SETIT 2005
3rd International Conference: **S**ciences of **E**lectronic, **T**echnologies of **I**nformation and **T**elecommunications
March 27-31, 2005 – TUNISIA

Recycling Bits in LZ77-Based Compression

Danny Dubé* and Vincent Beaudoin*

Université Laval, Canada

Danny.Dube@ift.ulaval.ca

Vincent.Beaudoin.1@ulaval.ca

Abstract: We present a technique that exploits the multiplicity of the ways a text may be encoded using an LZ77-based compression method. With such methods, repeated parts of the text are encoded as length-distance pairs that refer to previously seen text. In general, given the maximum length of a repeated part, there may be more than one distance at which there is a copy of the repeated part. The compressor is free to select any of these distances since the decompressor is able to recover the same text anyway. The mere act of choosing one of these distances can be used to convey information to the decompressor. We present the details of our technique. We have integrated our technique in a high performance compressor in order to make measurements. The experiments show that our technique significantly improves compression rates on many files of the Calgary corpus.

Key words: data compression, LZ77 compression, multiplicity of encodings, recycling of bits, steganography.

1 Introduction

Many of the most used lossless data compressors use an algorithm that is a derivative of the substitution-based technique presented by Lempel and Ziv (Lempel and Ziv, 1977). This technique encodes the text by finding repeated parts and replacing them by references to preceding occurrences. The compressor takes care to do such replacements only when the reference happens to take less space than the part of the text that is being replaced. From now on, we will use the abbreviation LZ77 to refer to the original technique by Lempel and Ziv.

The popularity of the LZ77 derivatives comes from the good compression ratios that they achieve and their relatively high speed. There are techniques that achieve significantly better compression, such as those based on prediction by partial matching (PPM), but that tend to be slower (Cleary and Witten, 1984). The Burrows-Wheeler transform (BWT) is also very competitive when it comes to compression rates and speed (Burrows and Wheeler, 1994).

What we present here is a way to improve the compression rates that an LZ77-based compressor can achieve. The opportunity for improvement comes from the fact that, with an LZ77-based technique, there is a multitude of different ways to compress a single file. We exploit this fact by giving to the compressor the ability to send hidden messages to the decompressor. It does so by the mere act of choosing one way to compress over another.

Section 2 quickly reviews the fundamental principles of the LZ77-based techniques. Section 3 explains where, when, and how the compressor can send hidden messages to the decompressor in order to *recycle bits*. Section 4 describes our implementation of the technique. Section 5 presents the experimental results. Section 6 presents related work.

2 Review of LZ77-based compression

As with most compression techniques, LZ77-based compression consists in a way for a sender, the *compressor*, to describe to a receiver, the *decompressor*, what some original text is. The description made by the compressor goes over a communication channel, which is usually a compressed file. The compressor and decompressor need not work simultaneously but it helps to understand the process when we think that way. What we are interested in here (and what LZ77 is about) is *lossless compression*, which means that the text that the decompressor recovers from the description made by the compressor is identical to the original text.

The goal of this kind of communication is to have a description that is as short as possible. Hopefully, the description ought to be significantly shorter than the original text. The savings in the length of the description depends on the characteristics of the text to describe but also on the kinds of messages that the compressor and the decompressor use. In the case of LZ77-based compression, the messages are *literals* and *matches*.

2.1 Literals and matches

The description of the original text by the compressor is strictly sequential: at any given moment during the communication, a prefix of the original text has been described and is known to the decompressor while the rest of the original text still has to be described and is yet unknown to the decompressor. As long as the end of the original text has not been reached, the compressor continues explaining to the decompressor what the next bytes are.

The simplest means with which the compressor may continue its description is by sending a *literal byte*. That is, the message simply goes like: "the next byte in the text is *c*".

