Unicode Compression: Does Size Really Matter? TR CS-2002-11

Steve Atkin
IBM Globalization Center of Competency
International Business Machines
Austin, Texas USA 78758
atkin@us.ibm.com

Ryan Stansifer
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901
ryan@cs.fit.edu

July 2003

Abstract

The Unicode standard provides several algorithms, techniques, and strategies for assigning, transmitting, and compressing Unicode characters. These techniques allow Unicode data to be represented in a concise format in several contexts. In this paper we examine several techniques and strategies for compressing Unicode data using the programs `gzip` and `bzip`. Unicode compression algorithms known as SCSU and BOCU are also examined. As far as size is concerned, algorithms designed specifically for Unicode may not be necessary.

1 Introduction

Characters these days are more than one 8-bit byte. Hence, many are concerned about the space text files use, even in an age of cheap storage. Will storing and transmitting Unicode [18] take a lot more space? In this paper we ask how compression affects Unicode and how Unicode affects compression.

Unicode is used to encode natural-language text as opposed to programs or binary data. Just what is natural-language text? The question seems simple, yet there are complications. In the information age we are accustomed to discretization of all kinds: music with, for instance, MP3; and pictures with, for instance, JPG. Also, a vast amount of text is stored and transmitted digitally. Yet discretizing text is not generally considered much of a problem. This may be because the English language, western society, and computer technology all evolved relatively smoothly together. However, diverse languages, writing systems, and encodings of other places belie this sanguinity. Practices such as font-specific encodings, using pictures of text, incomplete and changing standards, all suggest that text-processing practices need attention. In the future, natural-language text may not be represented that same as it is today. Digital text may be built around different abstractions rather than simple code points.

One obvious strategy for representing natural-language is to use Unicode. But just what is Unicode? Is it data? Some characters have the same appearance but different meaning:

```
U+212B ANGSTROM SIGN
U+00C5 LATIN CAPITAL LETTER A WITH RING
```

Is it a collection of glyphs? Some code points represent different forms of the same data.

```
U+0671 ARABIC LETTER ALEF WASLA
U+FB50 ARABIC LETTER ALEF WASLA ISOLATED FORM
```

Is it a mechanism for interchange? The full-width characters come from Shift-JIS.

```
U+FF21 FULLWIDTH LATIN CAPITAL LETTER A
```

Is it mark-up? As, for instance, suggested by the following two code points:

```
U+200D Zero Width Joiner
U+2029 Paragraph Separator
```

Is it a compression scheme? UTF-8 [21] is often used to save space in storing and transmitting Unicode characters.

At the heart of our study is the coded character set UCS-2, a fixed-width, 16 bit encoding of Unicode characters. We completely ignore surrogates. This makes software programming easier and follows the practice of the Java programming language [1]. In Java, Unicode as USC-2 is used to interpret the computer bits representing characters internally (by which we mean inside the software application) and UTF-8 is often used externally (by which we mean storing and exchanging text).

An encoding scheme and a compression algorithm are similar in that they both pick a representation for the symbols. In this paper we explore this similarity. With vast computing power and clever compression algorithms available today, are encoding schemes necessary? Can each application be free to pick the representation of code points? Are there some representations that lend themselves better to compression? To address these questions we tried many different ways of compressing Unicode natural-language text.

Our experiments in compression do not hold all the answers to these questions. But we have an extensive study of the size of different representations. The complete results of our experiments as well as the corpus of text can be found at the WWW site:

`http://www.cs.fit.edu/˜ryan/compress`

2 Corpus

Collecting plain text is much more difficult than we expected. At first the Internet appears to be a boundless source of text. But, we avoided "marked-up" text like HTML documents. Perhaps marked-up text is a different kind of document: a hybrid natural/unnatural document. On the other hand, maybe the mark-up can be safely ignored or stripped out in this context. In any case, only one sample in our corpus of text is an HTML document. It is worth pondering if all electronically-stored text will be "marked-up" in the future. This does have an impact on character repertoire and character codes; see [2].

