DEVELOPMENT STANDARD

QIC-154
Revision A
10 Mar 94

ADAPTIVE LOSSLESS DATA COMPRESSION (ALDC)

Quarter-Inch Cartridge Drive Standards, Inc.

311 East Carrillo Street
Santa Barbara, California 93101
Telephone (805) 963-3853
Fax (805) 962-1541
www.qic.org

(See important notices on the following page)

SCOPE

This document describes and formally defines the ALDC data compression method and compressed data stream format.

REFERENCES

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Trans. Inform. Theory, vol. IT-23, no.3, pp. 337-343, 1977.

[2] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," J. ACM, vol. 29, no. 4, pp. 928-951, 1982.

[3] T. C. Bell, "Better OPM/L Text Compression," IEEE Trans. Comm. vol. COM-34, no. 12, December, 1986.

[4] R. P. Brent, "A Linear Algorithm for Data Compression," Australian Computer J., vol. 19, no. 2, pp. 64-68, May, 1987.

[5] ALDC Product information packet, available from IBM Corp., 1000 River Street, Dept. N97/Bldg 861-2, Essex Junction, VT 05452-4299.

OVERVIEW

The ALDC (Adaptive Lossless Data Compression) algorithm is one variant of the LZ_1 (Lempel-Ziv 1) class of data compression algorithms, first proposed by Abraham Lempel and Jacob Ziv in 1977 (see Reference 1).

LZ_1 algorithms achieve compression by building and maintaining a data structure, called a HISTORY_BUFFER. An LZ_1 encode process and an LZ_1 decode process both initialize this structure to the same known state, and update it in an identical fashion. The encoder does this using the input data it receives for compression, while the decoder generates an identical data stream as its output, which it also uses for the update process. Consequently, these two histories can remain identical, so it is never usually necessary to include history content information with the compressed data stream sent from an encoder to a decoder.

Usually, for general purpose use, LZ_1 encoders or decoders both reset their history structures to a cleared or empty state. Incoming data is copied into the history, and is initially encoded explicitly. However, as the history fills, it becomes increasingly possible for the encoder to represent incoming data by encoding a reference to a string already present in this history. This is the principal mechanism by which LZ_1 algorithms are able to achieve compression.

ENCODER AND DECODER OPERATION

An encoder processes incoming data 1 byte at a time. Each byte of data processed is always copied in sequence to the history, the oldest data being displaced if the history is already full. Thus a sequential copy of the most recently processed data is always available.

The compression process consists of examining the incoming data stream to identify any sequences or strings of data bytes which already exist in the encoder history. If an identical such history is available to a decoder, this matching string can be encoded and output as a 2 element COPY_POINTER, containing a byte count and history location. It is then possible for a decoder to reproduce this string exactly, by copying it from the given location in its own history. If the COPY_POINTER can be encoded in fewer bits of information than required for the data string it specifies, compression is achieved.

If an incoming byte of data does NOT form part of a matching string, a LITERAL, containing this embedded value, is encoded and then output to explicitly represent this byte.

A decoder performs the inverse operation by first parsing a compressed data stream into LITERALS and COPY_POINTERS for processing.

A LITERAL is processed simply by determining the data byte value which is embedded in it. This value is then output as the next data byte and is also copied into the decoder history. The decoder history will then once more be identical to that existing within the encoder.

A COPY_POINTER is processed by first decoding the specified byte count and history location elements of the pointer. Then this string of data byte values is read, one byte at a time, from the decoder history. The data byte values are each output as decoded data, and also copied into the history, before the next value is accessed. Once the entire string has been processed in this manner, the decoder history is identical to that in the encoder once more.

DERIVATION OF ALDC ENCODING STRUCTURE

The many LZ_1 variants differ mainly in the size of history structure, and in the encoding method used to represent literals and pointers.

The original scheme published by Lempel and Zip (see Reference 1) used a series of triples as its output. These can be regarded as a sequence of strictly alternating history references and characters, the history reference requiring two values of the triple and the character one.

In the literature, this scheme is commonly referred to as LZ77, and in terms of the nomenclature introduced above, an alternating sequence of COPY_POINTERS and LITERALS is always generated as its output.

In Bell's 1986 paper (see Reference 3) he implements a suggestion made earlier by Storer and Szymanski in 1982 (see Reference 2), in which it is proposed that the Lempel-Ziv algorithms use a free mixture of these history pointers and literal characters, literals only being used when a history pointer takes up more space than the characters it codes.

Bell's implementation of this scheme is called LZSS, and adds an extra bit to each pointer or character to distinguish between them. The ALDC encoding uses this same principle exactly, thus each LITERAL is always 9 bits in length, consisting of a single 0 bit, followed by the 8 data bits of the embedded character value.

The 1987 paper by Brent (see Reference 4) describes the use of hashing to implement the history string match process rapidly in software, and also uses variable length codes to efficiently encode both the literal values and the ordered value pairs constituting history references.

Brent's implementation of this scheme is called SLH, and it uses a two pass technique, in which the literal values and the history references are first determined. An optimal Huffman code is then generated, based on the statistical frequencies of these values, and used to encode the output in a second pass. Brent also describes the use of a single pass method in which an adaptive Huffman or Arithmetic coding technique can be used to encode the literals and history references as they occur.