The (only) other means with which the compressor may continue its description is by sending the description of a *match*. A match is a correspondence between the sequence made of the very next bytes and an identical sequence located previously in the original text. In order to send a match, the compressor must find a copy of the next bytes in the part of the text that it has already described to the decompressor. The description of a match includes its length and the distance to its location. A message describing a match looks like: "the next *n* bytes are identical to the ones appearing *d* bytes before where we are now". Of course, the copy has to be located *before* the bytes that are being described.

Typically, for space-efficiency reasons, the compressor and decompressor only allow matches for which the corresponding copies lie at positions that are not too distant from the current position. The maximum distance allowed depends on particular algorithms. This way, the compressor and decompressor do not have to keep the whole text in memory. The set of positions that are within a reasonable distance from the current position and to which matches may refer is called the *window*. It is also often called the *sliding window* because its distant edge usually slides behind the current position at a constant distance.

Clearly, if compression is to be obtained, the compressor has to describe the text using matches whenever possible. Description using literals rarely leads to substantial compression. A match, on the other hand, allows the compressor to describe many bytes using an (almost) constant-length message.

The description process made by the compressor is a cyclic one. The first step of a cycle consists in indicating whether the compressor is going to send a literal or not. If so, the second step consists in sending the literal byte. Otherwise, the second step consists in sending the length and the distance of a match. After both steps have been performed, the cycle is finished and a new one begins.

The operations that we sketch here are only conceptual. Real implementations often combine steps and pieces of information together while they split others. Typically, each piece of information is carefully encoded by a statistical encoder. Frequently seen pieces of information are given short codewords and rarely seen ones are given long codewords. The encoding process is done by statistical encoders which encompass Huffman encoders (Huffman, 1952) and arithmetic encoders (Witten et al., 1987).

2.2 Selection of the longest match

When the compressor has the opportunity to describe the next bytes using a match, it almost always uses the *longest* match it can find. That is, it tries to find the position in the sliding window where as many as possible of the bytes there are identical to the next bytes to describe.

It makes sense for the compressor to always select the longest match because the description of an *(l+1)*-byte match is typically not much longer than the description of an *l*-byte match (if at all) while the former allows it to describe one more byte to the decompressor.

2.3 Selection of the closest longest match

Given the length *l* of the longest match, there may be more than one position in the window where a copy of the next *l* bytes is located. Clearly, the compressor may choose to send the distance to any of these positions as they all indicate the start of the same subsequence of bytes. How should the compressor select the distance that it is going to send?

While a compressor could select one of the valid distances randomly, this is not what is done by most implementations. Typically, it is the *shortest* of the distances to a longest match that is chosen. There are two reasons to do so.

The first reason is that, by systematically selecting the closest match, the statistical distribution of the transmitted distances tends to be uneven, with higher frequencies for the short distances. This allows the statistical encoders to take advantage of the unevenness and send shorter codewords on average.

The second reason is that some original texts have a tendency to be locally similar. That is, the characteristics of the original text are not constant throughout its whole length. Consequently, copies of a particular sequence of bytes often appear relatively close to the sequence itself. Such texts contribute to the unevenness of the distribution of the distances.

While these reasons suggest that it is profitable to always select the closest longest match, they do not seem as strong as the reason that suggests that it is profitable to always select the (or a) longest match. Indeed, not all texts are similar only locally. Also, when there is only one longest match, the compressor has to select it, no matter where it stands in the window. The latter situation tends to flatten the distribution of the distances. Not so surprisingly, we propose to abandon the closest longest match strategy for another one.

3 Recycling bits

The strategy that we propose still dictates that we should use the longest match but it also says that we should select the distance more wisely when there is more than one match. Indeed, instead of simply contributing to the unevenness of a statistical distribution, the compressor will use the chance that it has to select among many distances to implicitly transmit bits to the decompressor. In a sense, these implicit bits "come for free" as the description of a match has to involve the transmission of both a length and a distance anyway. We call this process of implicitly transmitting bits the *recycling of bits*.

3.1 Choosing between two matches

Let us make an example to explain how the compressor can implicitly transmit a bit to the decompressor.

