Einführung

Die Datenkompression oder Datenkomprimierung ist ein Vorgang, bei dem die Menge digitaler Daten reduziert wird. Die Vorteile der Datenkomression liegen auf der Hand. Dieselben Daten können mit erheblich weniger Speicherkapazität persistiert werden. Die Datenübertragung wird verkürzt und Ressourcen können eingespart werden. So ebnete das MP3-Format den Weg zum Austausch von Musik über das Internet. Das Verfahren zur verlustbehafteten Kompression digital gespeicherter Audiodaten wurde Anfang 1990 bekannt und ermöglichte eine Einsparung der benötigten Speichergröße um den Faktor 10, gegenüber dem bis dato verwendeten WAV-Format, bei nicht oder nur kaum verringerter wahrgenommener Audioqualität.

Bei der Datenkompression ist man primär an einem sog. Codiergewinn interessiert, der sich als Differenz aus der Darstellung der Information gegenüber einer alternativen Darstellung der gleichen Information ergibt. Man spricht von einem Codiergewinn, wenn sich die alternative Darstellung als effizienter (kürzer) erweist, als die ursprüngliche Darstellung der Information. Grundsätzlich wird bei der Datenkompression versucht, überflüssige Information zu entfernen. Dazu werden die Daten in eine Darstellung überführt, mit der sich alle – oder zumindest die meisten – Informationen in kürzerer Form darstellen lassen. Diesen Vorgang übernimmt ein Kodierer und man bezeichnet den Vorgang als Kompression oder Komprimierung. Die Umkehrung bezeichnet man als Dekompression oder Dekomprimierung.

Fundamentalprinzipien der Datenkompression

Die Datenkompression bzw. Datenreduktion ist nur aufgrund zweier fundamentaler Prinzipien möglich. Das Erste ist die Entfernung von Redundanz (Kompression) i.e.S. und das Zweite ist die Entfernung von Irrelevanz (Reduktion) aus einer beliebigen Informationsquelle. Der Unterschied zwischen Kompression und Reduktion liegt bei näherer Betrachtung auf der Hand: Bei der Kompression (i.e.S) bleiben die Originaldaten nach der Abbildung (Kompressionsvorgang) vollständig rekonstruierbar und im Sinne einer Rückabbildung erhalten. Mathematisch formuliert spricht man von einer bijektiven Abbildung. Bei der Datenreduktion hingegen sind die Originaldaten nach dem Abbildungsvorgang nicht immer fehlerfrei rekonstruierbar. Das bedeutet, es kommt zu einem Fehler zwischen den Originaldaten und den rekonstruierbaren Daten. Eine Rückabbildungsvorschrift ist zwar vorhanden, diese ist jedoch nicht in der Lage, sämtliche Originaldaten einwandfrei wieder herzustellen.

Redundanzreduktion

Als redundant bezeichnet man umgangssprachlich Informationen oder Daten, die sich aufgrund bereits überlieferter/übertragener Informationen durch Deduktion ableiten lassen. Es ist demnach nicht notwendig diese sog. redundanten Daten gesondert, zusätzlich oder überhaupt

zu übertragen. Redundanz beschreibt den Grad der Vorhersagbarkeit von Teilen einer Nachricht und ist durch die Entropie festgelegt.

Die Redundanz innerhalb einer Nachricht/Information lässt sich mathematisch exakt bestimmen. Es ist trotzdem weder mit beliebigem technischem oder mathematischem Aufwand möglich eine größere Menge an Redundanz aus einer Nachricht zu entfernen, als die Nachricht enthält. Es sei denn man verfälscht die ursprüngliche Nachricht oder nimmt bewusst Fehler in der rekonstruierten Nachricht in Kauf.

Die Vermeidung von Redundanz erfolgt mit den Methoden zur verlustfreien Datenkompression. Bei der verlustfreien Datenkompression geht es darum, die zu übertragende Nachricht durch eine kürzere aber äquivalente Nachricht zu ersetzen, sie anschließend zu übertragen und auf der Seite des Empfängers fehlerfrei zu rekonstruieren. Bei der Codierung unterscheidet man statistische, Wörterbuch-basierende und kombinierte Ansätze.