The overwhelming majority of plain text we encountered is in English and other European languages. These samples were the easiest to collect. Project Gutenberg [14] provided five texts in English, Spanish, and German. Non-English text was harder for us to acquire. Many scholarly archives are not freely accessible over the Internet. And our efforts were constrained by our bounded time, persistence, and limited linguistic ability.

We have collected 15 large samples of natural-language plain text, two of which are multilingual. We also added 9 samples of artificial text, some randomly created. These texts compress differently because natural-language text generally has lots of extraneous information that does not affect the intrinsic content. The entire list is shown in Table 1.

Table 2 has more information about the corpus including the coded character set we think the file is in, the number of Unicode characters, how many different characters occur in the file, and the entropy of the file. Entropy H is a measure of the information content:

$$H = -\sum_{i=1}^{n} P_i \log_2 P_i \qquad \text{bits per symbol}$$

where P_i is the probability of occurrence of the i-th symbol. Files with high entropy will be more random and harder to compress.

Though not relevant for the purposes of compression, the interpretation of the text files is not as easy as we expected—especially for non-English text. What did the authors intend? Did they make spelling errors? Did the 17th century typesetter run out of the letter 'U' and use 'V' instead (as

Table 1: Texts Used in Experiments

work	file	code set	language	characters
Alice in Wonderland by Carroll	alice30.txt	ISO-8859-1	English	148,542
Hamlet by Shakespeare	hamlet.txt	ISO-8859-1	English	162,850
Ulysses by Joyce	ulysses.txt	ISO-8859-1	English	1,517,848
Don Quijote by Cervantes	quijote.txt	ISO-8859-1	Spanish	2,093,952
Cinq Semaines by Verne	cinq10.txt	ISO-8859-1	French	489,772
Faust I by Goethe	faust1.txt	ISO-8859-1	German	187,764
Writings by Neḿeth	nemeth6.hun	ISO-8859-2	Hungarian	118,695
Quran	quran.txt	ISO-8859-6	Arabic	516,342
Odyssey by Homer	odyssey.txt	ISO-8859-7	Greek	46,622
Anna Karenina by Tolstoy	annak.txt	KOI8-R	Russian	1,704,065
Malik Muhammad Jayasi	introduction.isc	ISCII	Hindi	381,306
Thai word list	th_18057.txt	TIS-620	Thai	135,450
Kim Van Kieu by Nguyen	kieu175.vscii	VISCII	Vietnamese	6,783
The Tale of the Bamboo Cutter	taketori.txt	EUC-JP	Japanese	27,268
Three Kingdoms by Guanzhong Luo	sanguo.txt	GBK	Chinese	635,632
"Provincial" by Kaplan	provincial.utf8	UTF-8	multi-lingual	6,980
"UTF-8" by Kuhn	kuhn.utf8	UTF-8	multi-lingual	7,224
Maribyrnong Library Home Page	maribyrnong.html	UTF-8	multi-lingual	7,607
SML source code for a functor	convert.sml	ISO-8859-1	program	26,179
Java source code for a class	LZW.java	ISO-8859-1	program	20,034
C source code for a library	regex.c	ISO-8859-1	program	171,188
all 'A's	aaaa.txt	ISO-8859-1	artificial	12,000
four different 'A's	aAaA.utf8	UTF-8	artificial	12,000
random Unicode characters	random.utf8	UTF-8	artificial	200,000
every Unicode character	sequence.utf8	UTF-8	artificial	49,257
random bytes	bytes.data	bytes	artificial	12,000

