A Secure, Lossless, and Compressed Base62 Encoding

Kejing He*, Xiancheng Xu†* and Qiang Yue‡
*School of Computer Science and Engineering, South China University of Technology, Guangzhou, China 510641
†School of Informatics, Guangdong University of Foreign Studies, Guangzhou, China 510420
‡Guangdong Nortel R&D Center, 380#, Bin Jiang Zhong Road, Hai Zhu District, Guangzhou, China 510220
Email: kejinghe@ieee.org, xcxu@gdufs.edu.cn, stevenyue@gdnt.com.cn

***Abstract*—In many applications and protocols, the number of usable characters is limited. Base64 or other similar algorithms are widely used to encode the binary data into a printable representation. But base64 itself can't compress data, contrarily, it inflates the binary data by 33%. This paper presents a lossless base 62 compressed encoding method that only use alphanumeric characters to represent the original data. Burrows-Wheeler Transform (BWT) is used along with range coding to compress data before it is transfered to base 62 encoder. And a simple encryption mechanism is incorporated to provide basic security. This base62 compressed encoding has been tested using the Canterbury corpus, and this paper also presents a comparative study with other similar methods. It can be used for compressing data into base 62 format in wide-range applications, e.g., URL encoding, Internet Mail Transfer, and embedding binary objects into XML files.**

I. Introduction

There are numerous situation where non-textual is unusable. For example, the Simple Message Transfer Protocal (SMTP) supports only ASCII characters, and only alphanumerics, the special characters "$-_.+!*'(),", and reserved characters used for their reserved purposes may be used unencoded within a URL [1]. To transfer or represent other unusable characters using these constrained protocols or systems, different encoding methods have been developed to represent binary data using only these usable characters. Among them, the most famous one may be the binary-to-text base64 encoding [2], which was used for sending other non-ASCII data in Privacy-Enhanced Mail (PEM) [3] and Multipurpose Internet Mail Extensions (MIME) [4]. However, base64 encoding itself doesn't have compression ability, data encoded with base64 will inflate to 133% of its original size. And the non-universal selection of the 63rd and 64th characters may, if improperly implemented, lead to mis-decoding of the encoded data. The 62 alphanumeric characters (A-Z, a-z, 0-9) are the undebatable choice of first 62 symbols, but different systems have different restriction on the selection of the last two symbols. MIME selects "+" and "/", but for filename and URL, the best candidates may be "-" and "_". Along with the popularity of XML and service-oriented applications [5], XML has become the *de facto* standard for data transfer of all types of applications. To embed binary data into XML files, an efficient compressed binary-to-text encoding method is necessary. There are also a few studies integrated general compression methods, mostly dictionary-based Lempel-Ziv (LZ) [6] variations, with base64. LZJU90 encoding [7] first compresses the binary data, using a DEFLATE [8] algorithm. It then encodes each 6 bits of the output of compression as a text character. However, it doesn't include any intrinsic security mechanism.

LZ-derived methods are fast and widely used in many programs (e.g. ZIP file format, Graphics Interchange Format). Prediction by Partial Matching (PPM) [9] is another adaptive statistical data compression method based on context modeling and prediction. It compresses most files more effectively than gzip or ZIP but is slower. Burrows-Wheeler Transform (BWT) [10] is a block-sorting compression method, and its performance is in the middle of LZ variations and PPM. Its compression rate is better than LZ-based methods, and its speed is faster than PPM. The typical use of BWT is the Unix bzip program. Bzip uses BWT with adaptive arithmetic coding [11], and due to the US patent protection of arithmetic coding, bzip2 uses BWT with adaptive Huffman coding [12].

This paper presents a secure and lossless base62 data compression. We integrate BWT with adaptive range coding [13], which is theoretically identical to arithmetic coding and is free from patent claims. After range coding, the data stream is passed through a base62 encryption encoder, and the output is the encrypted alphanumeric representation of the original data. This paper is organized as follows. Following the introduction in Section I, Section II presents the lossless base62 compressed encoding method. Section III tests the base62 encoding on the Canterbury corpus and compares it with gzip, bzip2, LZJU90, and other compression methods. Finally, some conclusions and discussions are made in Section IV.