Irrelevanzreduktion

Als irrelevant bezeichnet man Informationen oder Daten, die vom Beobachter bzw. Empfänger nicht wahrgenommen werden können oder deren Genauigkeit nicht hinreichend wahrnehmbar ist. Als schlechtes Beispiel könnte man einen Rot/Grün-Blinden wählen, der bekanntlich schwer zwischen Rot und Grün unterscheiden kann. Durch die fehlende Unterscheidbarkeit beim Empfänger würde es bspw. ausreichen, wahlweise den Rot oder den Grün-Anteil zu übertragen. Das heißt man benutzt physiologische Grundlagen, Gegebenheiten und Modelle um zu entscheiden, ob eine Information eine Übertragung wert ist oder nicht. Das MP3-Format nutzt die Irrelevanzreduktion, indem sie Frequenzen ausfiltert, die normalerweise vom menschlichen Gehör nicht wahrgenommen werden können.

Bei der Irrelevanz ist es mittels technischer oder mathematischer Modelle möglich, relevante Informationen von irrelevanten Informationen zu trennen. Meist müssen dazu komplexe Modelle, Messmethoden oder empirische Werte gefunden werden, anhand derer die Entscheidung vorgenommen werden kann, ob eine Information relevant oder irrelevant ist. Dabei spielen insbesondere objektive und subjektive Kriterien eine Rolle.

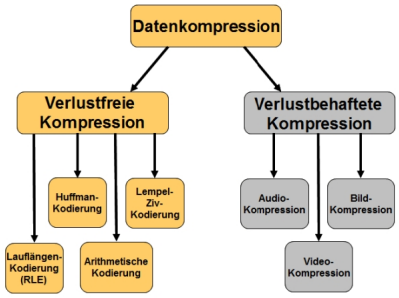
Darüber hinaus ist es möglich, nicht nur nicht hinreichend wahrnehmbare Informationen zu unterschlagen, sondern bewusst auch wahrnehmbare Fehler in Kauf zu nehmen, um so die zu übertragende Information an die Art der Codierung anzupassen. Der Vorteil der Anpassung an die Art der Codierung besteht darin, dass der Codierer entscheiden kann, welche Modifikation

während der Codierung zu einer besseren Kompressionsleistung führen. Der Codierer erzeugt durch die Modifikation sog. Kompressionsartefakte, die es dem Codierer möglich machen eine vorgegebene Kompressionsrate zu erreichen oder zu unterschreiten. Die Verfahren mit denen diese Art der Codierung erreicht werden kann, werden im Kapitel Verlustbehaftete Kompression behandelt.

Verlustfreie oder verlustbehaftete Datenkompression

Grundsätzlich unterscheidet man zwischen verlustfreier oder verlustbehafteter Datenkompression. Man spricht von verlustfreier Kompression (oder verlustfreier Kodierung), wenn aus den komprimierten Daten wieder alle Originaldaten gewonnen werden können. Das ist beispielsweise bei der Kompression ausführbarer Programmdateien notwendig.

Bei der verlustbehafteten Kompression können die Originaldaten nicht mehr aus den komprimierten Daten zurückgewonnen werden, das heißt ein Teil der Information ging verloren. Solche Verfahren werden häufig zur Bild- oder Videokompression und Audiodatenkompression eingesetzt.



Datenkompression in der Informationstheorie

Es gibt eine Vielzahl von Beispielen, anhand derer gezeigt werden kann, dass die verlustfreie oder verlustbehaftete Datenkompression sinnvoll sind. Die Datenkompression ist eng mit der Informationstheorie, wie sie von Claude Shannon im Jahr 1948 formuliert wurde, verbunden. Die Informationstheorie und die Informationstheoretischen Ansätze haben nach über 50 Jahren der Existenz derart viele Bereiche berührt, dass die moderne Informatik und Informationstechnik nicht ohne diese theoretische Abhandlung denkbar wären. Heute geht es mehr denn je, um die Speicherung und Verarbeitung digitaler Daten und es ist völlig egal in welche Richtung man blickt. Die moderne Medizin kommt heute nicht ohne bildverarbeitende Methoden aus. Nachdem die Information digital wurden, suchte man nach Möglichkeiten sie auch digital zu übertragen, digital zu speichern und digital zu verarbeiten. Erst dadurch gelang es Informationen digital und somit verlustfrei zu verarbeiten oder zu kopieren.