Suppose that the window contains the bytes "0abc1abc2" and that the next bytes to describe are "abc3". Clearly, the longest match has length 3 and there happens to be two longest matches: one at distance 4 and one at distance 8.

According to our strategy, the compressor indicates to the decompressor that it is going to send a match and that the match has length 3. Then, it has to decide whether it describes the match using distance 4 or using distance 8. Both distances would allow the decompressor to correctly determine the next 3 bytes. According to our strategy, the decision should not be taken gratuitously. The compressor has the opportunity to implicitly send a bit of information here. The compressor can implicitly send a 0 by selecting (say) distance 4, and a 1 by selecting distance 8. Suppose that the compressor needs to implicitly send a 1, so it selects distance 8 and transmits it.

Now, how can the decompressor detect the bit of information that the compressor has never sent through the bit stream explicitly? It does so by first proceeding as usual: it reads the messages from the compressor that say that it is going to receive a match, that the match has length 3, and that the copy is located at distance 8. From this information, the decompressor determines that the next 3 bytes are "abc". It is at this point that the decompressor can detect whether implicit bits were transmitted and, if so, what these are. It searches for all the possible matches with "abc" that existed in the window when the compressor transmitted the match. In our example, it is able to detect that distances 4 and 8 could have been used. Since there are two possibilities, it knows that the compressor has sent an implicit bit through the choice of the distance. Finally, by the fact that distance 8 has been selected, the decompressor concludes that it has implicitly received the bit 1.

The most obvious way for the decompressor to use the implicitly transmitted bit is to put it in front of its input bit stream. This causes the added bit to be part of the next message from the compressor. Or, considered from another point of view, it shortens by one bit the length of the next message from the compressor. We say that the bit has been *recycled*.

3.2 Choosing between many matches

We just showed through an example how the compressor could implicitly transmit a bit to the decompressor simply by intentionally choosing one of two valid distances for a match.

In a similar fashion, it is relatively simple to see how the compressor could implicitly send 2 bits to the decompressor if it had 4 valid distances among which to choose. More generally, the compressor is able to implicitly transmit n bits to the decompressor when there are 2^n distances among which to choose. In particular, note that when there exists only one distance to a longest match, then the compression can implicitly transmit 0 bit or, in other words, none.

In fact, there is no real need to have a number of distances that is a power of 2. The compressor and decompressor simply need to have the ability to build prefix codes for n equiprobable events on the fly. For instance, when one of 5 distances is to be chosen, both the compressor and the decompressor may build the following prefix code: 000, 001, 01, 10, 11. Depending on the choice made, one of the 5 bit sequences is implicitly transmitted. It is easy to devise a method for efficiently emitting and reading prefix codes for n equiprobable events on the fly. We will consider this problem as solved and denote as e_{in} the codeword corresponding to the ith of n equiprobable events.

3.3 Making the choices

Having the ability to transmit bits implicitly is useless if the latter do not carry meaningful information. As we suggest at the end of Section 3.1, a sensible way of using the implicitly transmitted bits is by putting them back in front of the input bit stream of the decompressor. However, as sensible as it may be for the decompressor to do so, it creates a non-trivial problem for the compressor.

Indeed, when the compressor has to transmit a distance to the decompressor, it may have to choose among more than one distance. It has to choose among the distances in such a way that a particular sequence of bits gets implicitly transmitted. But these bits are intended to be part of the description of the next literal or the next match. However, at this point, the compressor normally has not yet decided what to do with the next match or literal and may even be ignorant of whether it is going to be a match or a literal to start with!

In fact, the situation is even worse. Suppose that the compressor did determine that the next message it will have to send describes a match of length (say) 4, this next match may also be described by more than one distance. If it does not know for now how the next match will be encoded, i.e. using which sequence

of bits, it may not be able to decide which distance it has to choose for the current transmission. In turn, there may be a second next match after the next one and it may be yet undecided too. In the worst case, the choice of a distance for the current transmission may depend upon an arbitrarily long chain of yet undecided matches.