II. The Method

The lossless base62 compressed encoding is composed of 6 steps, among which, the first Run Length Encoding (RLE) step is optional. The schematic workflow is illustrated in Fig. 1. According to the Shannon information theory, no file can be compressed smaller than its information entropy, and the compressibility is determined by the redundancy of original data. The purposes of all the steps are to get rid of the information redundancy maximally. They will be detailed in the following sub-sections.

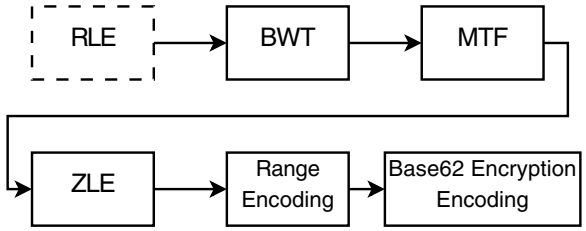

Fig. 1. The flowchart of lossless base62 compressed encoding.

A. Run Length Encoding (RLE)

The basic idea behind Run Length Encoding is to replace consecutive occurrences of identical characters with a single character value and count pair. This is most efficient on data that contains many long runs: for example, faxed document and bitmap files. However, there are not many repetitions in normal binary and text data, so this RLE step is optional - only applicable when the data contains many long runs.

B. Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform is a reversible transformation that can make a given data more localized. BWT itself neither compress data, nor change character values. It only permutes the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times. This is useful for succeeding compression, since it tends to be easy to compress a string that has runs of repeated characters.

C. Move-To-Front (MTF)

MTF [14] transform does not alter the size of the processed data. It maintains an array of most recently used (MRU) characters, and each processed character is replaced by its index in the MRU array. The effect is that consecutive occurrences of any arbitrary characters become runs of zeros, and frequent recurring characters become low integers. After MTF, the processed data contains many long runs of zeros and much more low intergers than high intergers. And there are other researches use different schemes to maintain the MRU array. The classic MTF always moves the most recently used character to position 0 in the array. The first modification [15] changes the scheme slightly: if the MRU character was at any position higher than 1, it is moved to position 1. If the MRU character was at position 1, it is moved to position 0. And the second modification [16] changes the rule further by restricting the character moving from position 1 to position 0 only when the last recently used character was not at position 0.

D. Zero Length Encoding (ZLE)

Data processed by MTF is dominated by long runs of zeros, and runs of nonzero bytes are rare. We use Zero Length Encoding (ZLE) to explicitly compresses long runs of zeros. The algorithm is working on the streaming mode. When met a run of N zero bytes, it translate N+1 into binary, elides the most significant bit (which is always 1), and output the remaining bits. When met a nonzero (literal) byte $b(b < 0xFE)$, it outputs the binary presentation of $b+1$. If b equals 0xFE, it outputs 0xFF 00. If b equals 0xFF, it outputs 0x FF 01. And the decoder works based on the same mapping accordingly. This algorithm is extremely efficient and was used in bzip, bzip2.

E. Range Encoding

Most compression softwares use arithmetic coding or Huffman coding as the entropy encoding method. However, the arithmetic coding, though is almost optimal, is a patent encumbered technique. It means arithmetic coding cannot legally be used without obtaining one or more licenses. Due to the patent protection, some systems had to throw the superior arithmetic coding away and replace it with Huffman coding. Although Huffman coding offers worse compression than arithmetic coding, it is simpler and faster than arithmetic coding.

We use range encoding [13] after Zero Length Encoding to save values in an optimal way. Range encoding is mathematically equivalent to arithmetic coding, and free from patent encumbrance. Unlike arithmetic coding that encodes symbols into the numerators of fractions, range encoding encodes all the symbols of the message into one big integer number. Also, it is differ from Huffman coding which assigns each symbol a code and concatenates all the codes together.

F. Base62 Encoding

The purpose of final base62 encoding is not compression, but to make the compressed data printable and portable. In this step, 62 alphanumeric characters are used to represent 256 different binary characters. The base62 encoder operates in the streaming mode, where the codec inputs a byte or several bytes, processes them, and continues until an end-of-file is sensed. Similar to base64 encoding, the base62 encoder reads 6 bits from the input stream each time, and try to encode it into an alphanumeric character. But since 6 bits lead to $2^6 = 64$ different combinations and we have only 62 alphanumeric characters, a special handling is introduced. If the first 5 bits is 11110, base62 encoder will encode it into the 61st character, and the 1 bit left was postponed for next round encoding. If the first 5 bits is 11111, base62 encoder will encode it into the 62nd character, and the 1 bit left was also postponed.

