Der Lempel-Ziv-Markov Chain Algorithmus

Arvid Butting

SS 2012

Zusammenfassung

Im Laufe der Zeit wurden Wörterbuchkompressionsalgorithmen ausgehend von LZ77 [1] mit vielen verschiedenen Ansätzen weiterentwickelt. Eine weitere Verbesserung stellt der *Lempel-Ziv-Markov Chain Algorithmus* (LZMA) dar. Diese Arbeit beschreibt schrittweise das (De-)Kompressionsverfahren von LZMA sowie die Grundzüge des neueren Formates LZMA2.

Grundlegend dazu wird der Bereichskodierer eingeführt. Zudem werden die Dateiformate 7z und xz erläutert und bezüglich ihrer Kompressionsrate mit anderen Kompressionsformaten verglichen.

Inhaltsverzeichnis

1 Einleitung

LZMA ist ein verlustfreies Kompressionsverfahren, das 1998 von Igor Pavlov entwickelt wurde. Der Eingabetext wird zunächst mit Hilfe eines Wörterbuchs komprimiert und die dadurch erhaltenen Daten werden anschließend mit einem Bereichskodierer kodiert. Um gute Kompressionsraten beim Bereichskodierer zu erhalten, macht LZMA eine Vorhersage der als nächstes zu erwartenden Zeichen. Zudem gibt es für jedes einzelne Bit der Daten einen spezifischen Kontext, mit dem die Verschmelzung nicht zusammengehöriger Daten, welche wiederum zu schlechterer Kompression führen würde, vermieden werden soll.

Besondere Vorteile von LZMA sind der geringe Speicherbedarf beim Dekomprimieren sowie das bis zu 4GByte große Wörterbuch, welches auch weit voneinander entfernte zusammenhängende Daten komprimieren kann.

Diese Arbeit beruht in großen Teilen auf dem *Handbook of Data Compression* von David Salomon und Giovanni Motta [2], sowie auf dem Artikel *An introduction to arithmetic coding* von Glen G. Langdon [3].

1.1 LZ77

Die Wörterbuchkompression in LZMA basiert auf dem 1977 von Abraham Lempel und Jacob Ziv entwickelten *Lempel-Ziv Algorithmus* (LZ77), wurde aber an einigen Stellen weiter verbessert.

LZ77 durchläuft den einzulesenden Datenstrom mit einem gleitenden Fenster, das aus einem Vorschaupuffer und einem Textfeld besteht. Der Vorschaupuffer enthält die als nächstes zu kodierenden Daten. Das Textfeld beinhaltet den letzten Teil der bereits eingelesenen Daten und dient dem Algorithmus so als Wörterbuch zum Finden von Übereinstimmungen variabler Länge (*Matches*).

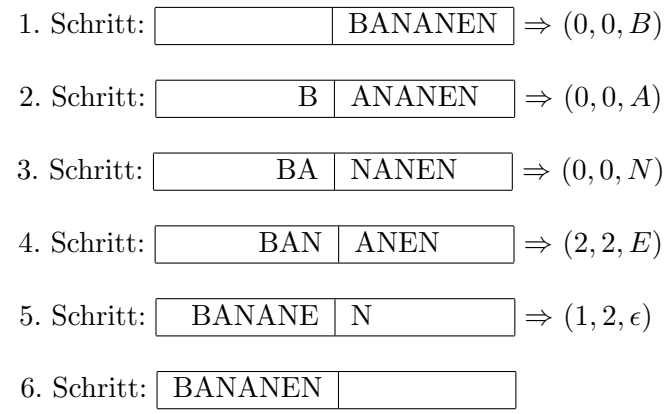
Wird ein solcher Match gefunden, so ersetzt der Algorithmus die Stelle im Vorschaupuffer durch einen Verweis auf die sich im Textfeld befindende Stelle. Der Verweis besteht in LZ77 aus einem Tripel $(dist, len, c)$. $dist$ ist der Abstand vom ersten eingelesenen Zeichens im Match zum analogen Zeichen im Textfeld, len ist die Anzahl der Zeichen, die gematcht werden können und c ist das fehlerverursachende Zeichen, also das erste Zeichen im Vorschaupuffer nach dem gefundenen Match. Anschließend wird das Fenster um $len + 1$ Zeichen weitergeschoben.

Wenn ein Zeichen z nicht gematcht werden kann, wird ein Tripel $(0, 0, z)$ im Textfeld gespeichert. Dies wird im Folgenden auch als *Literal* bezeichnet.

Beispiel Deutlich wird das Verfahren anhand eines Beispiels. Es soll das Wort "BANANEN"kodiert werden. Es wir angenommen, dass Textfeld und Vorschaupuffer jeweils die Länge 7 haben. Dies ist in der Praxis eher unüblich, da ein großes Textfeld zum Finden von Matches wichtiger ist als ein großer

Vorschaupuffer. Daher ist das Textfeld meist deutlich größer als der Vorschaupuffer.

Im nun folgenden Beispiel werden zunächst die drei Literale B, A und N kodiert, danach ein *Match* der Länge 2 mit einer Distanz von 2 und dem fehlerverursachenden Zeichen E als Tripel $(2, 2, E)$ und schließlich ein Match der Länge 1 für den Buchstaben N.



Die kodierte Ausgabe wäre demnach: (0,0,B) (0,0,A)(0,0,N)(2,2,E)(1,2,ϵ), wobei ϵ ein spezieller Marker dafür ist, dass das Ende der einzulesenden Datei erreicht ist.