Nachdem die Information digital aufgezeichnet werden konnte, wollte man erfahren, wie viel Information tatsächlich in einer Nachricht steckt. Diese Frage ist sowohl von philosophischem, als auch von ingenieurstechnischem Interesse. Die ingenieurstechnischen Fragen beantwortete Claude Elwood Shannon (Vater der Informationstheorie) mit seiner berühmt gewordenen Arbeit "A Mathematical Theory of Communication". Sie wurde 1949 unter dem Titel "The Mathematical Theory of Communication" leicht erweitert und erneut veröffentlicht.

Information: der negative Kehrwert der Wahrscheinlichkeit.

Die Informationstheorie gibt durch den Informationsgehalt eine minimale Anzahl an Bits vor, die zur Kodierung eines Symbols benötigt werden. Dabei ist die Entropie ein Maß für den mittleren Informationsgehalt oder auch Informationsdichte einer Nachricht. Der Begriff in der Informationstheorie ist in Analogie zur Entropie in der Thermodynamik und Statistischen Mechanik benannt.

Je kleiner die Auftrittswahrscheinlichkeit eines Zeichens ist, desto höher ist seine Information. Andersherum ist die Information eines Zeichens gering, wenn es oft vorkommt. Eine niedrige Entropie bedeutet, dass es mehr redundante Informationen gibt und es lassen sich somit gute Kompressionsraten erzielen. Aus einer niedrigen Entropie kann man auf regelmäßige Strukturen in der Nachricht schließen. Umgekehrt gilt: ist die Entropie hoch, so sind weniger redundante Informationen in der Nachricht enthalten und sie lässt sich nur schlecht komprimieren. Die Entropie kann somit als Maß dafür gewertet werden, wie gut sich eine Datenquelle für eine Kompression eignet. Die Entropie spielt auch bei der Generierung von Zufallszahlen und der Verschlüsselung von Daten eine zentrale Rolle. Sie ist auch der Grund, warum beispielsweise verschlüsselte Daten in der Regel nicht mehr effektiv komprimiert werden können.

Nach über 50 Jahren der Informationstheorie gibt es keinen Bereich, der nicht von der Informationstheorie berührt worden ist und es gibt keinen Bereich, der sich früher oder später nicht mit der Codierung und Kompression von Daten bzw. Informationen beschäftigen muss.

Anwendung entwickeln

In diesem Artikel wird die Entropiekodierung zur verlustfreien Datenkompression vorgestellt. Die Entwicklung der Software vollzieht sich in mehreren Schritten. Zunächst werden Klassen zur Serialisierung von Daten entworfen. Anschließend wird die Huffman-Kodierung in C++ implementiert. Am Ende werden die Teilschritte in eine fertige Anwendung integriert, die die Komprimierung von beleibigen Dateien ermöglicht.

Serialisierung von Datentypen

In der Informatik bezeichnet die Serialisierung im Kontext der Datenspeicherung und Datenübertragung den Prozess der Konvertierung von Datenstrukturen oder Objekten in ein Format, das persistent gespeichert oder über ein Netzwerk übertragen werden kann um anschließend in derselben oder einer anderen Umgebung wieder „genutzt“ zu wer-den. Die Serialisierung ist eine Abbildung von Objekten auf eine sequenzielle Darstellung. Wenn die sequenziellen Bits wieder nach dem Objektserialisierungsformat gelesen werden, ist es möglich einen semantisch identischen Klon des Originalobjekts zu erzeugen.

Die Umkehrung der Serialisierung, bei der ein Datenstrom in ein Objekt konvertiert wird, heißt Deserialisierung.

Objektserialisierung

Bei der Entwicklung einer Anwendung zur Datenkomprimierung steht eine konsistente Serialisierung von Objekten im Vordergrund. So müssen nicht nur primitive Datentypen komprimiert werden, sondern auch Bilder und andere komplexe Datenquellen. Anders als beispielsweise JavaTM, unterstützt die Programmiersprache C++ direkt keine Objektserialisierung. Basierend auf dem Qt-Framework werden deshalb zunächst Streaming-Klassen entworfen. Diese Klassen ermöglichen die Serialisierung von Daten zu einem so genannten IODevice. Der binäre Datenstrom ist zu 100% unabhängig von dem laufenden Betriebssystem, sowie unabhängig von der Byte-Reihenfolge der CPU. Die Klassen stellen Methoden für die Serialisierung der grundlegenden C++-Datentypen bereit, wie char, short, int, double, etc. Die Serialisierung von komplexen Datentypen wird durch die Nutzung der primitiven Datentypen realisiert und über die friend-Funktionalität von C++ ermöglicht.