Base62 encoding use an automatic intelligent method to handle the padding of last several bits. If the length of last bits is less than 6 (5 for 11110 and 11111), the encoder will expand it to 6 bits by adding leading zeros, and encode the expanded bits as normal. The decoder works the similar way. If the decoder find the decoded bits can't match the byte-alignment, it will know how many leading zeros need to be removed, and after that the bits can be put into the last byte of output tightly.

The pseudocode of base62 encoder is defined as follows:

BASE62-ENCODE (IS, OS, T)

1 ▷ IS and OS are the input and ouput stream respectively. T is the 62-characters-length symbol table.
2 $bitLeft \leftarrow$ NULL

```
3  bits ← READ6BITS (IS)
4  while ISENDOFSTREAM (IS) = FALSE
5     bitLeft ← ENCODE-6BITS (OS, T, CONCATENATE (bitLeft, bits))
6     if LENGTH (bitLeft) > 0
7        bits ← READ5BITS (IS)
8     else
9        bits ← READ6BITS (IS)
10 ▷ padding of last several bits
11 bits ← CONCATENATE (bitLeft, bits)
12 if LENGTH (bits) > 0
13    bits ← PADDING (bits)
14    ENCODE-6BITS (OS, T, bits)
```

In real implementation, you can read one byte or several bytes at a time rather than 5 or 6 bits to get better performance. The ENCODE-6BITS (OS, T, $bits$) used in BASE62-ENCODE is defined as follows:

```
ENCODE-6BITS (OS, T, bits)
1  ▷ Array indexes start at 0.
2  if bits < 60
3     WRITECHAR (OS, T[bits])
4     return NULL
5  else if bits < 62
6     WRITECHAR (OS, T[60])
7     return bits[5]
8  else
9     WRITECHAR (OS, T[61])
10    return bits[5]
```

The pseudocode of base62 decoder is defined as follows:

```
BASE62-DECODE (IS, OS, T)
1  ▷ IS and OS are the input and ouput stream respectively.
   T is the 62-characters-length symbol table.
2  for i ← 0 to 61
3     CharToBits[T[i]] ← i
4  char ← READCHAR (IS)
5  while ISENDOFSTREAM (IS) = FALSE
6     if char = T[60]
7        WRITE5BITS (OS, "11110")
8     else if char = T[61]
9        WRITE5BITS (OS, "11111")
10    else
11       WRITE6BITS (OS, CharToBits[char])
12    char ← READCHAR (IS)
```

Fig. 2 illustrates an example of base62 encoding that encodes 3 bytes into 5 encoded alphanumeric characters. In this example, the ending two bits are expanded to 6 bits and encoded as 0x000010.

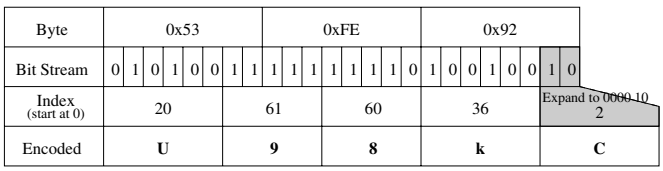

Fig. 2. An example of base62 encoding.

Using this scheme, 60 six bits and 4 five bits will be encoded into 64 symbols averagely. So the bit rate is $\frac{64\times 8}{(6\times 60+5\times 4)/8} \approx 10.779bpc$. Its compression rate is slightly (0.28%) worse than perfect base 62 numerical system, in which the compression rate is $\frac{8}{(\log_2 62)/8} \approx 10.749bpc$. But the base 62 encoding presented here is much faster than the traditional base 62 to binary conversion which needs a lot of expensive modulo operations.

G. Simple Base62 Encryption

The base62 encoding presented above doesn't use any encryption key, so it looks like an obfuscation method more than an encryption method. The traditional base64 is similar, the cracker could intercept the encoded message as it pass over the Internet, and can decode the message without any obstacles.