ALDC implementations use an exhaustive search, that is, every possible candidate is considered for string matching operations. Empirically, a fairly uniform distribution of COPY_POINTER location values results if this is done, while the length values tend to approximate closely to a logarithmic distribution. This result is observed over a wide range of types of data commonly stored or manipulated by computer systems.

Consequently, ALDC uses a non-adaptive coding method for these values, the history location being encoded on a fixed length binary field, and the length values on a quasi-logarithmic code, these being the optimal theoretical solutions for encoding these kinds of distributions.

The length code used in ALDC varies in size from 2 to 12 bits, and can represent a range of 286 values in total. The last 16 consecutive code values in the range, all 12-bit numbers, are reserved for control use, and at this time only the uppermost code is defined. This is used as a marker to indicate the end of a compressed data stream.

There are thus 270 length codes available, which allow encoding of the matching string lengths of from 2 thru 271 bytes. Matching single byte strings are not encoded as

COPY_POINTERS in ALDC, since no compression would result (see Reference 3).

ALDC history structures must be reset to an empty, or cleared state at the start of an operation. Data is always stored in ascending location sequence to these history structures, starting with a location address of zero. This location, or displacement value, must reset to zero each time the entire history is filled, and the update location is NOT used for either initiating or continuing string matching operations.

ALDC HISTORY_BUFFER SIZES

ALDC is defined for three different sizes of HISTORY_BUFFER, expressed in multiples of 512 byte units represented by a suffix. Thus ALDC with a 512 byte history structure is referred to as ALDC_1, if the size for the history structure is 1024 bytes, it is denoted by ALDC_2, while if a 2048 byte history structure is provided, it is denoted by ALDC_4.

The history location field in the COPY_POINTER is a binary field which simply contains the displacement in the history structure of the start of the referenced string, and its size is therefore 9 bits for ALDC_1, 10 for ALDC_2 and 11 bits for ALDC_4 implementations.

Compression in general increases with larger history structures, but a typical increase in compression ratio is about 3% for doubling history structure size from 512 to 1024 bytes, and from 1024 to 2048 bytes. In the case of hardware implementations, however, chip area and power are roughly doubled each time. The 512 byte history structure size is thus recommended as providing the best tradeoff.

This results in simple, fast hardware designs which achieve comparable compression ratios to other LZ_1 implementations requiring much larger history structures. ALDC hardware speeds are approximately an order of magnitude faster than other contemporary LZ_1 hardware implementations and require very little Silicon area to implement, typically under 1/4 to 1/10 of the area of a 10 x 10 mm chip for ALDC_1 (see Reference 5).

FORMAL ALDC FORMAT DEFINITION

The compressed data stream encoding format is described using BNF-like language, the symbols being defined as follows :

Symbol	Definition
:=	the non-terminal on the left side of the ":=" can be re-placed by the expression on the right side
<name>	non-terminal expression

[]	the expression inside the "[]" may occur 0 or more times
|	the logical disjunction "or"
()	the text enclosed within the () is a comment only, added for clarity
0,1	the terminal binary digits 0 or 1

ALDC_1 ALGORITHM - FORMAL DEFINITION (512 BYTE HISTORY)

```
<Compressed_Data>    :=   [ 0 <LITERAL> | 1 <COPY_PTR> ] 1 <CONTROL>

<LITERAL>   :=    <b><b><b><b><b><b><b><b>          (8-bit byte data)

<b> := 0 | 1         (if you've not realized it by now, | denotes OR)

<COPY_PTR> := <length_code><displacement>

<length_code> :=     (the length coding can be 2, 4, 6, 8 or 12 bits)
```

(length_code)	(field value)	(COPY_PTR length)
00	(0)	(2 bytes) |
01	(1)	(3 bytes) |
10 00	(2)	(4 bytes) |
: : : :	: : :	: : : : : : :
10 11	(5)	(7 bytes) |
110 000	(6)	(8 bytes) |
: : : : : :	: : :	: : : : : : :
110 111	(13)	(15 bytes) |
1110 0000	(14)	(16 bytes) |
: : : : : : : :	: : :	: : : : : : :
1110 1111	(29)	(31 bytes) |

1111 0000 0000	(30)	(32 bytes) |
: : : : : : : : : : : :	: : :	: : : : : : :
1111 1110 1110	(268)	(270 bytes) |
1111 1110 1111	(269)	(271 bytes)

<displacement> := <b><b><b><b><b><b><b><b><b> (9-bit)

<CONTROL> :=

(ctl_code)	(field value)	(control specified)
1111 1111 0000	(270)	(the 12-bit field values of 270 to 284 are reserved codes and cannot be used)
: : : : : : : : : : : :	: : :	
1111 1111 1110	(284)	
1111 1111 1111	(285)	(End_Marker control)

ALDC_2 ALGORITHM - FORMAL DEFINITION (1024 BYTE HISTORY)

(identical to ALDC_1 definition except for 10-bit displacement field)

<displacement> := <b><b><b><b><b><b><b><b><b><b> (10-bit)

ALDC_4 ALGORITHM - FORMAL DEFINITION (2048 BYTE HISTORY)

(identical to ALDC_1 definition except for 11-bit displacement field)

<displacement> := <b><b><b><b><b><b><b><b><b><b><b> (11-bit)