Ziel ist es ein leistungsfähiges Serialisierungmodul zu entwickeln, das zum einen mit allen wesentlichen Datentypen umgehen kann, dabei eine einfach zu bedienende API bereitstellt und bei Bedarf jederzeit erweitert werden kann. Die API, sowie der Aufbau der Datenströme orientiert sich dabei an JavaTM.

Byte-Reihenfolge

Werden Daten bitweise seriell übertragen, so ist die Bit-Reihenfolge festzulegen. Die sogenannte Byte-Reihenfolge oder auch Byte-Order bezeichnet die Speicherorganisation für einfache Zahlenwerte, in erster Linie die Ablage von ganzzahligen Werten im Arbeitsspeicher. Die Festlegung des zu verwendenden Speicherungsformats ist immer dann nötig, wenn zur Codierung der zu speichernden Zahl mehr Bits erforderlich sind, als in der kleinsten adressierbaren Einheit zur Verfügung stehen. Im Regelfall ist die kleinste adressierbare Einheit

auf modernen Computerarchitekturen ein Byte bzw. acht Bit. Die Speicherung einer Zahl erfolgt nun, falls hierfür mehr als ein Byte benötigt wird, in mehreren Bytes, so dass diese in den Speicheradressen direkt aufeinander folgen.

Die heute existierenden Varianten der Byte-Reihenfolge beschränken sich fast ausschließlich auf zwei Varianten, „Big-Endian“ und „Little-Endian“, sowie eine Misch-form „Middle-Endian“ beider Varianten.

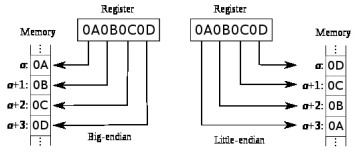
	Big Endian			Little Endian			Middle Endian		
Adresse	Hex	Dez	Binär	Hex	Dez	Binär	Hex	Dez	Binär
10000	A1	161	10100001	D4	212	11010100	B2	178	10110010
10001	B2	178	10110010	C3	195	11000011	A1	161	10100001
10002	C3	195	11000011	B2	178	10110010	D4	212	11010100
10003	D4	212	11010100	A1	161	10100001	C3	195	11000011

In Tabelle sind die drei wichtigen Byte-Reihenfolgen samt Adresse dargestellt. Das Format Little-Endian wurde ursprünglich bei den Intel-x86-Prozessoren verwendet. Dagegen wurde das Big-Endian-Format beispielsweise bei der Motorola beziehungs-weise Coldfire-Familie, sowie Sun-SPARC-CPUs und dem PowerPC eingesetzt. Das Betriebssystem Windows verwendet auf jeder Hardware das Format Little-Endian. Das betrifft x86, x86-64, Alpha, PowerPC, MIPS und Itanium Hardware-Architekturen. Unter Linux ergibt sich ein diversifiziertes Bild. Während Linux für die gängigen Systeme x86 und x86-64 ebenfalls Little-Endian einsetzt, wird z. B. auf einem MIPS, SPARC, PA-RISC, POWER, PowerPC, oder Xtensa auf Big-Endian zurückgegriffen. Das Betriebssystem von Apple, das Mac OS X, verwendet nur noch für den alten PowerPC das Format Big-Endian. Auch die gewöhnliche Darstellung von Dezimalzahlen ist – im Sinne der Leserichtung der meisten europäischen Sprachen von links nach rechts – Big Endian. In der Industrie, sowie auf diversen Plattformen hat sich aber zunehmend Little-Endian durchgesetzt. Sowohl Linux als auch Windows verwenden heute auf modernen x86 und x86-x64 Plattformen Little-Endian, so dass für gewöhnlich die Daten in diesem Format im Speicher abgelegt sind.

Die grundlegenden Unterschiede zwischen Little-Endian und Big-Endian sind:

- Unter Little-Endian wird das Byte mit den niederstwertigen Bits (engl. least significant bit) an der kleinsten Speicheradresse gespeichert beziehungsweise das kleinstwertige Element zuerst genannt.
- Unter Big-Endian wird das Byte mit den höchstwertigen Bits (engl. most signi-ficant bit) zuerst gespeichert, das heißt an der kleinsten Speicheradresse. Das bedeutet, dass Daten mit dem größtwertigen Element zuerst genannt werden.