So, how can the compressor decide which distance to choose when the choice may depend so much on future matches? The solution we propose is to do the compression in three phases. The first phase consists in *searching for the matches and keeping all the information about them*. The information the compressor has to keep about a match is its length and the set of distances to the copies. The information it has to keep about a literal character is simply the character itself. The second phase consists in *solving the matches*. We describe this phase just below. The third phase *consists in sending the bit stream* that has been produced by the second phase.

The first and third phases are pretty straightforward and we will not provide more explanations about them. For convenience, in our explanations about the second phase, we will use the term *message* to refer indiscriminately to a literal character or a match.

Let us consider that the first phase has divided the original text into M messages, numbered 0 to $M-1$. We are going to solve the matches by computing a bit stream for each suffix of the sequence of M messages. Let s_i be the bit stream that would allow the decompressor to decode the necessary information about messages i to $M-1$. In particular, s_M denotes the bit stream that describes the empty sequence of messages. Also, s_0 denotes the bit stream that describes all the messages. Note that s_M may include the end of file indicator. There are many well-known ways of transmitting the length of the original file in a compressed file. To keep our presentation simple, we choose to ignore the end-of-file issue.

Now, we explain how the resolution process is done. We compute the $M+1$ bit streams backwards, that is, from s_M to s_0. We begin by letting s_M be the empty stream:

$$s_M = \varepsilon$$

Then, we inductively compute s_i from s_{i+1}. There are two cases to consider: the one where message i is a literal; and the one where message i is a match.

Let us first consider the case where message i is a character literal. Let c be that character. This case is easy as there are no choices to make. Let w be the sequence of bits that indicate that a literal is to be transmitted and that the transmitted character is c. Then, the new bit stream is obtained by adding w in front of it:

$$s_i = w\, s_{i+1}$$

Next, we consider the case where message i is a match. Let l be the length of the match, n be the number of distances to longest matches, and d_j be the jth distance, for j going from 1 to n. Let w_j, for j going from 1 to n, be the sequences of bits that indicate that a match of length l located at distance d_j is transmitted. The first step consists in factoring s_{i+1} into an implicitly transmitted part and an explicitly transmitted part:

$$s_{i+1} = e_{jn}\, t$$

That is, a prefix codeword representing the jth of n events is extracted from the front of s_{i+1}. The remainder of the bit stream is denoted as t. The extracted codeword indicates to the compressor which distance it should select in order for the decompressor to recover the correct sequence of implicitly transmitted bits. All there remains to be done is to add the appropriate sequence in front of the explicitly transmitted part:

$$s_i = w_j\, t$$

This almost concludes the presentation of the resolution process. There only remains a little problem. When we extract a codeword e_{jn} from the front of a bit stream, we presume that there are enough bits in the bit stream for the extraction to succeed. This may not always be the case. In particular, if the last message is a match that possesses more than one distance, the extraction of a non-trivial codeword will be attempted on s_M, the empty stream.

The simplest approach consists in aborting the resolution and redefining s_M as a stream of one 0 bit. If, during the second execution of the resolution, an infeasible extraction is once again attempted, then another 0 ought to be added to s_M. Yet another execution of the resolution has to be performed. As many 0s should be added as there are attempts at illegal extractions. Aborted resolutions are not costly since only the last match or the last few matches may attempt illegal extractions. The bits that are added to the stream quickly outnumber the bits that are extracted. The bits that have to be artificially added to s_M can be seen as superfluous bits that occur after the end of the bit stream consumed by the decompressor.

3.4 Meaning of the recycled bits

The possibility of obtaining "free" bits just by carefully choosing the distance of the matches may seem surprising at first. However, we could devise another variant of LZ77 compression that would provide essentially the same benefits as those provided by the recycling of bits.