Table 2: Characteristics of the Corpus

file	code set	roundtrip/ repl char	total characters	distinct characters	entropy
alice30.txt	ISO-8859-1	yes; 0	152,089	72	4.568
hamlet.txt	ISO-8859-1	yes; 0	162,850	65	4.569
ulysses.txt	ISO-8859-1	yes; 0	1,517,848	79	4.820
quijote.txt	ISO-8859-1	yes; 0	2,093,952	91	4.381
cinq10.txt	ISO-8859-1	yes; 0	489,772	103	4.553
faust1.txt	ISO-8859-1	yes; 0	187,764	72	4.843
nemeth6.hun	ISO-8859-2	yes; 0	118,695	94	4.783
quran.txt	ISO-8859-6	no; 93	516,342	61	4.622
odyssey.txt	ISO-8859-7	yes; 0	46,622	74	4.837
annak.txt	KOI8-R	yes; 0	1,704,065	136	4.734
intrduction.isc	ISCII	yes; 0	381,306	109	4.810
th_18057.txt	TIS620	no; 100	135,450	105	5.000
kieu175.vscii	VISCII	yes; 0	6,783	124	4.833
taketori.txt	EUC-JP	yes; 0	27,268	740	6.625
sanguo.txt	GBK	no; 22	635,632	3,921	8.918
provincial.utf8	UTF-8	yes; 0	6,977	613	6.305
kuhn.utf8	UTF-8	yes; 1	7,224	680	6.891
maribyrnong.html	UTF-8	yes; 0	7,607	247	5.505
convert.sml	ISO-8859-1	yes; 0	20,034	89	4.645
LZW.java	ISO-8859-1	yes; 0	20,034	88	4.731
regex.c	ISO-8859-1	yes; 0	171,188	97	4.762
aaaa.txt	ISO-8859-1	yes; 0	12,000	1	0.000
aAaA.utf8	UTF-8	yes; 0	12,000	4	2.000
random.utf8	UTF-8	yes; 4	200,000	48,459	15.402
sequence.utf8	UTF-8	yes; 1	49,257	49,257	15.588
bytes.data			12,000	256	7.984

sometimes occurred in the printing of Shakespeare's plays). Does the data represent the author's intention or the printed page? Transcribers had to pick characters to represent the work. Maybe there was no character available (discretization error) in the character set. For example, in *Alice in Wonderland*, we see the common practice of using the apostrophe and the grave accent for single quotation marks, since single quotation marks are unavailable in ISO-8859-1. Encoding the work in Unicode might be considered more accurate as all the characters in question are part of the Unicode repertoire:

```
U+0027  APOSTROPHE
U+0060  GRAVE ACCENT
U+2018  LEFT SINGLE QUOTATION MARK
U+2019  RIGHT SINGLE QUOTATION MARK
```

The size of the character set and the character encoding scheme have an impact on the meaning of the data.

3 Character Codes

Just a few samples were collected directly in Unicode (specifically UTF-8). The rest were in various 8-bit code sets and a few in multi-byte code sets. Naturally, much of it was encoded (apparently) in ISO-8859-1, commonly known as Latin-1.

Most text samples were converted into Unicode characters by using Java's internal character conversion logic. The following code fragment illustrates how it was done:

```
final String file_name = "alice30.txt";
final String encoding = "ISO-8859-1";
final BufferedReader in=new BufferedReader(new InputStreamReader(
    new FileInputStream (file_name, encoding));
for (;;) {
   final int unicode_char = in.read();
   if (unicode_char == -1) break;
}
in.close();
```

Another possibility for such conversion is the GNU software recode [13]. It was used for the Vietnamese character codes VISCII and VIQR [10], which are not predefined in the Java system of code-point converters. (Possible errors in UTF-8 discouraged us from using recode uniformly for all conversions.)

Among the other coded character sets used in the corpus are KOI8-R [6], GBK (simplified Chinese) [12], and ISO-8859-2. The complete list can be found in Table 2.

