CITS2200 Project (Lossless data compression/decompression)
A practical implementation of the DEFLATE algorithm

Ridge Shrubsall (21112211)

Introduction

The DEFLATE algorithm is a lossless data compression algorithm that uses a combination of both the LZ77 algorithm and Huffman coding. It is used in a wide range of applications, including ZIP archives, GZIP compressed files and PNG images. For this project I have implemented the DEFLATE algorithm as part of a working GZIP file compression program. The GZIP file format is a simple encapsulation of a DEFLATE-compressed data stream which includes extra file-related information that can be used to verify the integrity of the data.

Firstly, I will discuss how the two different compression algorithms that are used by DEFLATE work in general; secondly, I will give an in-depth explanation of how the DEFLATE algorithm operates as a whole and finally I will describe the structure of a GZIP file.

Lempel-Ziv sliding window compression

The LZ77 algorithm works by finding sequences of characters that occur multiple times within the data and replacing them with a reference to an earlier occurrence. To do this, the algorithm must maintain a window of the characters that it has last seen and then look through it to find a match for a certain position in the data. These matches are encoded as a pair of values called a distance/length pair; the distance being how far back in the window the match is and the length being the size of the match. Take the following example:

Window	Data
a b c d e f g h i j A	a b c d e f B C D d e f E F G

The string “abcdef” is the longest sequence of characters that we can make from the current position that also exists in the window. The ‘a’ is 11 characters back in the window, so we output a distance/length pair of <11, 6>.

Window	Data
a b c d e f g h i j A a b c d e f B C D	d e f E F G

Here the longest possible match is “def”, which occurs twice in the window. The DEFLATE algorithm favours the match with the shortest distance in order to take advantage of the Huffman encoding, so we use the match at distance 6 rather than the match at distance 17. DEFLATE also limits the match length to a minimum of 3 and a maximum of 258.

Window	Data
a b c d e .	b c d e . b c d e . b c d e . b c d e . 1 2 3

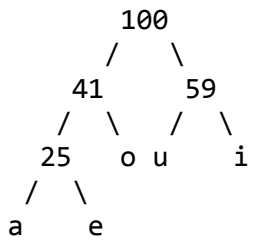
One important point to note is that the length of a match can exceed the current length of the window. At whatever point that we begin looking in the window, the window wraps back around to that point indefinitely. In this case, we can have a match “bcde.” of length 20.

Huffman coding

The Huffman algorithm aims to reduce the overall entropy of the data by transforming sequences of bits that occur frequently into shorter ones and sequences of bits that are less frequent into longer ones. The original bit sequences are called symbols and the resulting bit sequences are called codes. To construct a set of codes with optimal lengths, the algorithm builds a prefix tree based on the probabilities of each symbol. Next, the algorithm creates a leaf node for each symbol and assigns it a probability. The algorithm then proceeds to recursively join the two nodes with the lowest probability, creating a new node with a combined probability until there is only one node left. This node becomes the root node.

Here is an example of a generated Huffman tree:

Character	Frequency	Symbol	Code
a	10	01100001	000
e	15	01100101	001
i	30	01101001	11
o	16	01101111	01
u	29	01110101	10



The DEFLATE algorithm requires that these codes be converted to canonical codes, which means that for all codes of a given length, the codes are lexicographically ordered starting from the lowest possible value. The canonical codes can be constructed from the optimal code lengths by going up through the lengths and assigning the next possible code to each symbol with that length in the same order as the symbols. These codes can be stored more efficiently as only the code lengths need to be written to the output file.

DEFLATE also places some restrictions on the generation of the Huffman tree. The tree is required to have at least two leaves, and the generated codes are required to be length-limited to a maximum length of 15 bits when encoding literals. This means that the tree will need balancing after being created if its height exceeds 15. The worst case for tree height occurs when the frequencies are in a Fibonacci-like sequence, resulting in a degenerate tree similar to a linked list. At most $\log_2(n)$ bits are needed to represent n different symbols, so for 288 different literal symbols the tree can always be balanced within 15 bits as the symbols can all be represented with a maximum of 9 bits.

DEFLATE

The DEFLATE algorithm works by compressing a file in a series of blocks; each block is processed with the LZ77 algorithm and then Huffman coded using two different trees, one for literals and lengths and one for distances. The Huffman trees for each block are generated independently of one another but the LZ77 sliding window is continuous and can reference previous blocks up to a maximum distance of 32 KB.

Each block has a header containing the code lengths for the two Huffman trees that it uses followed by the compressed data. (The code lengths themselves are also run-length encoded and then Huffman coded.) The data consists of both literal-length symbols with values between 0 and 285 and distance symbols with values between 0 and 29.
For a literal-length symbol, any value between 0 and 255 is interpreted as a normal byte. A value of 256 signifies the end of the block. Any value between 257 and 285 is interpreted as a length symbol for a distance/length pair. After a length symbol follows some extra bits (dependent on the length symbol), a distance symbol and some more bits (dependent on the distance symbol). A distance symbol always occurs directly after a length symbol so there is no ambiguity between the symbol sets.