The other variant would go like this. Almost all the communication scheme between the compressor and the decompressor would stay the same except for the transmission of the distances. When the compressor intends to encode a length l match, instead on transmitting a distance, it would build a Huffman code for all the length l subsequences of bytes that are present in the sliding window and then transmit the codeword corresponding to the matched subsequence. The idea is that each subsequence would be considered as a symbol. Given a codeword that the decompressor would receive, the latter would simply access an equivalent Huffman code of its own and be

able to recover the subsequence of bytes meant by the compressor.

During the construction of the Huffman code for the subsequences, identical subsequences would be considered as the same symbol. Consequently, more frequent subsequences would turn into more frequent symbols, to which shorter codewords would be assigned. For instance, a subsequence that occurs 4 times would likely be assigned a codeword that is 2 bits shorter than the one assigned to another subsequence that occurs only once. In our technique, these 2 bits are saved using recycling.

Even if both techniques would offer roughly the same level of compression, ours remains pretty efficient as the set of distances for a match is small on average compared to the total number of subsequences present in the sliding window.

4 Prototype

We have implemented our technique in a well-known compressor called GZIP (Gailly and Adler). GZIP uses the Deflate compression method which is a very effective derivative of LZ77 compression (Deutsch, 1996). We briefly present the characteristics of the Deflate method and then we sketch the modifications that we made to GZIP.

4.1 Characteristics of GZIP

GZIP uses a sliding window of 32 kB. A hash table is used to find matches efficiently. The transmission of a message consists either in the transmission of a literal character or in the transmission of a length-distance pair. A single Huffman code describes both the literals and the lengths. Upon reception of such a codeword, the decompressor immediately knows whether it has received a literal character or the length part of a match. In the case of a match, a codeword drawn from another Huffman code is transmitted. Thanks to the Huffman codes, frequently used literals, lengths, and distances are given shorter codewords.

GZIP performs the compression in two phases. It first looks for the matches and gathers information about them. Based on this information, it builds the two Huffman codes (one for the literals-and-lengths and one for the distances). Only then are the literals and matches really encoded into bits.

GZIP always selects the closest (longest) match. Consequently, it tends to produce unevenness in the statistics of the distribution of the distances which in turn contributes to improve compression through the use of the Huffman codes. In fast modes, GZIP does not necessarily look for the longest match. It may decide to transmit only a "sufficiently long" match it has found. In some circumstances, it will drop a match in favor of a literal but only because it has found that a significantly longer match was available after the literal.

The Deflate method requires matches to be at least 3 bytes long. This causes GZIP to emit many literals. However, GZIP encodes literals effectively because of its literals-and-lengths Huffman code.

Finally, GZIP is able to change the Huffman codes used to encode the information. It can do so whenever it sees fit by terminating a block of compressed data and starting another. Also, when some part of the original text does not seem compressible, it can revert to plain storage mode, ensuring that the compressed text will barely be bigger then the original one, if at all.

4.2 Modifications to GZIP

We needed to make only a few modifications to GZIP. First, since GZIP already uses a multi-phase approach, it was simple to add the mechanisms that collected the sets of distances to the matches. The sets of distances were obtained using GZIP's hash table.

Second, we added the "match resolution" phase (see Section 3) between GZIP's two phases. Third, we modified the way it emits distances. Instead of using the encoding of distances provided by its Huffman codes, we transmit distances as plain 15-bit numbers. Since our technique encodes a match by describing any one of the longest matches, and not always the closest one, we expect the chosen matches to appear anywhere inside of the sliding window, resulting in a flat statistical distribution of distances.

Corresponding changes were made to the implementation of the decompressor. The main difference is that all the operations are performed in a single phase in the decompressor.

To summarize, all the features of GZIP are kept intact in our prototype except for the encoding of the distances. Consequently, one would expect a clear advantage of our technique over ordinary Deflate method since ours is able to recycle bits.