In several cases we are unable to convert back to the original data from the Unicode representation. We indicate this in the column labeled "round-trip" in Table 2. The original may have contained unassigned code points, or may be relying on a different version of the encoding standard. In these cases the conversion software may have inserted the replacement character; we take a replacement character as evidence of some sort of potential problem in the natural language text. For example, in sanguo.txt we find the 3,635th character is the replacement character. Here is a portion of that file, readers may draw their own conclusions.

```
...
3,631  U+6709   CJK UNIFIED IDEOGRAPH-6709    YOU3
3,632  U+591A   CJK UNIFIED IDEOGRAPH-591A    DUO1
3,633  U+5927   CJK UNIFIED IDEOGRAPH-5927    DA4
3,634  U+5173   CJK UNIFIED IDEOGRAPH-5173    GUAN1
3,635  U+FFFD   REPLACEMENT CHARACTER
3,636  U+FF0C   FULLWIDTH COMMA
...
6,118  U+4E00   CJK UNIFIED IDEOGRAPH-4E00
6,119  U+53E5   CJK UNIFIED IDEOGRAPH-53E5    JU4
6,120  U+FFFD   REPLACEMENT CHARACTER
6,121  U+7EB9   CJK UNIFIED IDEOGRAPH-7EB9    WEN2
6,122  U+305B   HIRAGANA LETTER SE
6,123  U+554A   CJK UNIFIED IDEOGRAPH-554A    A5 QIANG1
6,124  U+FFFD   REPLACEMENT CHARACTER
6,125  U+000A   LINE FEED
```

Also, the file th_18057.txt appears corrupted from about character 120,825. And, in the UTF-8 file by Markus Kuhn we find one occurrence of the replacement character. We have no way of knowing whether that was intentional.

Yet another problem of interpretation is raised with the Arabic text. The Arabic language is written in the right-to-left direction. When the text is rendered using the Unicode bi-directional algorithm, as, say by the Java widget `JTextPane`, the result is not what was intended. Mirroring of brackets is done "incorrectly" (not in the way intended by the authors). The difficulties of bi-directional text have been the subject of another investigation [3].

Other more subtle sources of misinterpretation are possible. Often the character encoding used is not given. And even if hints are given it is hard to be sure. These encoding standards are evolving or are confused by common alternative practice or mis-practice. Finally, it is possible that the converters have bugs or that its developers are confused by the same problems confronting the authors of the text.

In conclusion, a portion of the non-English corpus may not be "meaningful" as revealed by even a superficial analysis of the data. This raises troubling questions about the fidelity of the data representing natural language in scripts other than Latin.

4 Character Encodings

Fundamentally the underlying data is a collection of Unicode characters. But this information can be represented in different ways. These different representations have different advantages and disadvantages. We examine a number of representations, and see which ones take up the least space and which ones compress the best.

We have chosen a number of different formats to see if any significant differences emerge. Table 3 lists the different intermediate formats we used. The first group of formats are “plain” Unicode—just the 16-bit code point. Even so there are some variations that may make a difference. We might pad out the 16 bits to 32 bits. This format we call UCS-4. Also the byte order might affect compression, so we try both little-endian and big-endian: UCS-2 LE and UCS-2 BE, respectively.

Table 3: Different Intermediate Formats

UCS-4	Unicode four octet representation (big endian)
UCS-2 BE	Unicode two octet representation (big endian)
UCS-2 LE	Unicode two octet representation (little endian)
UTF-8	1, 2, or 3 octet representation of 16 bits
UTF-7	1, 2, or 3 octet representation of 16 bits
base 64	base 64 encoding of UCS-2
hex	hexadecimal digits of UCS-2
entities	mark-up language entities
names	full Unicode character names
diff	differences encoded in UTF-8
SCSU	Standard Compression Scheme for Unicode
BOCU	Binary-Ordered Compression for Unicode

Since many of the leading bytes in the 16 bits are zero, other formats have been devised to save space. UTF-8 encodes the 16 bits in 1, 2, or 3 octets. UTF-7 does the same, but uses only the lower 7 bits of each octet as in US-ASCII. Similarly base-64 uses only 64 common, graphical symbols encoded in US-ASCII, but with a fixed-width format. Every three Unicode characters is encoded in 8 symbols.

The next group of formats are in a category we might call “quoted printable.” There really is a quoted-printable format [4] in which every octet using more than 7 bits is replaced by the digits representing the code point. This requires two characters plus an “escape” character (the equals sign) to encode one byte. Obviously any bit pattern can be encoded this way. We did not use the quoted-printable format because we used several very similar ones. One, we have called “hex,” is an extremely simple one that uses the four hexadecimal digits (in US-ASCII) for every Unicode character. Another, we have called “entities,” uses the HTML entity approach and contains the decimal digits in this format: `&#`*dddd*`;`. The format we have called "names" uses the formal Unicode names for each character terminated by a line-feed character. Naturally these approaches expand the number of bytes needed to store the files considerably. But, as we will see, they compress nearly as well as the other formats.