1.2 Bereichskodierer

LZMA kodiert einzelne Blöcke mit Hilfe des Bereichskodierers. Der Bereichskodierer [**?**], 1979 von G.N.N. Martin entwickelt, ist eine Implementierung des arithmetischen Kodierens mit natürlichen Zahlen. Er ist patentfrei und kann daher problemlos in LZMA integriert werden.

1.2.1 Kodieren

Der Bereichskodierer benötigt als Eingabe ein Alphabet Σ, ein Wort $w \in \Sigma^*$ bestehend aus Zeichen $a_i \in \Sigma$, ein Intervall $I = [0, N)$ mit $N \in \mathbb{N}$ als Obergrenze sowie Wahrscheinlichkeiten P_{a_i} für jedes Zeichen a_i des Eingabealphabets. P_{a_i} gibt jeweils die Wahrscheinlichkeit für das Auftreten des Zeichens a_i in den Eingabedaten an. Es muss $\sum_{a_i \in \Sigma} P_{a_i} = 1$ gelten.

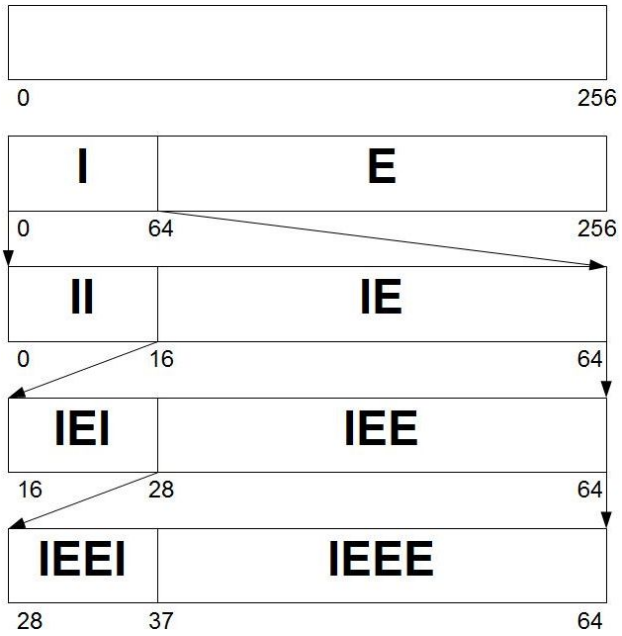
Die Ausgabe des Bereichskodierers ist dann ein Intervall $[low, range)$, wobei $low, range \in \mathbb{N}_0$ ist und $low \leq range \leq N$ gilt. Eine beliebige Zahl $c \in [low, range)$ ist dann eine eindeutige Kodierung des Eingabewortes w. Das Eingabewort wird nun von links nach rechts durchlaufen. Für jedes Zeichen des Eingabewortes w gibt es eine Iteration, in der ein neues Intervall berechnet wird. Eine Iteration läuft wie folgt ab:

Das Intervall aus vorherigen Iteration wird in $|\Sigma|$ Teilintervalle aufgeteilt, deren Größe ihrer Auftrittswahrscheinlichkeit entspricht. Dazu wird die Größe des Intervalls für jedes Zeichen $a_i \in \Sigma$ mit der Wahrscheinlichkeit P_{a_i} multipliziert. Das Produkt legt dann die Größe des Teilintervalls fest, welches den bisher eingelesenen Teil des Eingabewortes kodiert.

Es muss für eine eindeutige Kodierung eine feste Reihenfolge der Zeichen vorliegen. Zum Abschluss einer Iteration wird das Teilintervall ausgewählt, welches dem einzulesenden Zeichen des Eingabewortes entspricht. Die Grenzen dieses Teilintervalls stellen die Grenzen für das Intervall in der nächsten Iteration dar.

Beispiel Als Beispiel soll das Wort IEEE im dezimalen Zahlensystem kodiert werden. Die Auftrittswahrscheinlichkeit für den Buchstaben I ist trivialerweise $P_I = 0.25$, die Auftrittswahrscheinlichkeit für das Zeichen E ist $P_E = 0.75$. N sei in diesem Beispiel als 256 gewählt und I soll im unteren, sowie E im oberen Bereich kodiert werden.

In der ersten Iteration wird zunächst die Größe des Intervalls [0,256), also 256, mit der Wahrscheinlichkeit P_I multipliziert. Es ist $256 \cdot 0.25 = 64$. Folglich ist das Intervall, welches den Buchstaben I kodiert, [0,64). Anschließend wird die Größe des Intervalls mit $P_E = 0.75$ multipliziert, also $256 \cdot 0.75 = 192$. Das Intervall, welches E kodiert ist demnach [64,256). Da das erste zu kodierende Zeichen I ist, wird das untere Teilintervall ausgewählt und als Intervall für die nächste Iteration verwendet. Die weiteren Iterationen sind analog zur ersten Iteration und der nachfolgenden Grafik zu entnehmen.



In der letzten Iteration wird für das Eingabewort IEEE schließlich das Intervall [37,64) bestimmt. Jede Zahl aus diesem Intervall ist dann eine eindeutige Kodierung des Wortes IEEE.

Das Verfahren im binären Zahlensystem ist dem oben gezeigten Beispiel recht ähnlich. Da das binäre Alphabet nur aus den Zeichen 0 und 1 besteht, reicht eine Variable für die Wahrscheinlichkeit aus. *prob* sei die Wahrscheinlichkeit dafür, dass das eingelesene Zeichen Eins ist und $1 - prob$ die Wahrscheinlichkeit dafür dass eine Null eingelesen wird. In der Implementierung ist *prob* keine Gleitkommazahl, sondern ein Integerwert. Daher muss eine maximale Wahrscheinlichkeit MAX_PROB definiert werden, welche der Wahrscheinlichkeit 1 entspricht. Die Wahrscheinlichkeit berechnet sich dann aus $\frac{prob}{MAX\_PROB}$.