Unfortunately, this is not as simple as that. While our technique does save bits through recycling, it does so compared to a modified Deflate method where all distances are encoded on 15 bits. Now, we must recall that in ordinary Deflate, distances are encoded using a Huffman code and that the statistical distribution of the distances is far from flat. So the effective strategy of GZIP is replaced by one that we hope will be effective too. Note that the circumstances under which our approach is able to recycle many bits are those where many longest matches are available. These are exactly the circumstances under which it is likely that there exists a close longest match.

However, we expect our approach to perform well because it takes advantage of the multiplicity of the longest matches "at the source". That is, the existence of n longest matches allows our approach to immediately recycle about $\log_2(n)$ bits. On the other hand, ordinary Deflate can only expect to increase the bias in the statistical distribution of the distances. This distribution blends together the distributions for matches where many distances are available and those where few or only one are available. According to this

reasoning, ordinary Deflate should not be able to save as much as $\log_2(n)$ bits in the same context.

5 Experimental results

We ran some experiments to compare the performance of our technique to that of GZIP. We compressed the files of the Calgary corpus (Witten and Bell, 1990). The size of the files is presented in the following table. The second column presents the size of the original files. The following two columns show the size of the files compressed using GZIP in maximum compression mode and the size of the files compressed using our prototype. The sizes are all measured in bytes.

Name	Original	GZIP	Prototype
Bib	111 261	34 900	34 305
book1	768 771	312 281	301 955
book2	610 856	206 158	202 430
geo	102 400	68 414	66 542
news	377 109	144 400	142 808
obj1	21 504	10 320	11 147
obj2	246 814	81 087	84 289
paper1	53 161	18 543	18 783
paper2	82 199	29 667	29 401
paper3	46 526	18 074	18 198
paper4	13 286	5 534	5 986
paper5	11 954	4 995	5 467
paper6	38 105	13 213	13 783
pic	513 216	52 381	54 257
progc	39 611	13 261	13 852
progl	71 646	16 164	16 653
progp	49 379	11 186	11 787
trans	93 695	18 862	19 340

The results show that our technique can provide substantial improvements in some cases and substantial deteriorations in some other cases. Our technique is especially effective on text files. We explain this by the fact that these files contain mostly word-based redundancy. Moreover, occurrences do not seem to occur in clusters which would not penalize our flat 15-bit distance encoding too much.

On the other hand, the greatest difficulty for our technique comes mostly from the binary files. In particular, in the case of pic, we suspect that the copies of sequences of bytes are not distributed evenly in the file. A scheme that provides Huffman codes for the distances probably has an advantage over one that uses flat codes.

Still, it is difficult to analyze these results and draw sound conclusions without extracting more statistics about the compression of each file and being able to accurately identify the precise causes.

We took measurements of the compression times of GZIP and our prototype. The added mechanisms are not as costly as they may seem at first. We observed an increase of 0% to 100% in the compression times for our prototype over GZIP on all the files except for pic. In the case of pic, the compression times increased by a factor of about 50. This is caused by the very high redundancy found in the file. A large zone at the end of the file contains only bytes set to zero. At some point, every longest match (258 characters long, which is the maximum) can be described by every possible distance in the sliding window. This causes the sets of distances to contain up to about 32 000 distances each.

6 Related work

Other techniques have been devised that exploit the multiplicity of the encoding of a file using LZ-77 derivatives.

In particular, the idea of choosing one distance among many is used in a system that provides authentication for data compressed using LZ-77 (Atallah and Lonardi, 2003). Similarly to what our technique does, the authentication system transmits bits implicitly to the decompressor. However, these bits are used not to improve compression but to send a proof of the authenticity of the document. The authenticated document can be decompressed by any "ordinary" decompressor. More generally, the authors explain how compression with LZ-77 derivatives can be used to hide information inside of compressed files, i.e. to perform *steganography*. The authors use this ability to achieve authentication. They mention that the only previous work they were aware of that combines information hiding and text compression is from Cachin (Cachin, 1998).