These formats are perhaps related to a type of encoding that we have not yet investigated. We can imagine a human-readable, multi-character encoding. This could be like RFC1345 [16] that proposes a mnemonic system for some (not all) Unicode characters. Naturally, the challenge is including the ideographic script systems. Perhaps a general system of input could be used as the encoding. Some systematic input methods for alphabetic scripts have been studied [19].

The final group of formats have compression as part of their motivation. They take advantage of the fact that mono-lingual text does not require all of Unicode, so using 16 bits for every character is wasteful. This approach resembles UTF-8, except that UTF-8 favors US-ASCII—everything else must take more than one byte even if it is in an alphabetic script. We have created and implemented a format we call "diff." It is a simple-minded, stateful encoding of Unicode that puts in the higher-order bits of the USC-2 only when they change. The Standard Compression Scheme for Unicode (SCSU) [20] is a more sophisticated character encoding scheme that does the same. More recently another approach has been developed that in addition preserves the order of the characters, Binary-Ordered Compression for Unicode (BOCU) [15]. BOCU is a stateful, muti-byte, encoding of Unicode. The control codes including NUL, CR and LF are encoded with the same byte values as in US-ASCII.

Some formats are easier to visualize than others. We give an example of encoding a particular sequence of five Unicode characters in the list below. We use the first five characters of file `aAaA.utf8`. The format "diff" removes the bias toward US-ASCII, but all the spaces and line-feed characters in the data require the format to make large jumps back to where all the higher-order bits are zero.

		Unicode Name	BE	LE	UTF-8
1	U+0410	CYRILLIC CAPITAL LETTER A	0410	1004	d0 90
2	U+0041	LATIN CAPITAL LETTER A	0041	4100	41
3	U+0041	LATIN CAPITAL LETTER A	0041	4100	41
4	U+0391	GREEK CAPITAL LETTER ALPHA	0391	9103	ce 91
5	U+FF21	FULLWIDTH LATIN CAPITAL LETTER A	FF21	21FF	ef bc a1

		entities	diff	base 64
1	U+0410	А	0410=d0 90	041000=BBAA
2	U+0041	A	FC41=ef b1 81	410041=QQBB
3	U+0041	A	0041=41	
4	U+0391	Α	0391=ce 91	0391FF=A5H/
5	U+FF21	Ａ	FBA1=ef ae a1	21****=IQ==

5 Compression

The compression software we used was the GNU `gzip` and `bzip2` programs.

The `gzip` compressor used in our experiments is in the general class of LZ77 (Ziv and Lempel) lossless compressors[22]. The basic strategy used by LZ77 compressors is to replace substrings (phrases) with pointers to the place where they occurred before in the text, yielding a tuple containing a phrase position and a phrase length. This represents an adaptive approach where the prior text is the codebook itself. Decompression is relatively simple and fast. For each tuple encountered, go to that phrase position and write the number of bytes indicated by the phrase length.

The `bzip2` compressor used in our experiments is in the general class of block-sorting compressors. In general, block-sorters first reorder a section (block) of text using a sorting algorithm prior to any compression. Sorting the block transforms the text into a representation that lends itself to efficient compression. In the case of `bzip2` the Burrows-Wheeler Transform [5] is used to sort the block of text. Once the block has been sorted traditional compression techniques are then applied, such as run-length encoding.

All experiments were conducted on a Sun Microsystems Ultra-5 computer running Solaris. The compressors `gzip`, version 1.24 [8], and `bzip2`, version 1.0.2 [17], were invoked with their default parameters (neither their smallest nor largest block sizes). Both are designed to take sequences of octets (not characters) as input. Since all data must necessarily be binary, this does not exclude any input. However, this raises the interesting possibility of tuning these compressors for 16 bit or Unicode characters and seeing if that makes a difference. This possibility was examined in a study by Fenwick and Brierley [7]. They concluded an LZU compressor for 16 bits offers some improvement.

