This article appears in the Proceedings of the Science Information Management and Data Compression Workshop held at NASA/GSFC, Greenbelt, MD, U.S.A. on October 26-27, 1995.

Selecting a General-Purpose Data Compression Algorithm

Gary Jason Mathews

NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

Email: mathews@nssdc.gsfc.nasa.gov
Home Page: http://coney.gsfc.nasa.gov/~mathews/

Abstract

The National Space Science Data Center's Common Data Format (CDF) is capable of storing many types of data such as scalar data items, vectors, and multidimensional arrays of bytes, integers, or floating point values. However, regardless of the dimensionality and data type, the data break down into a sequence of bytes that can be fed into a data compression function to reduce the amount of data without losing data integrity and thus remaining fully reconstructible. Because of the diversity of data types and high performance speed requirements, a general-purpose, fast, simple data compression algorithm is required to incorporate data compression into CDF. The questions to ask are how to evaluate and compare compression algorithms, and what compression algorithm meets all requirements. The object of this paper is to address these questions and determine the most appropriate compression algorithm to use within the CDF data management package that would be applicable to other software packages with similar data compression needs.

Keywords: Data compression; Algorithm comparison; CDF

Introduction

Because of the large amounts of data collected by the National Aeronautics and Space Administration (NASA) and the scientific communities, data compression is an important consideration for archival and data management systems. The National Space Science Data Center (NSSDC) is evaluating various compression techniques for incorporation into the Common Data Format (CDF)[1], which is a data management package for storing, manipulating, and accessing multidimensional data sets.

Obviously, the best data compression algorithm would compress all data with the highest compression rate in the least amount of time, but no such algorithm exists. There is no best compression algorithm for everyone since the criteria for measurement are both data- and application-dependent. One user might want a high data compression ratio regardless of time, another fast compression with some sacrifice to the compression ratio, and others something in-between. There are trade-offs among time to compress and decompress the data, compression rates, memory requirements (hash tables, frequency counts, temporary buffers, etc.), and algorithmic complexity, so application developers must let their requirements determine what trade-offs are more or less important.

CDF provides a programming interface for applications whose goal is to provide fast access to the data; therefore, speed is a high priority for the desired compression algorithm. There is not much need for compression if most data cannot be compressed, so users need a general-purpose compression algorithm that compresses most data with a good compression ratio. The compression ratio is defined as the percentage of reduction achieved or as (1-compressed data size/uncompressed data size) x 100. A maximum of 100% compression is not possible unless the compressed size is zero, a situation only possible if the uncompressed data size is zero as well. Therefore, the upper limit is 100%, and the ratios will approach this limit. A negative compression ratio will result if the compressed data size is larger than the uncompressed data size, that is, the data have expanded with negative compression. There is no lower limit, since data can expand infinitely by recursively adding a single byte to the previous data ad infinitum. For all intensive purposes the lower limit can be set to zero, since if we fail to compress something, then we can leave it uncompressed with a compression ratio of zero. A good compression ratio is defined as a significant data reduction for all test data as a whole for a particular compression algorithm. Furthermore, CDF provides a library that is linked into all CDF applications, so also needed is a compression algorithm that does not dramatically increase the application size. Portability is another important aspect since both the CDF software and data sets are portable to DEC Alpha, DECstation, HP 9000, IBM PC, IBM RS-6000, Macintosh, NeXT, SGI, Sun, and VAX platforms. The hallmark of the CDF concept is its data set independence, so a compression implementation with minimum architecture- and machine-level dependencies is preferable to simplify porting to other platforms. In the context of the requirements classification, portability will be an implied requirement for a simple algorithm. Therefore, the following three factors will be used to compare the different compression algorithms and decide what algorithm is best for the users' needs:

- A fast algorithm that minimizes compression/decompression time.
- A good algorithm with a high compression rate.
- A simple algorithm with a small code size.