Zahlen werden in Programmiersprachen, wie C++, in der Regel im Format Little-Endian geschrieben. Die Zahl beginnt mit den höchstwertigen Bits und endet mit den niederstwertigen Bits (Leserichtung von links nach rechts). Die interne Darstellung folgt allerdings exakt der umgekehrten Richtung. Das **LSB** steht am Anfang im Speicher, während das MSB den Schluss markiert.



Vor der Übertragung von Datentypen, die mehr als ein Byte im Speicher beanspruchen, wird im Protokoll von der Klasse Endian zunächst die Byte-Reihenfolge des laufen-den Betriebssystems ermittelt. Die Klasse verfügt darüber hinaus über ein Template mit dem Namen ReverseByteOrderImpl. Mit dem Template kann die Byte-Reihenfolge von diversen Datentypen umgekehrt werden. Um die Performance zu stei-gern wurden für die ersten 8 Byte insgesamt vier Template-Spezialisierungen generiert, die über direkte Bit-Manipulationen die Byte-Reihenfolge ändern.

Die Klasse „Stream“

Grundlage für die Serialisierung aller Datentypen ist ein kontinuierlicher Datenstrom. Der Datenstrom oder auch Stream, bezeichnet kontinuierliche Abfolgen von Datensätzen, deren Ende nicht im Voraus abzusehen ist. Die einzelnen Datensätze sind dabei von beliebigem, aber festem Typ. Da fast alle Systeme heute ein Byte als minimalen Datensatz verwenden, bestehen die Abfolgen meist aus einzelnen Bytes. Datenströme werden häufig zur Interprozesskommunikation verwendet.

Der Entwurf der Stream-Klassen beginnt in der Regel bei dem Fundament. Oftmals wird auch zwischen Eingabe- und Ausgabe-Streams unterschieden. Aus diesem Grunde ist es sinnvoll eine abstrakte Basisklasse zu entwerfen, die sowohl für die Ein- als auch für die Ausgabe Methoden und Felder bereitstellt. Die Klasse Stream ist so eine Klas-se. Sie stellt insgesamt sechs virtuelle Methoden zur Verfügung. Darunter befinden sich Methoden zur Festlegung der Byte-Reihenfolge, als auch eine Methode zur Ermittlung der Größe des Datenstroms. Weiterhin werden die Enumeratoren Version und ByteOrder exponiert. Der erste Enumerator dient dazu bei Versionsübergängen Funktionalitäten unterscheiden zu können um beispielsweise eine Abwärtskompatibilität zu gewährleisten. Der zweite Enumerator legt die Byte-Reihenfolge des Datenstroms fest. Die Klasse Stream bildet die Basis für alle I/O-Unterklassen.

Der nächste Schritt besteht darin den Eingabe- und Ausgabe-Streams aufzuspalten. Dazu werden die beiden abstrakten Klassen InputStream und OutputStream entwickelt. Beide Klassen besitzen mehrfache Überladungen für unterschiedliche Datentypen und implementieren die Basisklasse Stream. Die Überladungen sind durch die Methoden read() für Eingabe-Streams und write() für Ausgabe-Streams implementiert. Weiterhin wurde für die Datentypen der operator>> bzw. operator> overloads to read data from the stream /// // virtual InputStream& operator>>(bool& data) = 0; virtual InputStream& operator>>(char& data) = 0; virtual InputStream& operator>>(int8_t& data) = 0; virtual InputStream& operator>>(uint8_t& data) = 0; virtual InputStream& operator>>(int16_t& data) = 0; virtual InputStream& operator>>(uint16_t& data) = 0; virtual InputStream& operator>>(int32_t& data) = 0; virtual InputStream& operator>>(uint32_t& data) = 0; virtual InputStream& operator>>(int64_t& data) = 0; virtual InputStream& operator>>(uint64_t& data) = 0; virtual InputStream& operator>>(float& data) = 0; virtual InputStream& operator>>(double& data) = 0; //virtual InputStream& operator>>(const char* data) = 0; //virtual InputStream& operator>>(const wchar_t* data) = 0; virtual InputStream& operator>>(std::string& data) = 0; virtual InputStream& operator>>(std::wstring& data) = 0; struct true_type { enum { value = true }; }; struct false_type { enum { value = false }; }; template void do_read(T& data, uint32_t bit, const false_type&) { read(data, bit); } template void do_read(T& data, uint32_t bit, const true_type&) { int32_t temp; read(temp, bit); data = static_cast(temp); } };[/code] **ByteArrayInputStream und ByteArrayOutputStream**