As our results substantiate, the compressed files of `bzip2` are generally smaller than `gzip`. Of the 320 files compressed (these includes all the various formats), `gzip` had smaller output in only 27 cases; and in these cases the difference is small. On the other hand, `bzip2` is generally held to be slower than `gzip`, but we made no such measurements ourselves. Here we consider only space and we ignore time. Both are important, but timing results can be difficult to interpret.

6 Results

Each file of the corpus was converted to Unicode characters and put in each of the twelve formats discussed previously. All the files were then compressed with `gzip` and `bzip2`.

We consider first the file ulysses.txt. *Ulysses*, an unusual work of English literature, exhibits behavior representative of the other natural-language texts encoded in ISO-8859-1 as far as our experiments go. The data we collected for this case is presented in Table 4.

For text encoded in ISO-8859-1, several of the other formats use the identical encoding of the characters. These are marked with an equals sign in Table 4, and the row of the table is omitted. In

Table 4: Results for ulysses.txt

				gzip		bzip2	
		octets	bits/char	octets	bits/char	octets	bits/char
text	=	**1,517,848**	8.000	**658,347**	3.470	**516,295**	2.721
UCS-4		6,071,392	32.000	1,011,501	5.331	548,646	2.892
UCS-2 BE		3,035,696	16.000	795,198	4.191	532,984	2.809
UCS-2 LE		3,035,696	16.000	795,200	4.191	533,307	2.811
UTF-8	=						
UTF-7	≈	1,517,854	7.000	658,350	3.470	516,681	2.723
base 64		2,023,800	8.000	866,530	4.567	595,115	3.137
hex		6,071,392	16.000	1,019,387	5.373	560,416	2.954
entities	≈	1,517,860	8.000	658,355	3.470	516,797	2.724
names		27,512,696	90.631	1,383,527	7.292	632,720	3.335
diff	=						
SCSU	=						
BOCU		1,517,848	8.000	658,383	3.470	516,306	2.721

the case of ulysses.txt two other encodings are very similar, but not identical. These encodings are marked with the ≈ sign. UTF-7 differs only because of the two occurrences of the character

```
U+002B  PLUS SIGN
```

(the least frequently occurring character in the file ulysses.txt) which is used as a meta character in UTF-7 and must itself be encoded. And the file of mark-up language entities differs from the original only because of the three occurrences of

```
U+0026  AMPERSAND
```

which serves as the escape character for HTML entities. Otherwise these formats would also be identical to the original.

The original text file takes up the least space and the Unicode character names take up the most. The names require a whopping 90 bits per Unicode character. This figure is so low only because we charged each octet with just 5 bits because the character repertoire used in the names—roughly the upper-case ASCII letters—will fit in 5 bits. (Similarly UTF-7 is charged 7 bits, base-64 6 bits, etc.) Furthermore, the original text file is the most compressible format. Both `gzip` and `bzip2` are able to compress the text file more than any other format.

The case of the Vietnamese text is interesting. The data we collected for this case is presented in Table 5. This is the only case that we have two text encodings. The first is VISCII which

Table 5: Results for kieu175.viscii

			gzip		bzip2	
	octets	bits/char	octets	bits/char	octets	bits/char
VISCII	**6,783**	8.000	**3,434**	4.050	2,933	3.459
VIQR	8,383	9.887	3,586	4.229	**2,896**	3.416
UCS-4	27,132	32.000	4,968	5.859	2,968	3.501
UCS-2 BE	13,566	16.000	4,190	4.942	2,966	3.498
UCS-2 LE	13,566	16.000	4,194	4.946	3,029	3.572
UTF-8	8,478	9.999	3,752	4.425	2,940	3.467
UTF-7	11,008	11.360	3,927	4.632	3,080	3.633
base 64	18,088	16.000	5,388	6.355	3,898	4.597
hex	27,132	16.000	4,700	5.543	3,017	3.558
entities	11,558	13.632	4,000	4.718	3,024	3.567
names	126,429	93.195	6,398	7.546	3,377	3.983
diff	10,507	12.392	4,120	4.859	3,166	3.734
SCSU	7,795	9.194	3,687	4.349	3,031	3.575
BOCU	8,903	10.500	4,112	4.850	3,349	3.950