Nullen sollen im unteren Teilintervall und Einsen im oberen Teilintervall kodiert werden.

Die untere Grenze des Intervalls einer Iteration wird durch die 33 Bit lange Variable *low* implementiert. Außerdem wird eine Variable für die Größe des Intervalls benötigt. Diese heißt *range* und ist 4 Byte groß. *bound* ist die Grenze zwischen dem unteren und dem oberen Teilintervall. *bound* ist relativ zu *low*, die absolute Grenze berechnet sich folglich durch $low + bound$. Die Obergrenze des Intervalls kann durch $low + range$ berechnet werden.

In LZMA wird ein adaptiver Bereichskodierer [4] verwendet, der die Wahrscheinlichkeit nach dem Kodieren jedes Bits nach oben oder unten korrigiert. Wird eine Null eingelesen, so ist die Wahrscheinlichkeit dafür dass im weiteren Verlauf eine Eins auftritt nun größer - unter der Annahme dass die verwendete Wahrscheinlichkeit die zu kodierenden Zeichen korrekt vorhersagt. Die Korrektur der Wahrscheinlichkeit geht mit dem Faktor $\frac{1}{32}$ in die gesamte Wahrscheinlichkeit ein.

Im Algorithmus wird nun bitweise die Funktion zum Kodieren mit den Parametern *prob* und *bitToCode* aufgerufen. Die Funktion ist im Folgenden in Pseudocode dargestellt:

```
bound = (range >> 11) * prob;
if(bitToCode == 0){
        range = bound;
        prob = prob + ((MAX_PROB - prob) >> 5);
}
if(bitToCode == 1){
        range = range - bound;
        low   = low + bound;
        prob = prob - (prob >> 5);
}
if(range <= 0x00FFFFFF){
        normalisiere();
```

}

Nach 5 Normalisierungen terminiert diese Funktion. Als Code wird die Untergrenze des Intervalls, also die Variable *low* gewählt.

Die Normalisierung wird immer dann aufgerufen, wenn das verbleibende Restintervall so klein ist, dass im obersten Byte von *range* nur noch Nullen stehen. Ist dies der Fall, so ist das obere Byte der Untergrenze identisch mit dem oberen Byte der Obergrenze des Intervalls. Für die weitere Kodierung spielt es also keine Rolle mehr und kann herausgeschoben werden. Außerdem kann der Bereich wieder vergrößert werden, da das obere Byte bereits ausgegeben wurde. Genau dies passiert in der Normalisierung.

Außerdem muss geprüft werden, ob das obere Byte der Untergrenze *low* aus Einsen besteht. Ist dies der Fall, so kann beim nächsten Aufruf der Normalisierung ein Überlauf auftreten. Daher wird zunächst in *cacheSize* die Anzahl der Bytes, die bei einem Überlauf betroffen wären, gespeichert. Ist das obere Byte von *low* kleiner als 0xFF, so kann kein Überlauf mehr auftreten. Dann wird das neuste Byte in die Ausgabe geschrieben und die Anzahl der Bytes mit Wert 0xFF, die in *cacheSize* zwischengespeichert waren.

Falls ein Überlauf auftritt, also wenn $low > 2^{32}$, so tritt ein Überlauf auf. Das aktuelle Byte wird um Eins inkrementiert und ausgegeben. Anschließend wird die in *cacheSize* gespeicherte Zahl an betroffenen Bytes ausgegeben, aufgrund des Überlaufes diesmal allerdings mit dem Wert 0x00. Normalisieren:

```
 if(low < 0xFF000000){
        output(cache);
        output1Bytes(cacheSize-1);
     cache = sel24To31OfLow();
     cacheSize = 0;
}
 if(low >  0xFFFFFFFF){
        output(cache+1);
        output0Bytes(cacheSize-1);
        cache = (cache ^ 0xFF000000);
        cacheSize = 0;
}
cacheSize++;
low = (low ^ 0x00FFFFFF) << 8;
range = range << 8;
```

Beispiel Es soll das Byte $(10110001)_2$ kodiert werden. Dazu werden *low* mit 0x00000000, range mit 0xFFFFFFFF und prob mit 0x400 initialisiert, sowie MAX_PROB als 0x800 definiert.Die initiale Wahrscheinlichkeit beträgt also 0.5.

Zunächst muss das erste Bit der Eingabedaten, eine Eins, kodiert werden. Die berechnete Grenze ist *bound* = 0x7FFFFC00. Da die Eins im oberen Intervall kodiert wird, muss die Untergrenze *low* mit dem Wert von *bound* überschrieben werden, und das Intervall hat die neue Größe *range* − *bound* = 0x800003FF. Die Wahrscheinlichkeit muss nach unten korrigiert werden und beträgt dann *prob* = 0x3E0.

Als nächstes wird die folgende Null kodiert. Die neu berechnete Grenze *bound* ist 0x3E000000. Nullen werden im unteren Teilintervall kodiert, also bleibt *low* unverändert. Es wird lediglich das obere Teilintervall "abgeschnitten", indem *range* = *bound* gesetzt wird. *prob* wird nach oben zu 0x401 korrigiert. Ohne Rundungsfehler würden sich wieder der Initialwert 0x400 ergeben.

Die Beschreibung für die darauf folgenden fünf Bits wird an dieser Stelle übersprungen, das Verfahren ist analog zu den ersten beiden hier gezeigten Fällen.

