gzip-1.2.4` to evaluate the impact of our method on compression performance. `gzip` is an implementation of the *sliding window* variant of LZ'77, that is, it issues pointers in a fixed-size window preceding the current position.

The modified `gzip`, called `gzipS`, implements directly LZS'77 as described in Section 2. It allows the user to specify a second file, which contains the text to be embedded in the pointers. The compression performance of the `gzipS` with respect to the original `gzip` was



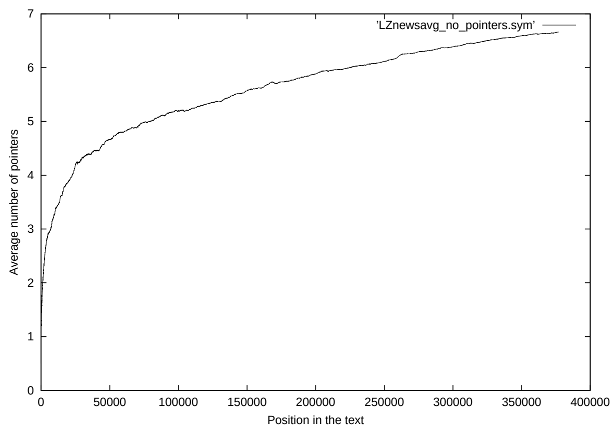


Figure 5: The average value of the pointer multiplicity q for increasing portions of `paper2` (left), and `news` (right) of the Calgary corpus

measured, and it is summarized in Table 1 on the Calgary corpus dataset. The embedding of the message slightly degrades the compression performance, in the order of 1%–2% on average for the files in the Calgary corpus. A file compressed with `gzipS` can be still be decompressed by `gzip`.

Finally, in the last implementation we coded the error-resilient LZRS'77. The prototype implementation is written in Python and linked to a public-domain library that implements the Reed-Solomon encoder/decoder [5]. Based on the considerations mentioned in introduction, we initially choose $e = 1$ and $e = 2$ which requires respectively at least 2 and 4 parity bytes on a block of data of size $255 - 2e$. We experimented the resilience to errors by introducing a controlled number of errors uniformly distributed over the b blocks of the compressed file. The graphs in Figure 6 shows the probability that the file did *not* uncompressed correctly for an increasing number of errors and for different choices of e and b. For example, using $e = 2$ over 100 blocks, LZRS'77 is able to decompress the file correctly with 20 uniformly distributed errors, 90% of the times.

References

[1] Atallah, M. J., and Lonardi, S. Authentication of LZ-77 compressed data. In *Proceedings of the ACM Symposium on Applied Computing* (Melbourne, FL, March 2003), to appear.

[2] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression.* Prentice Hall, 1990.

[3] Cover, T. M., and Thomas, J. A. *Elements of Information Theory.* Wiley Series in Telecommunication. John Wiley & Son, New York, NY, 1991.

[4] Jacquet, P. and Szpankowski, W. Autocorrelation on words and its applications. Analysis of suffix trees by string-ruler approach, *J. Combinatorial Theory. Ser. A*, 65, 237-269, 1994.

file size	`gzip`	`gzipS`	*file*	*redundant bytes*
111,261	39,473	39,511	`bib`	1,721
768,771	333,776	336,256	`book1`	14,524
610,856	228,321	228,242	`book2`	10,361
102,400	69,478	71,168	`geo`	4,101
377,109	155,290	156,150	`news`	5,956
21,504	10,584	10,783	`obj1`	353
246,814	89,467	89,757	`obj2`	3,628
53,161	20,110	20,204	`paper1`	937
82,199	32,529	32,507	`paper2`	1,551
46,526	19,450	19,567	`paper3`	893
13,286	5,853	5,898	`paper4`	249
11,954	5,252	5,294	`paper5`	210
38,105	14,433	14,506	`paper6`	738
513,216	62,357	61,259	`pic`	3,025
39,611	14,510	14,660	`progc`	736
71,646	18,310	18,407	`progl`	1,106
49,379	12,532	12,572	`progp`	741
93,695	22,178	22,098	`trans`	1,201

Table 1: The compression of "`gzip -3`" versus "`gzipS -3`" for the files of the Calgary corpus; the last column shows the total number of available bits for error correction

[5] Karn, P. General-purpose Reed-Solomon encoder/decoder in C. `http://www.ka9q.net/code/fec/`, 2002.

[6] Ma, F. Posting on newsgroup `comp.compression`, 2002.

[7] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *J. SIAM 8* (1960), 300–304.

[8] Reznik, Y. and Szpankowski, W. On average redundancy rate of the Lempel-Ziv codes with k-error protocol, *Information Sciences*, 135, 57-70, 2001.