The pseudocode of base62 encoder and decoder shows that the system works properly by using the same usable symbol table T. As long as the sender and receiver have the same symbol table T, the message can be decoded correctly. The base64 encoding use *A-Za-z0-9+/* as the 1st to the 64th symbols, and for the normal base62 encoding above, we may use *A-Za-z0-9*. Since the decoder only replys on the same table used by the encoder, which means the symbol table T doesn't have to be ordered, we can incorporate basic encryption mechanism into base62 encoding by shuffling the symbol table T. In the simple base62 encryption, we use a secret key key as the seed of a random number generator. The random generator initialized by key is used for generating a random permutation of the ordered symbol table T (*A-Za-z0-9*). And original messages are base62-encoded using the shuffled symbol table T'. The pseudocode of simple base62 encryption is defined as follows:

```
BASE62-ENCRYPT (IS, OS, key)
1  ▷ Encrypt the input stream IS using cipher key key and
   send the encrypted message to output stream OS.
2  T ← {A-Za-z0-9}
3  SETRANDGENSEED (key)    ▷ Set seed for the random
   number generator
4  T' ← FISHER-YATES-SHUFFLING (T)
5  BASE62-ENCODE (IS, OS, T')
```

The FISHER-YATES-SHUFFLING algorithm used in BASE62-ENCRYPT is an efficient shuffling method that generates a random permutation of a finite set. It was first published by R.A. Fisher and F. Yates [17] in ordinary language, and by R. Durstenfeld [18] in computer language. The security of BASE62-ENCRYPT depends on the strength of the 62-to-62 alphanumeric substitution, which is random permutation of the alphanumeric characters.

III. ANALYSIS AND EXPERIMENTS

In the simple base62 encryption, the size of the key space depends not only on the substitution box but also on the random number generator. In the BASE62-ENCRYPT algorithm, the alphanumeric substitution box is determined solely by the random number generator, which is determined by the random

number seed (or the cipher key key). So the size of key space, or the ability against brute force attack, equals to the minimum of 62! and the possible values of cipher key key.

Ordinaryly, simple substitution cipher can be easily broken using statistical techniques, such as frequency analysis. Letters do not occur with uniform frequency in an average text. And in normal simple substitution cipher, the ciphertext preserves the letter frequency of the plaintext. This information can be used to guess the mapping of substitution box. To investigate the ability of this simple substitution method againsting statistical attack, we studied the frequency distribution of 62 characters in base62 encoding.

Fig. 3 illustrates the mean symbol frequency and standard deviation of eight base62-encoded Canterbury corpus textual files. These eight textual files follow similar symbol distribution pattern. It convinced that symbols occur with varying frequencies after base62 encoding, and the frequency standard deviation of different files is small. These patterns and information have the potential to be exploited for the ciphertext-only attack.

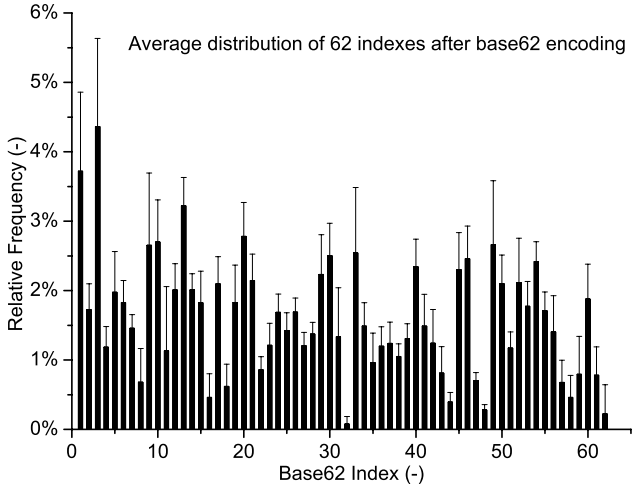

Fig. 3. Mean symbol frequency and standard deviation of eight non-compressed base62-encoded Canterbury corpus files. Sixty-two symbols occur with varying frequencies, and the frequency standard deviation of eight files is small.