Für das letze Bit ergibt sich ein Intervall *low* = 0xAEFD487F und *range* = 0x00E1EF2B. Da somit *range* ≤ 0x00FFFFFF, muss normalisiert werden. Da *low* aber weder kleiner als 0xFF000000, noch größer als 0xFFFFFFFF, wird noch kein Byte in den Ausgabedatenstrom geschrieben. (Dies wäre nach dem ersten Byte auch nicht wünschenswert, weil sonst keine Kompression stattgefunden hätte.) Es wird also nur die *cacheSize* erhöht, sowie der Bereich vergrößert und das nicht mehr benötigte obere Byte von *low* gelöscht und *low* um ein Byte nach links geschoben. Falls an dieser Stelle ein Übertrag in *low* aufgetreten wäre, so würde dieser ebenfalls gelöscht. Nach diesem Verfahren wird nun weiter verfahren, bis die Normalisierung fünf Mal ausgeführt wurde. Danach werden *low* und *range* wieder auf ihre initialen Werte resettet.

1.2.2 Dekodieren

Das Dekodieren funktioniert im Bereichskodierer ähnlich wie das Kodieren. Die Berechnung der Grenze *bound* sowie die adaptive Anpassung der Wahrscheinlichkeit *prob* geschehen analog zum Kodierungsvorgang. Das einzulesende Wort *code* wird jeweils mit der neu berechneten Grenze verglichen. Anhand dessen kann bestimmt werden, ob sich *code* im oberen oder im unteren Teil des Intervalls befindet und das zugehörige Bit kann mit dem Rückgabewert *returnValue* konkateniert werden.

Da die Ausgabe des Kodierens die Untergrenze *low* des Intervalls war, muss *code* auch die Untergrenze des Zielintervalls sein. Ist *code* im oberen Teilintervall, so wird demnach die Grenze *bound* von *code* abgezogen.

Bei der Kodierung wird die Normalisierung immer aufgerufen, wenn das oberste Byte des Bereichs *range* leer ist. Dies muss bei der Dekodierung auch analog geschehen, da sie sonst andere Ergebnisse liefern würde.

```
bound = (range >> 11) * prob);
```

```
if(code<bound){
        range = bound;
        prob = prob + ((MAX_PROB - prob) >> 5);
        returnValue = returnValue + '0';
}
else{
            range = range - bound;
        code  = code - bound;
        prob = prob - (prob >> 5);
        returnValue = returnValue + '1';
}
if(range < 0x01000000){
        normalisiere();
}
```

Die Normalisierung beim Dekodieren ist erheblich kürzer, da nicht auf Überläufe geprüft werden muss. Bei jedem Aufruf der Normalisierung wird also schlicht der Bereich *range* vergrößert und das oberste Byte von *code* herausgeschoben. Dieses kann herausgeschoben werden, da das oberste Byte schon im Intervall gespeichert ist. Folglich wird ein neues Byte des Inputstreams als unterstes Byte von *code* eingelesen. Die Normalisierung funktioniert demnach folgendermaßen:

```
range = range << 8;
code  = code<< 8;
code  = code + readByte();
```

Auch das Dekodieren wird nach fünf Normalisierungen unterbrochen und die Werte werden wieder auf ihre Anfangswerte zurückgesetzt.

2 LZMA

In der Literatur wird LZMA (vgl. [2]) meist etwa folgendermaßen beschrieben:

> LZMA ist ein Variante von LZ77 mit großem Wörterbuch, deren Ausgabe mit einem Bereichskodierer kodiert wird.

Dies legt zwar die grundlegende Funktionsweise dar, der Algorithmus bietet aber noch viele weitere Verbesserungen gegenüber vorherigen Varianten von LZ77. Die Verbesserung betreffen unter anderem die Kodierung von gefundenen Matches, den Aufbau des Wörterbuches und die Entscheidung, welcher Match ausgegeben wird.

2.1 Pakettypen

Bei LZ77 gibt es zwei verschiedene Pakettypen. Dies sind Literale, also Zeichenketten, die bisher noch nicht im Vorschaupuffer aufgetreten sind sowie Matches, die Verweise auf eine bereits eingelesene Zeichenkette darstellen. Beide werden im Grunde identisch abgespeichert.

Bei LZMA gibt es sieben verschiedene Pakettypen, die sich aus unterschiedlichen Teilen zusammensetzen. Jeder Pakettyp beginnt mit einem präfixfreien Bitcode, da die Länge einzelner Pakettypen unterschiedlich ist.

Im Gegensatz zu LZ77, wo Literale als Tripel $(0, 0, lit)$ für ein Literal *lit* abgespeichert werden, speichert LZMA nur eine 0, konkateniert mit dem Bytecode des eingelesenen Zeichens *lit*. Für Matches benötigt LZMA daher kein fehlerverursachendes Zeichen (siehe Abschnitt 1.1), sondern speichert nur das Präfix 10, gefolgt von der Länge *len* und Distanz *dist*.

Zudem bietet LZMA als weitere Verbesserung noch einen Schnellzugriff auf einen Match, der identisch zum letzten gefundenen Match ist (*Short Repetition*). Dieser besteht ausschließlich aus einem vier Bit langen Code.