is an 8-bit encoding that does not preserve the bottom half (0x00-0x7F) for US-ASCII like the ISO-8859 standards. There are just too many Vietnamese characters that need to be represented. The second text encoding embeds Vietnamese into US-ASCII. This encoding is called Vietnamese Quoted Readable (VIQR); the first verse of kieu175.viqr appears below:

```
1.     Tra(m na(m trong co~i ngu+o+‘i ta,
       Chu+~ ta‘i chu+~ me^.nh khe’o la‘ ghe’t nhau\.
       Tra?i qua mo^.t cuo^.c be^? da^u,
       Nhu+~ng ddie^‘u tro^ng tha^’y ma‘ ddau ddo+’n lo‘ng.
```

US-ASCII symbols are used as conjoining diacritics. The backslash character is used to mean the next character is *not* to be interpreted as a diacritic mark. This encoding is apparently fairly readable. Despite its multi-byte nature it compresses well; in fact, it is the most compressible of all the formats. See Table 5.

Table 6 summarizes our results. All the formats compressed to about the same size by `bzip2`. But some trends are evident. In the alphabetic scripts compressing the original text almost always results in a smaller file than first converting to UTF-8 or some other format and then compressing. Over all these texts, the compressed files require an average of 2.714 bits per characters. If the file is converted to UTF-8 first, for example, and then compressed an average of 2.750 bits per characters is required.

Table 6: Comparison of `bzip2` Compressibility

file	text	UCS-2 BE	UTF-8	hex	SCSU	BOCU
alice30.txt	2.325	2.326	2.325	2.384	2.325	2.321
hamlet.txt	2.642	2.642	2.642	2.715	2.642	2.642
ulysses.txt	2.721	2.809	2.721	2.954	2.721	2.721
cinq10.txt	2.317	2.386	2.318	2.602	2.317	2.345
quijote.txt	2.266	2.354	2.271	2.528	2.266	2.292
faust1.txt	2.601	2.598	2.601	2.667	2.601	2.620
nemeth6.hun	3.013	3.015	3.016	3.102	3.019	3.102
quran.txt	2.135	2.183	2.136	2.356	2.144	2.156
odyssey.txt	3.117	3.122	3.113	3.203	3.120	3.167
annak.txt	2.456	2.575	2.565	2.791	2.454	2.482
introduction.isc	2.621	2.623	2.668	2.827	2.621	2.645
th_18057.txt	3.615	3.617	3.626	3.738	3.611	3.621
kieu175.viscii	3.459	3.498	3.467	3.558	3.575	3.950
ALPHABETIC	2.714	2.750	2.728	2.879	2.724	2.774
taketori.txt	5.133	5.279	5.170	5.499	6.084	5.720
sanguo.txt	7.543	7.531	7.554	8.163	7.563	7.862
provincial.utf8		5.796	5.717	5.976	5.542	5.684
kuhn.utf8		6.003	5.854	6.198	5.888	6.152
IDEOGRAPHIC/MULTI		6.152	6.074	6.459	6.269	6.355
maribyrnong.html		3.488	3.432	3.570	3.495	3.534
convert.sml	2.306	2.282	2.306	2.404	2.306	2.305
LZW.java	2.102	2.086	2.102	2.183	2.102	2.106
regex.c	1.734	1.726	1.734	1.799	1.737	1.734
aaaa.txt	0.031	0.030	0.031	0.033	0.031	0.031
aAaA.utf8		2.213	2.226	2.519	2.712	2.325
random.utf8		16.030	16.086	16.259	16.739	18.261
sequence.utf8		4.533	5.900	5.398	3.294	2.444
bytes.data	8.325	8.333	8.451	8.522	8.339	9.248
ARTIFICIAL		4.525	4.696	4.743	4.528	4.665
ALL		3.888	3.924	4.075	3.894	3.980

Considering the files with a much larger range of characters, compressing the UCS-2 representation almost always results in the smallest output file. Whether it is little-endian or big-endian does not matter. The really simple hex format and the Unicode name format (not shown in Table 6) do not ever yield the most savings; but they are not always the least compressible of the formats.