According to Shannon's theory, a cipher will become less vulnerable when the redundant information is removed. So, to increase the security, we can compress the data before pass it to the base62 encoder. As illustrated in Fig. 1, the data stream was passed through the Burrows-Wheeler encoder, the move-to-front encoder, the zero length encoder, the range encoder, successively. Fig. 4 illustrates the mean symbol frequency and standard deviation of 11 base62-encoded Canterbury corpus files. These files are compressed using BWT, MTF, ZLE, and range encoding before being passed to base62 encoder. The frequencies of first 60 symbols are almost the same (around 1.6%). And the frequency of the 61st and 62nd symbol is about twice the frequency of preceding symbols. The even frequency distribution of symbols makes the statistical attack more difficult. So the simple base62 encryption method is more

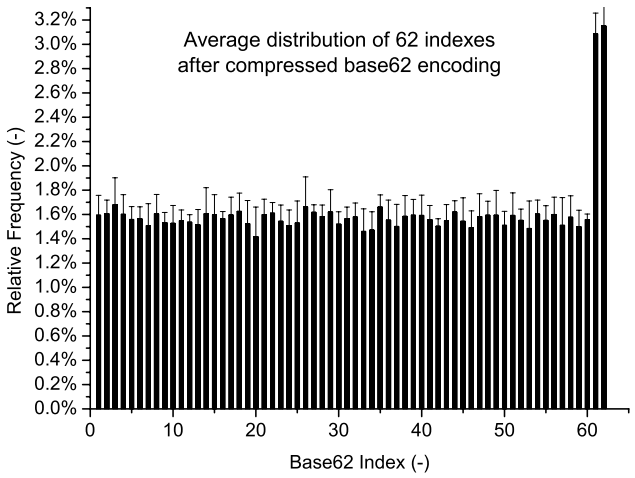

Fig. 4. Mean symbol frequency and standard deviation of 11 compressed base62-encoded Canterbury corpus files. Sixty-two symbols occur with almost the same frequency except the last two symbols.

secure when applied along with the preceding compressing steps.

Though the frequency distribution of symbols after compression steps is almost even, the encryption method contains only one substitution box and is practically too simple to become highly secure. But it is fast and simple, and suitable for applications that only need low security. And though the substitution box has 62! possible combinations, crackers only need to try small portion of them to crack part data.

We tested the compressed base62 encoding on the Canterbury corpus, and the results are presented in Table I. The compression rate is given in bits per character. The results of modern most-used compressors are also presented. All the results of bzip, bzip2, and gzip are obtained using their default settings. The LZJU90 encoder adopted is the sample implementation in the RFC 1505 [7]. Because base62 only use 62 alphanumeric characters to represent the compressed message, the compression rate of compressed base62 is lower

TABLE I
COMPRESSION (IN BITS PER CHARACTER) OF THE CANTERBURY CORPUS USING COMPRESSED BASE62 ENCODING.

File	base62	bzip	bzip2	gzip	LZJU90 [7]
alice29.txt	3.03	2.25	2.27	2.86	5.35
ptt5	1.04	0.77	0.78	0.88	1.76
fields.c	2.81	2.09	2.18	2.25	4.04
kennedy.xls	1.23	0.92	1.01	1.61	3.22
sum	3.52	2.61	2.7	2.7	5.11
lcet10.txt	2.69	2	2.02	2.72	5.13
plrabn12.txt	3.23	2.39	2.42	3.24	6.04
cp.html	3.29	2.44	2.48	2.6	4.51
grammar.lsp	3.43	2.55	2.79	2.65	4.57
xargs.1	4.23	3.13	3.33	3.31	5.48
asyoulik.txt	3.39	2.51	2.53	3.13	5.62
Average	2.90	2.15	2.23	2.54	4.62
Standard deviation	0.96	0.71	0.75	0.73	1.23

than that of bzip, bzip2, and gzip, which all make use of all 256 binary characters. But the advantage of base62 encoding is also obvious. First, it incorporates the compressing with alphanumeric encoding, so can be used in applications that can't use non-alphanumeric characters, such as URL and MIME. Second, its average compression rate (2.90 bpc) is much better than base64 (10.67 bpc) and LZJU90 (4.62 bpc), which both provide both compression and representation in a text format. Third, the base62 encoding provides a simple encryption mechanism and is suitable for normal use. It is also worth to note the compression rate of the data compression method presented here. If take the $\frac{10.779}{8} \approx 1.347$ times increase of bpc caused by base62 encoding into account, the compression method alone, which is based on BWT, MTF, ZLE, and range encoding, can achieve the average bpc of $\frac{2.90}{1.347} \approx 2.15$ on the Canterbury corpus. It is almost the same as bzip (2.15 bpc), which uses BWT, MTF, ZLE, and arithmetic encoding. The compression rate is better than that of bzip2 (2.23 bpc) and gzip (2.54 bpc). Bzip2 uses BWT, MTF, ZLE and Huffman encoding. Gzip uses DEFLATE algorithm [8], which is based on a variation of LZ77 combined with Huffman codes. So, it demonstrated that BWT, MTF, ZLE, and range encoding can be used together for efficient compression of normal data. It can achieve similar performance as arithmetic encoding but is free from patent encumbrance.