Außerdem speichert LZMA die Distanzen der letzten vier Matches, sodass es auch einen Schnellzugriff (*Repetition*) auf die letzten vier Matches gibt, bei denen nur deren Länge angegeben werden muss. Die Länge des Matches einer *Repetition* kann sich also von der Länge des ursprünglichen Matches unterscheiden. Es folgt eine Tabelle mit den Namen, Kodierungen und Bedeutungen der einzelnen Pakettypen:

Kodierung	**Name**	**Bedeutung**
0 + byteCode	LIT	1 Byte Literal
10 + length+dist	MATCH	LZ77 Sequenz
1100	SREP	Exakte Wiederholung des letzten Matches
1101 + length	REP[0]	Distanz wie letzter Match
1110 + length	REP[1]	Distanz wie vorletzter Match
11110 + length	REP[2]	Distanz wie drittletzter Match
11111 + length	REP[3]	Distanz wie viertletzter Match

Länge und Distanz eines Matches werden präfixfrei kodiert, um kürzere Codes für kürzere Längen bzw. Distanzen zu erhalten. Die Kodierung der Längen erfolgt nach folgendem Prinzip:
Eine binäre 0 gefolgt von 3 Bits kodiert die Längen 2 bis 9, eine binäre 10 gefolgt von 3 Bits Längen von 10 bis 17. Distanzen zwischen 19 und 273 werden durch die beiden Bits 11, gefolgt von 8 Bits kodiert. Ein ähnliches Verfahren kodiert die Distanzen.

Es gibt jedoch nicht nur Verbesserungen in der Kodierung einzelner Pakete. Auch das Textfeld (bzw. Wörterbuch) wurde angepasst.

2.2 Wörterbuch

Da in LZMA die Wörterbücher üblicherweise relativ groß sind, werden Datenstrukturen zur effektiven und schnellen Suche nach Matches benötigt. In LZMA gibt es zwei Möglichkeiten, wie auf eine Übereinstimmung mit einer bereits eingelesenen Zeichenkette überprüft wird. Dies funktioniert entweder mit verketteten Listen oder mit Hilfe von binären Bäumen. Beide Varianten verfahren relativ ähnlich.

Sie durchlaufen den Vorschaupuffer Byte für Byte von links nach rechts. Es werden nun (parameterabhängig) $n \in \{2, 3, 4\}$ Bytes, im weiteren als w_j bezeichnet, in einem Zug mit einer Hashfunktion gehasht. Die Ausgabe der Hashfunktion $hash : w_j \rightarrow i$ ist nun der Index i in einem Array $array[i]$. Die Länge dieses Arrays ist gleich der Größe des Wertebereichs der Hashfunktion, nämlich 2^{8n}.

Nun verfahren beide Varianten unterschiedlich. Wenn das Prüfen auf Matches mit verketteten Listen geschehen sollen, steht an $array[i]$ je eine verkettete Liste. Das Gleiche gilt analog für binäre Bäume.

2.2.1 Mit verketteter Liste

Bei der verketteten Liste wird ein neu gefundener Match immer an erster Position eingefügt, so dass mit höherer Position in der Liste die Distanz immer weiter steigt. Wenn nun eine Bytefolge w_j gehasht wurde und im Array an Stelle $array[hash(w_j)]$ eine leere Liste steht, muss zwangsweise ein Literal ausgegeben werden. Wenn an dieser Stelle allerdings eine nicht-leere Liste steht, wird diese von vorne nach hinten durchlaufen. Der optimale Match wird dann ausgegeben und weiterkodiert. "Optimal" bedeutet an dieser Stelle nicht unbedingt ein Match mit maximaler Länge, sondern derjenige, der sich am kürzesten kodieren lässt. So wird zum Beispiel ein längerer Match nicht ausgewählt, wenn ein kürzerer Match eine weitaus geringere Distanz aufweist, welche sich kürzer kodieren lässt.

In der Praxis werden nicht alle Matches überprüft, sondern nur eine parameterbestimmte Anzahl (Default: 24), als Kompromiss zwischen optimaler Kodierbarkeit und Laufzeit.

2.2.2 Mit binärem Suchbaum

Das Verfahren mit binären Bäumen ist recht ähnlich. Jeder binäre Baum an einer Position im Array muss zwei Eigenschaften erfüllen.

1. Gültiger binärer Suchbaum bzgl. der lexikographischen Ordnung

2. Gültiger Max-Heap bzgl. der Wörterbuchposition

Damit das Verfahren mit binären Suchbäumen wirklich einen Vorteil bietet, muss die Suchzeit zum Auffinden von Matches minimiert werden. Daher werden die Referenzen auf bereits gefundene Matches im binären

Suchbaum lexikographisch angeordnet. Die Sortierung geht auch über die Grenzen der gehashten Bytes hinaus, es werden also auch die folgenden Bytes mit einbezogen.

Die Erhaltung eines Max-Heaps bezüglich der Wörterbuchposition geschieht durch das Einfügen eines neuen Elementes als neue Wurzel des Binärbaums. Dies kann mit maximal einer Rotation des Restbaumes umgesetzt werden und ist damit hinreichend effizient. Durch den Max-Heap ist gewährleistet, dass neuere Matches schneller gefunden werden. Das ist wichtig, weil diese mit großer Wahrscheinlichkeit kürzer kodiert werden können.

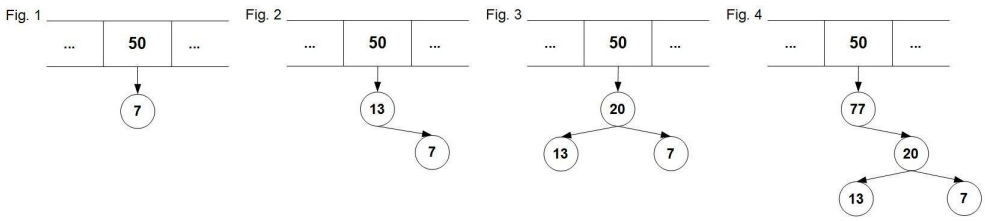
Trotzdem ist offensichtlich, dass dieser Baum nicht immer perfekt balanciert ist, im Worst-Case ergibt sich sogar eine lineare Liste. Eine bessere Balancierung ist aber zu teuer in der Laufzeit, daher wird in LZMA darauf verzichtet.