Using BOCU and SCSU saves space compared to uncompressed UTF-8, but compressing the original text with `gzip` or `bzip2` saves much more space. Even compressing the BOCU and SCSU files with `bzip2` rarely saves as much space. See Tables 4 and 5. The other cases are similar; though the higher the entropy, the lower the savings.

We conclude that the program `bzip2` compresses all the examples well. No matter how diverse the formats of the input, the resulting output files are about the same size. This suggests that the information content–the Unicode characters–is being efficiently represented by `bzip2`.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Progamming Language*. Java Series. Addison-Wesley, Reading, Massachusetts, third edition, 2000.

[2] Steve Atkin. *A Framework for Multilingual Information Processing*. PhD thesis, Florida Institute of Technology, 2001.

[3] Steve Atkin and Ryan Stansifer. Implementations of bidirectional reordering algorithms. In Unicode Consortium, editor, *Eighteenth International Unicode Conference (IUC18) Unicode and the Web: the Global Connection, April 24–27, 2001, Hong Kong*, San Jose, California, 2001. The Unicode Consortium.

[4] Nathaniel S. Borenstein and Ned Freed. RFC 1521: MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies, September 1993.

[5] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Systems Research Center, May 1994.

[6] A. Chernov. RFC 1489: Registration of a Cyrillic character set, July 1993.

[7] Peter Fenwick and Simon Brierley. Compression of Unicode files. In James A. Storer and Martin Cohn, editors, *DCC '98: Data Compression Conference, March 30–April 1, 1998, Snowbird, Utah*, Silver Spring, Maryland, 1998. IEEE Computer Society Press.

[8] Jean-loup Gailly and Mark Adler. Gzip, 1.2.4 (18 aug 93). `http://www.gzip.org/`.

[9] David Goldsmith and Mark Davis. RFC 2152: UTF-7: A mail-safe transformation format of Unicode, May 1997. `http://www.faqs.org/rfcs/rfc2152.html`.

[10] Vietnamese Standardization Working Group. RFC 1456: Conventions for encoding the Vietnamese language. VISCII: VIetnamese Standard Code for Information Interchange. VIQR: VIetnamese Quoted-Readable specification, May 1993.

[11] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, September 1987.

[12] Ken Lunde. *CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing*. O'Reilly & Associates, Inc., Sebastopol, California, 1999.

[13] Franois Pinard. Recode. `http://www.iro.umontreal.ca/contrib/recode/HTML/`.

[14] Project Gutenberg. `http://www.promo.net/pg`.

[15] Markus Scherer and Mark Davis. Bocu-1, 2002. `http://oss.software.ibm.com/cvs/icu/~checkout~/icuhtml/design/conversion/bocu1/bocu1.html`.

[16] Keld Simonsen. RFC 1345: Character mnemonics and character sets, June 1992.

[17] Julian Steward. Bzip2, version 1.0.2, 30-dec-2001. `http://sources.redhat.com/bzip2/`.

[18] The Unicode Consortium. *The Unicode Standard, Version 3.0*. Addison-Wesley, Reading, Massachusetts, 2000.

[19] Uwe Waldmann. A new input technique for accented letters in alphabetical scripts. In Unicode Consortium, editor, *Twentieth International Unicode Conference (IUC20) Unicode and the Web: the Global Connection, January 28 – February 1, 2002, Washington, DC, USA*, San Jose, California, 2002. The Unicode Consortium. `http://www.mpi-sb.mpg.de/~uwe/paper/AccInput-bibl.html`.

[20] Misha Wolf, Ken Whistler, Charles Wicksteed, Mark Davis, and Asmus Freytag. A standard compression scheme for Unicode. Unicode Technical Standard 6, The Unicode Consortium, San Jose, California, May 2002.

[21] Franois Yergeau. RFC 2279: UTF-8, a transformation format of ISO 10646, January 1998.

[22] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23:337–343, May 1977.