Data Characteristics

CDF can handle data organized conceptually as scalar and multidimensional with arrays of up to ten dimensions. The supported data types are those consistent with types available with C and FORTRAN compilers on most systems, which include 1-, 2-, and 4-byte signed and unsigned integers, 4-byte single-precision floating-point, 8-byte double-precision floating point, and 1-byte signed and unsigned character types. For example, there may be data defined as a single scalar 1-byte integer and another as a 512 x 512 x 16 array of double-precision floating point numbers. Although it is not very promising to compress a single scalar byte, compressing a large array or stream of bytes has significant results. Similarly, any data value with 2 or 3 bytes cannot be reduced because no reduction is possible. However, a 4-byte quantity can be reduced to possibly 3 bytes, so the minimum file size of 4 bytes is selected. Since most data stored in CDFs are typically under three dimensions, such as scalar data types, three-element vectors (e.g., x, y, z components), and two-dimensional images (e.g., 320 x 240), the size of the sample data files is arbitrarily chosen to range from 4 bytes to 144 Kbytes. Although larger files achieve higher compression ratios, the smaller data files better represent the typical size of data stored as components within a CDF.

To evaluate a general purpose compression algorithm, a representative collection of sample data to test the algorithms must be defined. As used by the *Dr. Dobb's Journal* (DDJ) Data Compression Contest[2], data samples were selected from graphics, text, executable, and sound categories. Although it is not very likely that MS-DOS executable programs will be stored in CDFs, these files have their own structure, and the goal is to find a general purpose compressor for all types of data, so all of the files used by the DDJ tests are included in the tests. In addition to these common file types, a subset of scientific data from the International Solar-Terrestrial Physics (ISTP) Key Parameter Data CD-ROM[3] was included since it represents the typical values and types of data that CDF actually stores, such as ISTP spacecraft orbit, attitude, magnetic field and particle measurements, and image data.

Program	Description
1 ahuff*	Adaptive Huffman coding
2 ar*	Haruhiki Okumura's archiver
3 arith*	Arithmetic coding
4 arith1*	Order-1 arithmetic coding
5 arj†	Robert Jung's ARJ, v2.41a
6 arth1e*	Order-1 arithmetic coding
7 cmp*	Linear conversion scheme for sound
8 compress†	UNIX compress uses LZW variant
9 cmp40-14†	Compress w/COMP40 option + 12-bit
10 comp40†	Compress with COMP40 option
11 comprs12†	Compress 12-bit
12 comprs14†	Compress 14-bit
13 comp_opt	Modified/optimized compress
14 copy	MS-DOS COPY command
15 dct*	Discrete cosine transformation
16 dogzip†	gzip-like LZ77 variant
17 doz†	UNIX-like LZW
18 gzip†	Gnu ZIP uses LZ77 variant[7]
19 huff†	Huffman coding
20 lha†	Harayasu Yoshizahi's LHA v2.13
21 lzari†	LZSS with adaptive arithmetic coding
22 lzhuf†	LZSS with adaptive Huffman coding
23 lzrw1	Ross Williams[5] LZ variant
24 lzrw3a	Ross Williams LZ variant
25 lzss†	Storer-Szymanski modified LZ77
26 lzss*	LZ77 with 12-bit sliding window
27 lzw12*	12-bit LZW compression
28 lzwcom†	Kent Williams LZW variant
29 pkarc†	PKARC v3.6
30 pkzip (default)†	PKZIP v2.04g LZ77 variant[7]
31 pkzip -ex†	PKZIP using maximal compression
32 rdc	Ross Data Compression uses run-length encoding[4]
33 snd*	Silence compression coding
34 zoo†	Rahul Dhesi's Zoo archiver, v2.1

* Source available on disks provided with the *Data Compression Book*[6].
† Source and/or executables available on SimTel anonymous ftp sites (e.g., oak.oakland.edu).

Table 1: Data Compression Programs Tested with CHURN

The compression programs evaluated are implementations for variations of the LZ77, LZ78, LZW, Huffman, run-length encoding, and arithmetic compression algorithms. Table 1 shows the programs that were included for the MS-DOS compression tests.