[9] Rodeh, M., Pratt, V. R., and Even, S. Linear algorithm for data compression via string matching. *J. Assoc. Comput. Mach. 28*, 1 (Jan. 1981), 16–24.

[10] Roman, S. *Coding and Information Theory*, Springer-Verlag, New York, 1992.

[11] Sayood, K., Otu, H., and Demir, N. Joint source/channel coding for variable length codes, *IEEE Trans. Commun.*, 48, 787–794, 2000.

[12] Storer, J., and Reif, J. Error resilient optimal data compression, *SIAM J. Computing*, 26, 934–949, 1997.

[13] Szpankowski, W. *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, New York, 2001.

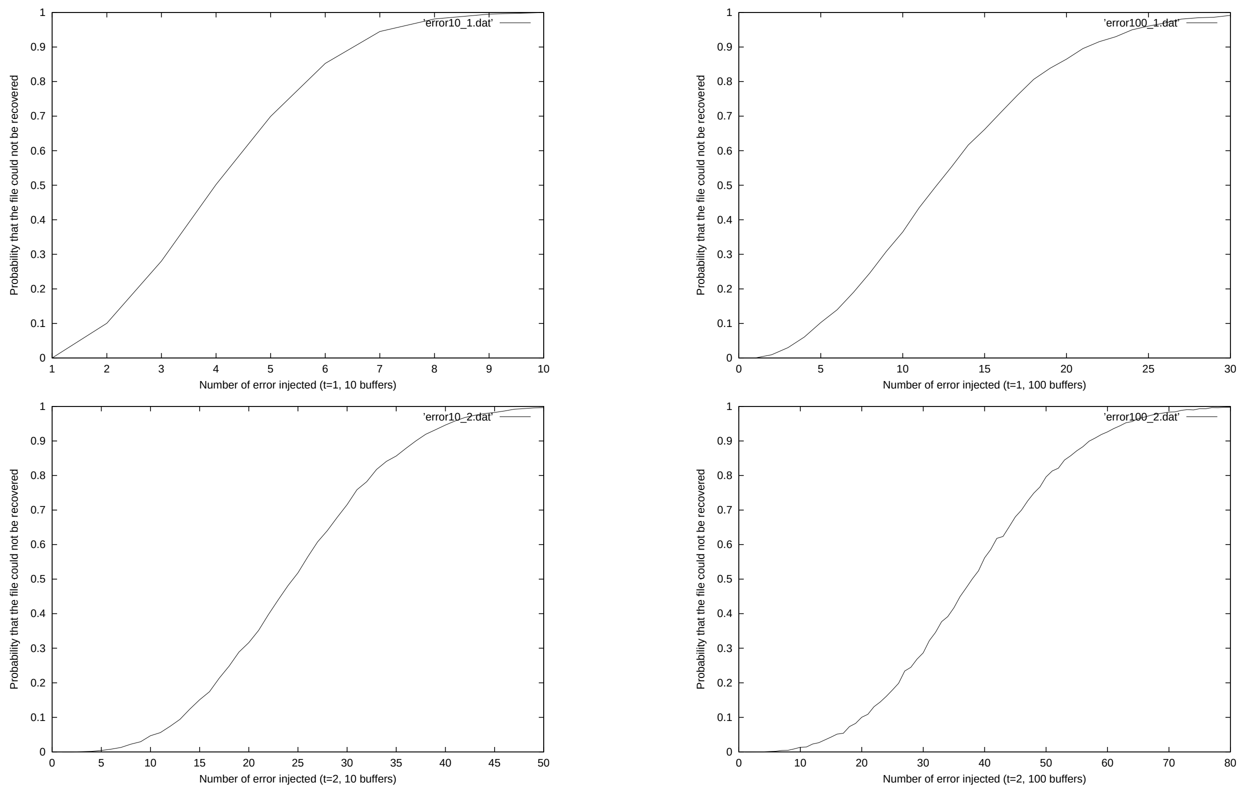


Figure 6: The probability that a file of b blocks could not be recovered correctly, for increasing number of errors uniformly distributed over the blocks. Top-left: $e = 1$ and $b = 10$, top-right: $e = 1$ and $b = 100$, lower-left: $e = 2$ and $b = 10$, lower-right: $e = 2$ and $b = 100$

[14] Wyner, A. J., and Ziv, J. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression, *IEEE Trans. Information Theory*, 35, 1250-1258, 1989.

[15] Wyner, A. J. The redundancy and distribution of the phrase lengths of the fixed-database Lempel-Ziv algorithm, *IEEE Trans. Information Theory*, 43, 1439–1465, 1997.

[16] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23:3, 337–343, 1977.