The DEFLATE standard specifies how many bits follow a certain length or distance symbol and how these are translated into length and distance values. This information is contained within “LZPair.java” and can be printed using its main method.

GZIP

The GZIP file format consists of a header and footer around the DEFLATE-compressed data. The header that this implementation uses is as follows:

```
0x1f 0x8b                GZIP header magic
0x08                     Compression algorithm (DEFLATE)
0x08                     Flags (filename present)
0x00 0x00 0x00 0x00      File modification time (not used)
0x00                     Extra flags (none)
0x00                     File system (FAT)
... 0x00                 File name, null terminated
```

The footer is as follows:

```
0x?? 0x?? 0x?? 0x??      CRC32 of the uncompressed data
0x?? 0x?? 0x?? 0x??      Size of the uncompressed data
```

Pseudo-code overview

```
Deflater.process() {
    Create in-memory block stream
    Create input buffer
    Create sliding window

    While we can fill the input buffer:
        If the block stream contains data:
            Output final bit + mode bits
            Output the block
            Clear the block stream

        Update running checksum of input data

        For an uncompressed block:
            Write block length
            Write block length (xor 65535)
            Write data
            Update window with data

        Initialise distance/length pair array
        Initialise frequency arrays
        Start looking byte-by-byte through the input buffer:
            Try and find a match in the window for the current position
            If a match was found:
                Save distance/length pair
                Update window with match bytes
                Update distance/lit-length symbol frequencies
                Skip ahead to the end of the match
            Else:
                Update window with current byte
                Update literal symbol frequencies
        Add one ‘end of block’ marker to frequencies

        For dynamic Huffman:
            Build Huffman tree for literals
            Generate canonical Huffman codes for literals
            Build Huffman tree for distances
            Generate canonical Huffman codes for distances
            Pack codelengths using a run-length encoding
            Find length symbol frequencies
            Build Huffman tree for lengths
            Generate canonical Huffman codes for lengths
        Else:
            Use fixed-length Huffman codes for literals
            Use fixed-length Huffman codes for distances

        For dynamic Huffman:
            Write number of used literals
            Write number of used distances
            Write number of used lengths
            Write length codelengths (in special ordering)
            Write packed literal/distance codelengths
        Go through the input buffer:
            Is there a saved distance/length pair for the current position?
                Write length symbol + extra bits
                Write distance symbol + extra bits
                Skip ahead ‘length’ bytes
```

```
            Else:
                Write literal symbol
        Write ‘end of block’ marker
        Flush remaining bits on block stream

    Output final bit + mode bits
    Output the final block
    Flush remaining bits on output stream
}

Inflater.process() {
    Read in a block:
        Read final bit + mode bits
        For an uncompressed block:
            processUncompressedBlock();
        For dynamic Huffman:
            readCodes();
            processHuffmanBlock();
        Else:
            loadDefaultCodes();
            processHuffmanBlock();
        Break if final bit set
}

Inflater.processUncompressedBlock() {
    Read block length
    Read block length (xor 65535)
    Read data
    Update running checksum of output data
    Update window with data
}

Inflater.loadDefaultCodes() {
    Use fixed-length Huffman codes for literals
    Build literal code map
    Use fixed-length Huffman codes for distances
    Build distance code map
}

Inflater.readCodes() {
    Read number of used literals
    Read number of used distances
    Read number of used lengths
    Read length codelengths (in special ordering)
    Build length codes:
        buildCodes(lenCodeLen);
    Build length code map
    Read and unpack literal/distance codelengths:
        readSymbol(lenCodes, lenCodeMap);
    Build literal codes:
        buildCodes(litCodeLen);
    Build literal code map
    Build distance codes:
        buildCodes(distCodeLen);
    Build distance code map
}
```

```
Inflater.processHuffmanBlock() {
    While not end of block reached:
        Read a literal symbol:
            readSymbol(litCodes, litCodeMap);
        If literal is between 0-255:
            Convert to byte
            Update running checksum with byte
            Update window with byte
            Output byte
        Else if literal is 256:
            End of block reached
        Else if literal is between 257-286:
            Convert to literal-length symbol
            Get length value
            Read distance symbol:
                readSymbol(distCodes, distCodeMap);
            Get distance value
            Copy bytes from window
            Update running checksum with bytes
            Update window with bytes
            Output bytes
}

Inflater.readSymbol() {
    Do:
        Check if maximum code length will be exceeded
        Read one bit of the code
        Increment code length
        Check for a matching code with the given length
    While no code found
    Return code index / symbol value
}

Inflater.buildCodes(int[] codeLen) {
    Make a list of all used code lengths
    Go upwards through the code lengths:
        Generate first code for symbols of the given length
        For each symbol of the given length:
            Generate the next possible code for the current symbol
}

HuffmanTree(int[] freq, int limit) {
    Create a leaf node for each symbol with a frequency > 0
    Ensure that there are at least two leaf nodes
    Add leaf nodes to priority queue
    Build the tree:
        Repeat n – 1 times (where n is the number of symbols):
            Take the next two nodes with the lowest frequencies
              from the priority queue
            Join the two nodes with a combined frequency
    Traverse the tree to find the maximum depth
    Balance the tree (adjust height to be <= limit)
        Pick a leaf over the limit and remove it
        Move its opposite leaf up one
        Move a leaf under the limit down one
        Reinsert the old leaf
        Recalculate maximum depth by traversing the tree again
}
```

```
HuffmanTree.getTable() {
    Go through each level of the tree:
        Generate first code for symbols at the given depth
        For each symbol at the given depth:
            Generate the next possible code for the given symbol
}

LZWindow.find(byte[] buffer, int off, int len) {
    Return null if window is empty

    Start looking backwards through the window:
        Mark starting position
        Initialise match length to 0
        Go through the window comparing byte-by-byte:
            Break if bytes are not the same
            Increase match length
        Return match if match length is > 3
}

LZWindow.getBytes(int dist, int len) {
    Create a new byte array of size ‘len’
    Go back ‘dist’ bytes
    Mark starting position
    Go through the window byte-by-byte ‘len’ times:
        Copy one byte
        Move up one byte
        Apply window wraparound (goes back to starting position)
    Return byte array
}
```