A compression utility program called CHURN, which accompanies Nelson's *Data Compression Book*[6], is capable of running compression and decompression programs on all data files in a specified disk volume for MS-DOS, measuring the elapsed compression and decompression times and verifying that the contents of the decompressed files match the original data files. CHURN is called with three arguments. The first argument is the drive letter and path name to recursively search for files to compress. The second parameter on the command line is the compression command. CHURN needs to compress the input file to a file called “TEST.CMP”. The compression command tells CHURN how to do this. CHURN will execute the compression command by passing the command line to DOS using the system() function call. It inserts the file name into the compression command by calling sprintf(), with the file name as an argument. This means that if the compression command has a %s symbol anywhere in it, the name of the input file should be substituted for it. Finally, the third argument on the command line should be the command CHURN needs to spawn to decompress TEST.CMP to TEST.OUT. An example of how this works would look like this:

```
CHURN  D:\DATA\  "LZSS-C %%s test.cmp"  "LZSS-D test.cmp test.out"
```

The double % symbols are present to defeat variable substitution under some command line interpreters such as 4DOS. A more complicated example testing PKZIP might look like this:

```
CHURN  D:\DATA\  "PKZIP.BAT %%s"  "PKUNZIP TEST.CMP"
```

where PKZIP.BAT has two lines that look like this:

```
COPY  %1  TEST.OUT
PKZIP  -M  TEST.CMP  TEST.OUT
```

CHURN creates a summary of compression results in a file called “CHURN.LOG”. This file can be used for further analysis by other programs. This file is reformatted and used to generate the graphic output in Figure 1, where the numbers correspond to those in Table 1. CHURN and the corresponding compression programs were tested on an IBM compatible 80486-33 with MS-DOS 6.20.