In another work (Lonardi et al., 2004), the authors use the implicitly transmitted bits to embed error-correcting codes. A document that is compressed in this way can be decompressed by an ordinary decompressor but, if the appropriate decompressor is used, the latter is able to extract the error-correcting codes in addition to the original document and so, even when errors have occurred during transmission of the compressed document.

We also found a steganography tool that exploits the properties of LZ-77 compression (Brown, 1994). The tool allows one to hide a file into the compressed version of another (sufficiently long) file. The technique that is used there to transmit bits implicitly is different from that of the previous works and ours. When the compressor finds a match that is long enough, it may choose to transmit the length of the match shortened by 1, depending on the next hidden bit that is to be sent. The author mentions that it causes the compression to degrade only slightly. Once again, the compressed file (the container) can be decompressed using an ordinary decompressor despite the presence of the hidden file.

In order to briefly compare our work with previous ones, we say that our technique exploits the potential to hide information based on the selection of one distance among many in order, that this ability is used to send parts of the compressed document itself, and

that documents compressed using our technique are impossible to decode using an ordinary LZ77 decompressor.

Conclusion

We presented a variant of LZ77 compression that exploits the fact that this compression method allows multiple ways to encode texts. We exploited the fact that matches can often be described by more than one length-distance pair, even when we restrict ourselves to the longest matches. We showed how the availability of many such pairs provided a way for the compressor to implicitly send information to the decompressor simply by carefully selecting one particular distance over others. Although the idea of implicit transmission happens not to be original (Atallah and Lonardi, 2003; Lonardi et al., 2004), the use of the implicit transmission to send parts of the compressed file itself in order to improve compression is new. We call such a technique "bit recycling". We presented the algorithms that allow one to implement bit recycling.

We have implemented our technique by making some modifications to the well-known GZIP compressor. We showed experimentally that we could obtain improvements in the compression of about half of the files of the Calgary corpus and that, in some cases, this improvement could be substantial.

Acknowledgments

The authors wish to thank the National Science and Engineering Research Council of Canada and Université Laval for supporting this research.

References

Atallah M.J. and Lonardi S., "Authentication of LZ-77 Compressed Data", in Proceedings of the 18th ACM Symposium on Applied Computing, Melbourne, Florida, pp. 282-287, 2003.

Brown A., gzip-steg, 1994, http://www.mirrors.wiretapped.net/security/steganography/gzip-steg/gzip-steg-README.txt.

Burrows M. and Wheeler D.J., "A Block-Sorting Lossless Data Compression Algorithm", Technical report no. 124, Digital Equipment Corporation, Palo Alto, California, 1994.

Cachin C., "An Information-Theoretic Model for Steganography", in Proceedings of the Workshop on Information Hiding, volume 1525 of Lecture Notes in Computer Science, pages 306-318, Springer-Verlag, Berlin, 1998.

Cleary J.G. and Witten I.H., "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions on Communications, Vol. 32, No. 4, pp. 396-402, 1984.

Deutsch P., "Request for Comments: 1951", 1996, http://www.ietf.org/rfc/rfc1051.txt.

Gailly J.L. and Adler M., The GZIP Compressor, http://www.gzip.org/.

Huffman D.A., "A Method for Construction of Minimum Redundancy Codes", Proceedings of the IRE, Vol. 40, No. 9, pp. 1098-1101, 1952.

Lonardi S., Szpankowski W., and Ward M.D., "Error Resilient LZ'77 Scheme and its Analysis", Proceedings of the 2004 International Symposium on Information Theory, Chicago, p. 56, 2004.

Witten I.H. and Bell T.C., "The Calgary/Canterbury Text Compression Corpus", 1990, ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/.

Witten I.H., Neal R.M., and Cleary J.G., "Arithmetic Coding for Data Compression", Communications of the ACM, Vol. 30, pp. 520-540, 1987.

Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343, 1997.