Beispiel Nun folgt ein Beispiel für den Aufbau eines Binärbaums, welches an das Beispiel aus Kapitel 6.26 des *Handbook of Data Compression* [2] angelehnt ist. Es sei der Parameter n als 2 gewählt, also werden je zwei Bytes in einem Zug gehasht. Die beiden Bytes, die im Beispiel betrachtet werden sollen, sind *ba*. Außerdem wird angenommen, dass die Hashfunktion für diese beiden Bytes den Wert 50 zurückgibt, also $hash(ba) = 50$. Gegeben sei folgender Vorschaupuffer:

Index	1	2	3	...	7	8	9	...	13	14	15	16	...	20	21	22	...	77	78	79	80	...
Wert	x	y	z	...	b	a	d	...	b	a	b	x	...	b	a	c	...	b	a	b	d	...

Es würden nun zunächst die ersten beiden Bytes *xy* gehasht, und als nächstes die Bytes *yz*. In diesem Beispiel wird jedoch auf alle Bytes außer *ba* nicht genauer eingegangen.

Schließlich werden auch die Bytes *ba* mit Index 7 und 8 gehasht. Aus den Annahmen ist ersichtlich, dass diese auf den Wert 50 gehasht werden. An Stelle 50 des Arrays *array* steht, da vorher die Bytes *ba* noch nicht im Vorschaupuffer eingelesen wurden, ein leerer Baum. Nun muss ein Verweis auf den ersten Index der Bytes in den Baum als neue Wurzel eingefügt werden. Da es das erste Vorkommen der Bytes *ba* ist, muss ein Literal ausgegeben werden. Diese Situation wird durch Fig.1 verdeutlicht.



Das nächste Vorkommen der Bytes *ba* ist an Stelle 13. An der Stelle im Hasharray steht der zuvor eingefügte Verweis auf den Index 7. Der neue Match

muss wieder als Wurzel eingefügt werden. 7 ist der rechte Nachfolger von 13, da an Index 7 auf die Bytes *ba* ein *d* folgt, welches lexikographisch größer ist als das an Stelle 13 folgende *bx* (vgl. Fig. 2). Da sich das *d* und *bx* aber nicht matchen lassen, hat der maximale (und bisher einzige) Match die Länge 2.

Nach einiger Zeit hasht der Algorithmus an Stelle 20 wieder die Bytes *ba*. Es wird eine Referenz auf diese Stelle abgespeichert und in den binären Suchbaum eingefügt. Da der Match an Stelle 7 nun die gleiche Länge hat wie derjenige an Stelle 13, aber weiter von der aktuellen Position entfernt ist, würde der Match zur Stelle 13 ausgegeben.

In Abb. 4 ist zu sehen, wie der nächste Match des Bytes *ba* an Stelle 77 in den Baum eingefügt wird. Der Match zur Stelle 13 im dargestellten Vorschaupuffer hat die Länge 3 ('bab') und würde daher wahrscheinlich ausgewählt. In diesem Fall würde das Hashen der Bytes ab 78 übersprungen, da diese bereits vollständig gematcht wurden.

Früher gab es neben binären Suchbäumen und verketteter Liste noch die Möglichkeit, PATRICIA-Tries zur Speicherung des Wörterbuches zu verwenden. Diese erwiesen sich jedoch in der Praxis für LZMA als ineffizienter als binäre Suchbäume und sie werden daher nicht mehr unterstützt.

Der Unterschied eines PATRICIA-Tries zu einem normalen Trie liegt im Speichervermögen eines Knotens. Im normalen Trie kann nur ein Zeichen pro Knoten abgespeichert werden kann. Dies ist bei langen Wörtern, von denen große Teile gleich sind, unpraktisch und ineffizient, weswegen bei PATRICIA-Tries in einem Knoten mehrere Zeichen abgespeichert werden können.

Siehe den Artikel *PATRICIA-Practical Algorithm To Retrieve Information Coded in Alphanumeric* von Donald Morrison [5] für weitere Informationen über PATRICIA-Tries.

2.3 Kontextbezogene Kodierung

Die dem Bereichskodierer übergebenen Wahrscheinlichkeiten werden in LZMA immer aus dem Kontext zu jedem Bit bestimmt. So wird verhindert, dass aufeinanderfolgende Bits, die nicht zueinander gehören, die Wahrscheinlichkeiten ändern. So hat etwa die Länge eines Matches nichts mit dessen Distanz zu tun. Aber die Länge von zwei aufeinander folgenden Matches kann gleich sein, beispielsweise beim Einlesen eines Arrays von einem Datentyp mit fester Länge.

Dazu macht LZMA eine wahrscheinlichkeitsbasierte Vorhersage über das nächste folgende Paket. Das wird mit einem Transitionssystem realisiert. Es gibt 12 Zustände, mit denen jeweils Aussagen über die vorherigen Pakete getroffen werden können. Der Zustand wird nach dem Kodieren jedes einzelnen Paketes aktualisiert. Die Übergangsfunktion des Transitionssystems ist in folgender Tabelle dargestellt.