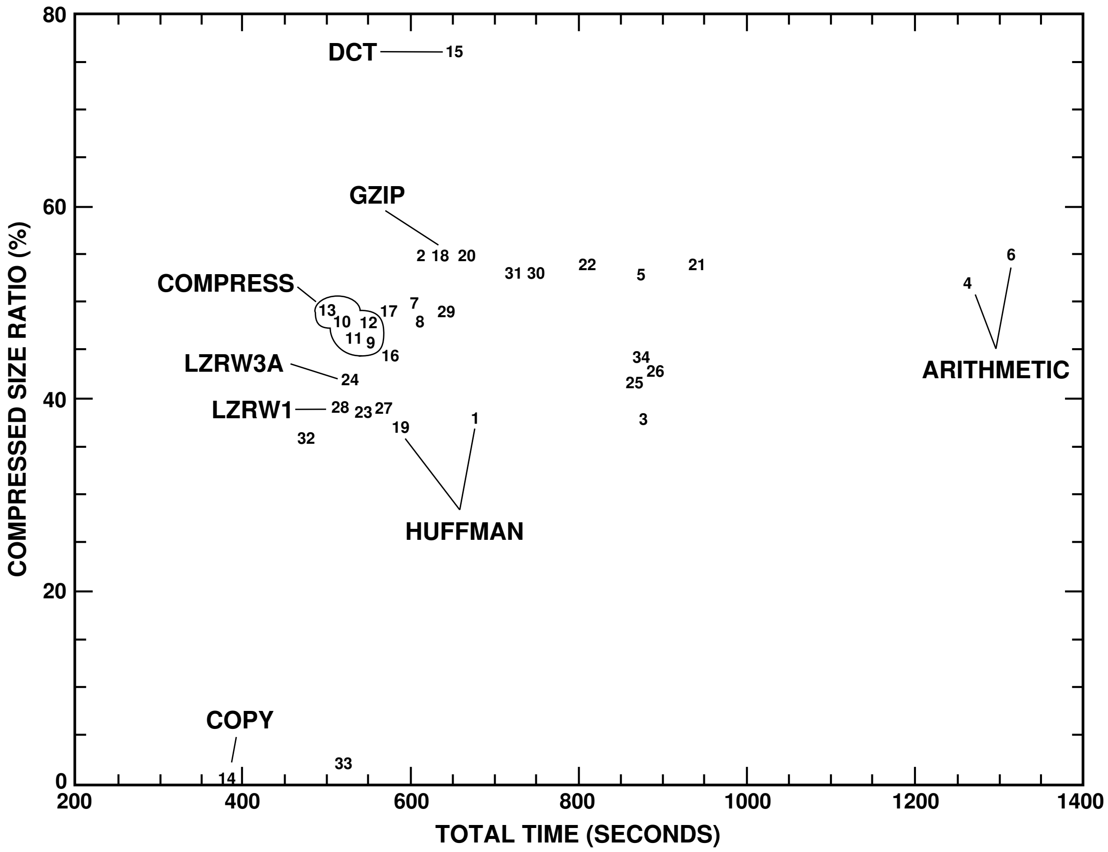

Figure 1. CHURN Compression Results for 30 Programs

The DOS COPY command, listed in Table 1 as item 14, measures the base-line time to copy a file byte-by-byte with no compression. No compression program should be able to read the entire input file, analyze the data, and compress the results in less time than a straightforward copy, but some compression programs actually approach this time with significant compression. Therefore, the overhead for compressing data is not going to impact performance if the faster compression algorithms are used. The DCT program appears to have the highest compression rate, but this is a lossy algorithm included in the test suite only for comparison since lossy algorithms do not meet the requirement for full reconstruction of the data. However, the COMPRESS, GZIP, LZRW1, and LZRW3A programs all achieve high compression rates with fast execution times.

Another test was conducted on six algorithms, listed below in Table 2, using the Computerized Reduction Using Selected Heuristics (CRUSH) data compression program. CRUSH is a portable, multi-method data compression utility that incorporates different algorithms into a common interface for VMS and UNIX systems. CRUSH takes a different approach than CHURN by linking different compression and decompression algorithms into one program rather than by calling external programs. CRUSH is a very flexible utility because users can either compress files using a specific compression algorithm or select automatic mode, which tries all algorithms and selects the method with the best compression. The automatic compression mode ensures that the data have the maximum compression given the available algorithms but at a high cost of time. The algorithms included in CRUSH were developed by Ian H. Witten, Radford M. Neal, and John G. Cleary (Department of Computer Science, University of Calgary); David M. Abrahamson (Trinity College, Dublin, Ireland); Ross N. Williams (Renaissance Software, Adelaide, Australia); Robert F. Rice (Jet Propulsion Laboratory); and Pen-Shu Yeh and Warner Miller (Goddard Space Flight Center). CRUSH is public domain, and the source code for CRUSH is available via anonymous ftp from dftnic.gsfc.nasa.gov (128.183.115.71) in the software/unix/crushv3 directory.

Method Name	**Algorithm Description**
ADAP	Adaptive dependency WNC method
LZC	Lempel/Ziv/Welch, ala UNIX compress
LZRW1	Fast Lempel-Ziv method[5]
LZRW3A	Fast Lempel-Ziv method
RICE	Rice machine
WNC	Witten/Neal/Cleary Arithmetic code

Table 2. Data Compression Algorithms Tested with CRUSH

These tests were run on a Sun 4 machine under SunOS 4.1. This machine, however, is not a dedicated machine and the elapsed times may have been slightly affected by other activity on the system, but the tests were run at off-hour times. CRUSH compresses each data file using each of the six algorithms. For tests of an individual data file, the best compression algorithm varied, but the overall results can be viewed in Figure 2. LZC has the highest compression ratio and one of the lowest times. The next best class of algorithms with a high best compression/time ratio is LZRW3A and LZRW1. WNC and ADAP both offer comparable compression ratios, and ADAP is even slightly better than LZC, but both algorithms take a much longer time. Overall, the RICE algorithm does not work well as a general purpose compressor, but it does work best on several individual files.