IV. Conclusion

This paper presented a secure compressed base62 encoding method. Burrows-Wheeler transform is used along with move-to-front transform, zero length encoding, and range encoding, to compress the original data. The base62 encoding only adopt 62 alphanumeric characters and a simple substitution encryption is incorporated. The base62 encoding can be used for efficient compressed representation of the original data using only alphanumeric characters. This paper analyzed the compression rate of base62 encoding and compared it with general purpose compressors, such as bzip, bzip2, and gzip. Also, this paper analyzed the security strength of base62 encoding.

If you only want to compress your normal data into binary format, the ordinary bzip2 and gzip compressors can get smaller compressed file size. However, if you want to compress your data and represent it using only 62 alphanumeric characters, compressed base62 encoding is the choice.

The fast encryption method presented here also provides simple security for the base62 encoding. It can be used in applications when the security is not strict and critical. The potential uses of base62 encoding include:

1) URL encoding: Base62 encoding is useful when fairly lengthy identifying information is used in the HTTP environment. It can be used to encode a relatively large data (e.g. 128-bit UUIDs) into an alphanumeric string for use as an HTTP parameter in HTTP forms or HTTP GET URLs. Base62 is a convenient encoding to represent them in not only a compact way, but also in a relatively unreadable way.
2) SMTP/MIME: Base62 encoding can be used as an efficient binary-to-text encoding scheme for the MIME (Multipurpose Internet Mail Extensions) specification. It is useful for sending other non-ASCII data through email.
3) Embed binary data in ASCII files: Base62 encoding can be used to embed binary data in ASCII files, such as the widely-used XML files. Since base62 is a secure, compressed encoding, it can help decreasing the final ASCII file size and integrating basic security.

References

[1] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC 1738 (Proposed Standard), Dec. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1738.txt

[2] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 4648 (Proposed Standard), Oct. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4648.txt

[3] J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421 (Historic), Feb. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1421.txt

[4] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," RFC 2045 (Draft Standard), Nov. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2045.txt

[5] K. He, S. Dong, L. Zheng, and L. Tang, "Service-oriented grid computation for large-scale parameter estimation in complex environmental modeling," in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2006, pp. 741–745.

[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[7] A. Costanzo, D. Robinson, and R. Ullmann, "Encoding Header Field for Internet Messages," RFC 1505 (Experimental), Aug. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1505.txt

[8] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951 (Informational), May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[9] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[10] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Research Report 124, Digital Systems Research Center, Tech. Rep., 1994.

[11] A. Moffat, R. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, July 1998.

[12] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proceedings of the Institute of Electronics and Radio Engineers*, vol. 40, pp. 1098–1101, 1952.

[13] G. N. N. Martin, "Range encoding: an algorithm for removing redundancy from a digitized message," Video and Data Recording Conference, Southampton, July 1979.

[14] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[15] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov, "Modifications of the burrows and wheeler data compression algorithm," in *DCC '99: Proceedings of the Conference on Data Compression*. Washington, DC, USA: IEEE Computer Society, 1999, p. 188.

[16] B. Balkenhol and Y. Shtarkov, "One attempt of a compression algorithm using the bwt," Preprint 99-133, SFB343 : Discrete Structures in Mathematics, Falculty of Mathematics, University of Bielefeld, 1999. [Online]. Available: http://www.math.uni-bielefeld.de/sfb343/preprints/pr99133.ps.gz

[17] R. Fisher and F. Yates, *Statistical Tables for Biological, Agricultural and Medical Research*. Edinburgh: Oliver and Boyd, 1938.

[18] R. Durstenfeld, "Algorithm 235: Random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.