Zustand	LIT	MATCH	REP[*]	SREP
0	0	7	8	9
1	0	7	8	9
2	0	7	8	9
3	0	7	8	9
4	1	7	8	9
5	2	7	8	9
6	3	7	8	9
7	4	10	11	11
8	5	10	11	11
9	6	10	11	11
10	4	10	11	11
11	5	10	11	11

Beispiel Als Beispiel für die Aussage, die abhängig vom aktuellen Zustand über bereits eingelesene Pakete getroffen werden können, wird diese hier exemplarisch für den Zustand 9 bestimmt. Ist der aktuelle Zustand 9, muss das vorherige Paket ein SREP sein, da das die einzige Möglichkeit darstellt, in den Zustand 9 zu gelangen. Genauer gesagt muss der vorherige Zustand zwischen 0 und 6 sein, da ein SREP in den Zuständen 7 bis 11 in den Zustand 11 geführt hätte. In die Zustände 0 bis 6 wiederum gelangt man nur, wenn das Zeichen davor ein Literal war. Aussagen über Pakete davor können aber nicht getroffen werden.

Interpertiert man die Übergänge als Wahrscheinlichkeiten für das Auftreten eines Paketes der jeweiligen Art, so ist das Transitionssystem eine *Markov Chain*, welche namensgebend für LZMA ist.

Sinn dieser wahrscheinlichkeitsbasierten Vorhersage ist die Auswahl eines geeigneten Kontextes, der die dem Bereichskodierer übergebenen Wahrscheinlichkeiten *prob* für jedes Bit festlegt.

2.4 LZMA2

Bei schlecht komprimierbaren oder gar unkomprimierbaren Daten, wie etwa bereits mit anderen Algorithmen komprimierten Daten, verhält sich LZMA nicht besonders effizient. Da diese Daten dicht an ihrer Entropie sind, ist es nahezu unmöglich, weitere Matches zu finden. Daher werden solche Daten, wenn man sie mit LZMA komprimiert noch größer als sie ursprünglich waren. Abhilfe schafft hier eine Weiterentwicklung der ersten Version von LZMA. 2009 verbesserte Igor Pavlov seine erste Version von LZMA zu einem neuen Format mit dem Namen LZMA2.

Im Gegensatz zu LZMA sind die Daten in LZMA2 in Blöcke (auch *Chunks*) unterteilt. Ein Block hat typischerweise eine Größe im Kilobyte-

bis Megabytebereich und umfasst somit eine Vielzahl an Paketen. LZMA2 komprimiert nun jeden Block mit LZMA und vergleicht die Größe des komprimierten und des unkomprimierten Blocks. Der kleinere von beiden wird in die Ausgabe geschrieben.

Dies hat den Vorteil, dass die Eingabedaten im Worst-Case nun nur um die Summe der Blockheader vergrößert wird. Es bietet aber außerdem den Vorteil, dass die Blöcke weitestgehend unabhängig voneinander behandelt werden können. LZMA2 ist daher auch multithreadingfähig. Dabei wird zwischen allen Threads ein gemeinsames geteiltes Wörterbuch verwendet.

3 Dateiformate

Seit der Veröffentlichung von LZMA hat es nicht viele Dateiformate gegeben, die LZMA implementieren. Das erste Format war 7z, welches von Igor Pavlov alleine entwickelt wurde. Da da die Programme, die 7z kodieren können, aber zunächst ausschließlich für Windows existierten, u.a. weil Igor Pavlov seine Implementierung in der Software 7Zip nur für Windows geschrieben hat, gab es einige Zeit später erste Portierungen für Linux.

7z bietet neben der Kompression von Daten auch deren Archivierung. Dies empfanden einige User als ein unnötiges Feature, weil sie dies getrennt mit einem Archivierungsprogramm (z.B. *tar*) erledigen möchten. Daher wurde zunächst ein Format mit der Dateiendung *.lzma* entwickelt, dass einen reinen Datenstrom von LZMA Paketen ausgab ohne jegliche Header zu benötigen. Dies erwies sich jedoch als unkomfortabel, besonders im Zusammenhang mit LZMA2. Daher wurde von Lasse Collin das Format xz entwickelt, welches kompaktere Header hat und ausschließlich LZMA2 komprimiert.

3.1 7z

7z ist das erste Dateiformat, welches den LZMA Algorithmus implementiert. Es wurde, wie LZMA selber, von Igor Pavlov entwickelt und erstmals zusammen mit der von ihm geschriebenen Software 7Zip im Jahre 2000 verwendet. Die Standardkompressionsmethode von 7z ist LZMA. Es werden aber auch LZMA2, die *Burrows-Wheeler-Transformation*, *Prediction By Partial Matching* und *Deflate* unterstützt.

Außerdem werden die Vorfilter BCJ und BCJ2 unterstützt, welche in ausführbaren Dateien bereits vor der Kompression mehr Redundanz in den Daten schaffen.

Darüber hinaus bietet 7z noch eine Verschlüsselung der Daten mit AES-256 sowie die bereits erwähnte Archivierungsfunktion.

Als Parameter von 7z können viele Einstellungen für den Lempel-Ziv-Markov Chain Algorithmus getroffen werden. Eine Auswahl davon wird im folgenden Teil erläutert.

Mit dem Parameter **-d**[N] kann die Größe des Wörterbuchs festgelegt werden. Für ein $N \in \{0, 1, \ldots, 30\}$ ist die Wörterbuchgröße jeweils 2^N. Standardmäßig ist $N = 27$ ausgewählt, die maximale Größe ist 4GByte.

-mf[ID] legt den *Match Finder* für LZMA fest. ID kann die Werte $bt2$, $bt3$, $bt4$ und $hc4$ annehmen. Die Zahl steht jeweils für die Zahl der Bytes, die in einem Zug gehasht werden. *bt* steht für *Binary Tree* und *hc* steht für *Hash Chain*. Ohne spezielle Angabe eines Parameters werden 4 Bytes gehasht und in binäre Suchbäume eingefügt.