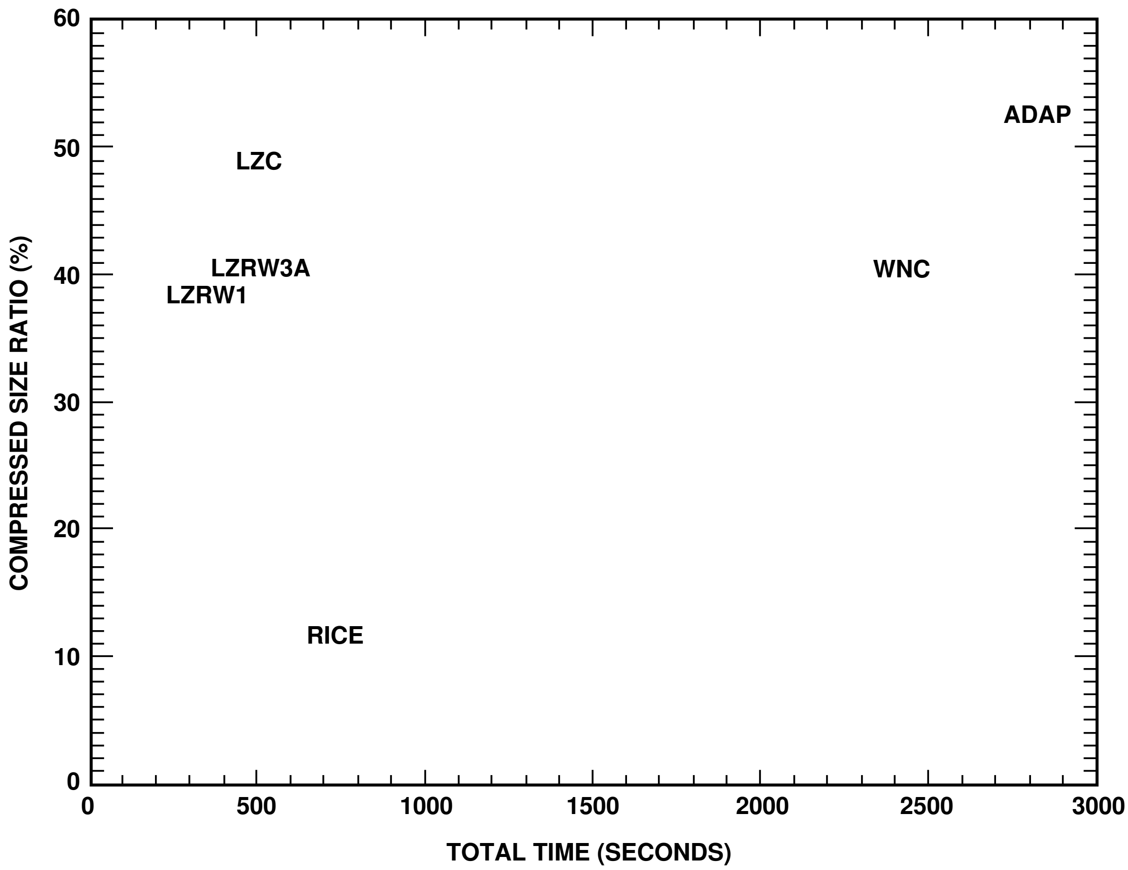

Figure 2. CRUSH Compression Results

The CRUSH program has a single interface with each compression algorithm having equivalent functionality, so the code complexity measurement can be estimated using the size of the corresponding object code files. The object code size is a rough approximation of complexity, but it gives some idea of how much is involved in the algorithm along with the other measurements. Since the compression programs tested on the IBM PC each have different functionality, error handling, and other bells and whistles, the object size and executable size are not very reliable measures of the algorithm's complexity, except where all algorithms are implemented with the same interface and equivalent error handling. For this reason, all complexity estimates will include only the algorithms available through the CRUSH interface, which has the equivalent interface and functionality among the six available algorithms.

The following table (Table 3) ranks each of the three measurements (compression ratio, speed, and complexity) from 1 to 4 (1 = least desirable and worst in its category; 4 = most desirable and best in its category). The compression ratio and speed (total elapsed time) ranks are extracted from the results of running CHURN with the test data and scaled appropriately. The overall score is a weighted sum (10 x compression rank + 10 x time rank + 5 x complexity rank) with a maximum of 100. The complexity weight is lower than the other two because it is the least reliable measurement. The weights can be adjusted appropriately if there is more or less priority on any of the three measurements.

Method	Compression Ratio	Speed	Complexity (Obj. Code Size)	Overall Score (Max=100)
LZC	4	3	2	80
LZRW1	2	4	4	80
LZRW3A	3	3	3	75
ADAP	4	1	3	65
WNC	2	1	2	40
RICE	1	2	2	40

Table 3. Ranking of Six Algorithms with CRUSH

The approaches taken by CRUSH and CHURN each have their own advantages and disadvantages. CRUSH compares compression algorithms and is itself a compression program in that it compresses and decompresses data files, but it is rather cumbersome to integrate many compression algorithms into CRUSH. CHURN is strictly a utility for testing compression programs, and it allows any compression and decompression programs to be called via command-line arguments. CHURN compresses any given input file to an arbitrary file named "TEST.CMP" and decompresses it to a file called "TEST.OUT". Some programs like PKZIP keep the original name of the input file, which require the overhead of copying the input file to a file called "TEST.OUT" as expected by CHURN and, thus, interfere with the overall compression times.