Die konkreten Implementierungen werden durch die beiden Klassen ByteArrayInputStream und ByteArrayOutputStream dargestellt. Beide Klassen realisieren ihre Basisklassen und operieren auf Datenströmen der Größe ein Byte oder 8-Bit. Instanziiert werden beide Klassen mit einem Objekt vom Typ IODevice oder direkt mit einem ByteArray.

In dem nachfolgenden Listing ist ein Beispiel für die Nutzung einer der beiden Klassen zu sehen:

[code xml:lang="csharp"]ByteArray arr; ByteArrayOutputStream bos(&arr); bos.setByteOrder(ByteArrayOutputStream::LittleEndian); int a = 0x52535455; bos *m_heap[bottom]) { std::swap(m_heap[parent], m_heap[bottom]); sortHeapUp(root, parent); } } } template void PriorityQueue::sortHeapDown(int32_t root, int32_t bottom) { int32_t minchild = 0, firstchild = 0, child = 0; firstchild = root * m_d + 1; if(firstchild setSymbol(static_cast(i)); q.push(tree); } } // Create the main Huffman tree from the sub-trees HuffmanTree *tree2, *tree3; // Loop until all sub-trees are combined into one single tree do { tree = q.top(); q.pop(); if(!q.empty()) { // Extract the two lowest frequency Huffman trees, combine // them into one tree and put the tree back into the priority queue tree2 = q.top(); q.pop(); tree3 = new HuffmanTree(); tree3->setFrequency(tree->frequency() + tree2->frequency()); tree3->setLeftChild(tree->root()); tree3->setRightChild(tree2->root()); q.push(tree3);

} } while(!q.empty());[/code]

Der Huffman-Algorithmus liefert nicht zwangsläufig bereits nach der ersten Anwendung die optimale Kompression. In der Regel muss encode() deshalb mehrmals aufgerufen werden. Da in C++ nur byteweise geschrieben und gelesen werden kann, verfügt die Klasse Huffman über zwei interne Hilfsfunktionen, die es ermöglichen bitweise zu lesen und zu schreiben.

Die endgültige Kompression wird von den Klassen Inflater und Deflater vorgenommen. Beide Klassen abstrahieren den Kompressionsalgorithmus, so dass dieser bei Bedarf auch durch neue Algorithmen ersetzt werden kann. Die Datenkompression ist im Namensraum CodePlanet.Articles.Zip eingebettet und kann in der Anwendung über die Methoden inflate() und deflate() der jeweiligen Klassen genutzt werden.

Der nachfolgende Codeausschnitt zeigt die Verwendung der Klasse Deflator mit der Daten komprimiert werden können. Der Konstruktor kann verschiedene Strategien übernehmen, unter anderem für eine schnelle oder bestmögliche Kompression.

[code xml:lang="cpp"]ByteArray input, output; // Fill input array with data... Deflater def(Deflater::BEST_COMPRESSION); def.setInput(input); def.finish(); unsigned int compressedDataLength = def.deflate(compressedOutput);[/code]

Die Klasse Deflater schreibt neben den komprimierten Daten einen kleinen Kopfbereich (*engl. Header*) in das ByteArray. Der Kopfbereich dient dem Dekodierer bzw. dem Inflater dazu, die Anzahl der Komprimierungsrunden und eventuell gesetzte Modi zu erkennen.

Schluss

In diesem Artikel wurde mit dem Huffman-Verfahren eine sehr wirkungsvolle Art der Datenkompression vorgestellt. Der Code stellt keinen Anspruch an eine größtmögliche Effizienz und steht auch nicht in Konkurrenz zu modernen Datenkompressionsprogrammen, wie WinRAR oder 7-Zip. Vielmehr kann er als theoretisches Beispiel oder als Grundlage für die Entwicklung eigener Programme zur Datenkompression betrachtet werden. Die fertige Anwendung kann im Forum heruntergeladen werden.