Mit **-a**[N] kann der Kompressionsmodus gewählt werden. Es gibt einen schnellen, einen normalen und einen maximalen Modus. Diese werden in selbiger Reihenfolge durch $N \in 0, 1, 2$ repräsentiert. Der Kompressionsmodus verändert eine Vielzahl an Parametern. Beispielsweise wird im schnellen Modus als *Match Finder* eine Hashkette anstelle des binären Suchbaums verwendet. Als Defaultwert für den Kompressionsmodus wird der maximale Modus gewählt.

3.2 xz

xz ist ein Format zur Kompression einzelner Dateien mit LZMA2, unterstützt aber keine weiteren Kompressionsalgorithmen oder Verschlüsselung. Es kann eine CRC32 Checksumme berechnet werden und es können BCJ bzw. BCJ2 Filter angewendet werden. Sowohl die für Linux entwickelten xz-Utils, als auch 7Zip und andere Kompressionsprogramme können xz-Dateien (de-) komprimieren.

Unter dem Namen "xz-Embedded"[6] gibt es eine abgespeckte Version von xz-Utils, welche speziell für eingebettete Systeme entwickelt wurde. Sie besteht nur aus einem Dekompressor, und ist in kompilierter Form weniger als 20 KB groß. xz-Embedded benötigt im Single-Call Mode nicht das gesamte Wörterbuch im Speicher, sondern nur alle Bytes vom aktuell bearbeiteten bis zu dem mit der längstmöglichen kodierbaren Distanz. Im Multitasking-Modus ist es unvermeidbar das gesamte Wörterbuch in den Speicher zu laden. Aufgrund der hohen Kompressionsrate zu vergleichbar akzeptabler Laufzeit werden der Linux-Kernel und die Linux-Core-Utils auch im xz-Format angeboten. Dazu steht im Linux-Kernel eine Version von xz-Embedded zur Verfügung.

4 Kompressionsratenvergleich

Abschließend noch ein Vergleich der Kompressionsraten von 7z (unter Verwendung von LZMA) mit zwei anderen verbreiteten Kompressionsverfahren, zip und bzip2. Zip verwendet Deflate, eine Wörterbuchkompression, welche auf LZ77 aufbaut und die Ausgabe anschließend mit Huffmankodierung kodiert. Für weitere Informationen siehe Kapitel 6.25 im *Handbook of Data Compression* von David Salomon und Giovanni Motta [2]. Bzip2 verwendet

die Burrows-Wheeler-Transformation, welche die Eingabedaten zunächst geschickt umsortiert und danach mit arithmetischer Kodierung kodiert. Die Burrows-Wheeler-Transformation wird u.a. in Kapitel 11.1 des *Handbook of Data Compression* von David Salomon und Giovanni Motta [2] erläutert.

Als Vergleichsdaten dienen die deutsche Ausgabe von Faust von Johann Wolfgang von Goethe [7], sowie eine Bitmap-Datei des 7Zip Logos [8] (von png zu bmp konvertiert, Auflösung 256 x 256 Pixel), eine Quellcodedatei in der Programmiersprache C (Progc aus dem *Calgary Corpus* [9]), einem kompilierten Objekt (Obj2 ebenfalls aus dem *Calgary Corpus*) und eine Datei mit zufälligen Daten, welche dem atmosphärischen Rauschen entnommen wurden [10].
Es wurden jeweils die Standardeinstellungen für die Parameter genutzt.

	Unkompr.	.7z (LZMA)	.zip (Deflate)	.bzip2 (BWT)
Text:	222 kB	75.9 kB	82.1 kB	69 kB
Bild:	192 kB	7.52 kB	11.6 kB	10 kB
Progc:	38.6 kB	12.3 kB	12.2 kB	12.8 kB
Obj2:	241 kB	60.3 kB	74.6 kB	77.2 kB
zufällig:	16 kB	16.3 kB	16.1 kB	16.4 kB

LZMA erreicht in den hier getesteten Dateien im Vergleich zu den verglichenen Algorithmen im Schnitt das beste Ergebnis. Die Größe der 7z Dateien ist aber immer ziemlich nah an den besten im Test erreichten Ergebnissen.

Allgemein lässt sich sagen, dass LZMA einen Vorteil bei großen Dateien hat, in denen Redundanzen in großem Abstand zueinander auftreten, weil LZMA mit Abstand das größte Wörterbuch benutzt. Dies hat natürlich den hohen Speicherbedarf beim Komprimieren sowie eine längere Laufzeit beim Finden von Matches zur Folge.

Literatur

[1] Lzma sdk. http://www.7-zip.org/sdk.html, 23.05.2012.

[2] David Salomon and Giovanni Motta. *Handbook of Data Compression (5. ed.)*. Springer, 2010.

[3] Glen G. Langdon. An introduction to arithmetic coding. *IBM J. Res. Dev.*, 28(2):135–149, March 1984.

[4] Xz for java. http://tukaani.org/xz/java.html, 23.05.2012.

[5] Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[6] Xz embedded. http://tukaani.org/xz/embedded.html, 23.05.2012.

[7] J.W. von Goethe. Faust. http://www.gutenberg.org/files/21000/21000-0.txt, 1808.

[8] 7zip logo. http://www.articlevoid.com/wp-content/uploads/7zip.png?9707a5, 23.05.2012.

[9] Calgary corpus. http://corpus.canterbury.ac.nz/descriptions/, 23.05.2012.

[10] Zufallszahlen. http://www.random.org/bytes/, 23.05.2012.