Another study that compares 47 compression programs over various large collections of data was conducted by Greg Flint of Purdue University. The raw test results are available via anonymous ftp from any SimTel archive site as arctst03.zip, such as at oak.oakland.edu (141.210.10.117) in the SimTel/msdos/info directory. Some preliminary examination of these results shows them to be consistent with those presented in this paper.

Results

The best class of programs is based on the Lempel-Ziv adaptive dictionary-based algorithm. Arithmetic compression is good in terms of compression ratio, but worst in terms of speed. The overall best are GZIP and LZC (UNIX Compress), which achieve the highest compression ratios and lowest elapsed time. As external stand-alone compression programs GZIP and LZC are the best all around in addition to being portable on MS-DOS, UNIX, VMS, and Macintosh platforms. The third comparison criterion requires small and simple functions to be incorporated into a much larger data management package. The two algorithms that meet all three criteria are the LZRW1 and LZRW3A algorithms, which are fast, good, simple algorithms.

Unfortunately the LZC implementation, based on the LZW algorithm, is covered by patents (US 4,464,650 and 4,558,302). The terms are defined under which Unisys will license the algorithm to modem manufacturers, but it has not stated how it applies to other types of products and this is one of the reasons for the CompuServe GIF/Unisys controversy. The LZRW algorithms are also covered by one or more patents. After speaking to the author of these two algorithms, it appears that the patents are thoroughly discouraging the implementation and use of many data compression techniques. The GZIP algorithm, however, is a variation of LZ77, free software, and free of patents. GZIP meets the first two criteria for a fast and good compression algorithm, but the code is neither simple nor straightforward for incorporating into a large data management package.

CDF is distributed worldwide as public domain software and part of several commercial data analysis and visualization packages, such as RSI's Interactive Data Language (IDL) and IBM's Visualization Data Explorer, so there may be some legal complications, but these legal issues are beyond the scope of this paper. Aside from these legal issues, these four LZ-based algorithms are the recommendations for good general-purpose data compression algorithms.

Future Plans

Up to now, data compression has been conducted on a collection of data files. The future plans for CDF are to incorporate the LZC and possibly another data compression algorithm into a test version of the CDF data management package, where compression and decompression will be transparent to CDF applications. Various techniques will be tried to determine what to compress for efficient accessing and minimizing the impact on performance.

Acknowledgments

Many thanks to Dr. Jim Thieman and Greg Goucher for their insight and guidance, Dr. Mona Kessel and Dr. Bill Mish for the ISTP data, Mark Nelson for developing CHURN, Ed Seiler for developing CRUSH, Miranda Robinson for editing this paper.

Author

Mr. Gary Jason Mathews is a computer engineer at the NSSDC Interoperable Systems Office of the Goddard Space Flight Center (GSFC). He has earned a B.S. in computer science from Columbia University and an M.S. in computer science from the George Washington University, and has been at GSFC for 5 years. The focus of his research is the development of portable and reusable software tools. In addition to data compression research, he develops advanced scientific data analysis systems, WWW-based data systems, data management tools, and electronic information management systems that support NASA missions.

References

[1] Goucher, G. W., and Mathews, G. J., *A Comprehensive Look at CDF*, NSSDC/WDC-A-R&S 94-07, August 1994. (See URL: http://nssdc.gsfc.nasa.gov/cdf/cdf_home.html)

[2] Nelson, M., “DDJ Data Compression Contest Results,” *Dr. Dobb's Journal*, November 1991, pp. 62-64.

[3] Teague, M., “ISTP Key Parameter Available on CD-ROM,” *Solar-Terrestrial Energy Program (STEP) International*, Vol. 4, No. 7, July 1994, pp. 4-5.

[4] Ross, E., “A Simple Data Compression Technique,” *The C User's Journal*, October 1992, pp. 113-120.

[5] Williams, R. N., “An Extremely Fast ZIV-Lempel Data Compression Algorithm,” *Proceedings Data Compression Conference '91*, pp. 362-371.

[6] Nelson, M., *The Data Compression Book*, M&T Books, San Mateo, CA, 1992.

[7] Ziv, J., and Lempel, A., "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, Vol. 23, No. 3, May 1977, pp. 337-343.