Theoretical analysis

Primitive arrays and byte-level manipulations have been used over lists or strings wherever possible to improve the overall compression/decompression speed.

The parts of the program with the highest complexity are the bit input and output streams, the Huffman tree balancing and traversal methods, the Huffman table codelength packing methods and the LZ77 sliding window lookup method.

The bit read/write methods run in O(n) time where n is the total number of bits – these methods are optimised to write whole bytes when n = 8 and the boundary is aligned, however further optimisations could be made to write several bits in one go and/or precalculating the reverse of all possible bytes.

The Huffman tree balancing runs in O(n*m) time where n is the total number of symbols and m is the total number of leaves that are moved by the operation – this could also be improved so that repeat traversals of the tree are unnecessary.

The Huffman table codelength packing method runs in O(n) time where n is the number of codelengths – not much can be improved in here.

The LZ77 sliding window lookup runs in O(n*w*l) time where n is the buffer size, w is the window size and l is the length of the longest match. This method is the main bottleneck of the code, running much slower if the window size is increased. The implementation used to perform the search is a greedy algorithm which returns the first match that has a length greater than or equal to the minimum of 3.

Experimental analysis

The following tables list the compression ratios and overall time taken to compress and decompress a various selection of files using two different window sizes…

Window size = 256 bytes

File	% of original size	Compression time	Decompression time
e to 1000 digits (e.txt)	45.5%	0.187 s	0.269 s
pi to 1000 digits (pi.txt)	45.5%	0.185 s	0.266 s
ACM-ICPC test data (acm.txt)	43.7%	0.047 s	0.061 s
Long character runs (longruns.txt)	0.2%	0.034 s	0.043 s
Project source code (project_java.tar)	35.4%	0.100 s	0.104 s
Project classes (project_classes.tar)	47.4%	0.103 s	0.093 s
English text (tom_sawyer.txt)	52.5%	0.607 s	0.950 s
Lecture slides (21-Compression.pdf)	61.6%	0.427 s	0.580 s
Compressed music (souleye.mp3)	100.1%	13.915 s	20.625 s

Window size = 32,768 bytes

File	% of original size	Compression time	Decompression time
e to 1000 digits (e.txt)	50.8%	0.259 s	0.208 s
pi to 1000 digits (pi.txt)	50.7%	0.257 s	0.209 s
ACM-ICPC test data (acm.txt)	46.8%	0.057 s	0.055 s
Long character runs (longruns.txt)	0.2%	0.040 s	0.039 s
Project source code (project_java.tar)	25.6%	0.232 s	0.134 s
Project classes (project_classes.tar)	40.1%	0.571 s	0.138 s
English text (tom_sawyer.txt)	47.2%	2.055 s	0.677 s
Lecture slides (21-Compression.pdf)	49.6%	7.879 s	0.483 s
Compressed music (souleye.mp3)	100.0%	461.563 s	20.167 s

The compression and decompression times for plain text and uncompressed binary files are blindingly fast, with most of the compression ratios being less than 50% (i.e. the filesizes have been more than halved by the compression). The times for binary files which are already compressed are much slower as they exhaustively search the window buffer, finding little to no matches. Interestingly, the sources of random data (the first three text files) performed worse with a larger window size so the LZ77 encoding may have actually bloated the data through having lots of short pairs.

References

RFC 1951, DEFLATE compressed format data specification v1.3
http://www.gzip.org/zlib/rfc-deflate.html

RFC 1952, GZIP file format specification v4.3
http://www.gzip.org/zlib/rfc-gzip.html

An explanation of the DEFLATE algorithm
http://www.zlib.net/feldspar.html

DEFLATE implementation notes
http://www.gzip.org/algorithm.txt

Huffman code discussion and implementation
http://michael.dipperstein.com/huffman/

Restricting Huffman codes to a maximum length
http://www.arturocampos.com/cp_ch3-4.html

Notes on DEFLATE in PNG files
http://halicery.com/inflate/deflatenotes.html