Kompression von Integern

Thomas Leichtle

Universität Ulm
Institut für theoretische Informatik
89069 Ulm, Germany
Thomas.Leichtle@uni-ulm.de

Zusammenfassung Diese Arbeit soll einen kleinen Überblick über verschiedene Kodierungsverfahren für die verlustfreie Kompression von positiven Ganzzahlen geben. Dazu werden die Verfahren beschrieben, klassifiziert, analysiert und verglichen. Zum Schluss werden mögliche Anwendungsgebiete genannt.

1 Motivation

Die Binärdarstellung einer natürlichen Zahl x hat die Länge $\lfloor \log_2 x \rfloor + 1$. In einem Datenstrom reichen diese Bits jedoch nicht aus, um die Zahl eindeutig wiederherzustellen. Ein kleines Beispiel: Dekodiere 11001110. Als eine Zahl betrachtet, erhält man in Dezimaldarstellung 206. Da die Anzahl der enthaltenen Zahlen jedoch nicht bekannt ist, wäre jede mögliche Zerlegung denkbar. 1100 & 1110, 11 & 00 & 1 & 1 & 10, ... Allein für diese Zahl gibt es 128 Kombinationen. Allgemein gibt es für eine Zahl der Länge n, wenn führende Nullen in der Darstellung erlaubt sind, 2^{n-1} Möglichkeiten sie zu zerlegen.
Eine Möglichkeit für eine eindeutige Darstellung wäre eine feste Länge zu wählen und kleinere Zahlen mit führenden 0en aufzufüllen. Dabei treten jedoch zwei Probleme auf. Das Erste ist, dass die größte darstellbare Zahl nach der Wahl der Länge festgelegt ist. Sollte eine größere Zahl auftauchen, so kann diese nicht kodiert werden. Wählt man die Länge sehr groß um diesem Problem entgegen zu wirken, so tritt das zweite Problem auf. Es werden sehr viele führenden Nullen auftauchen, die ebenfalls gespeichert/übertragen/verarbeitet werden müssen, was die Effizienz reduziert. Deshalb wurden einige weitere Möglichkeiten und Verfahren entwickelt von denen hier einige vorgestellt werden.

2 Codeklassen, Definitionen & Eigenschaften

Zuerst werden einige Definitionen und Eigenschaften genannt, welche helfen sollen Codes einzuordnen und zu bewerten. Auch Schranken für die Codewortlänge werden gegeben.

2.1 Shannon-Entropie

Claude Shannon definierte 1948 den Begriff der Entropie in der Informationstheorie. Die Entropie $H(P)$ ist ein Maß für den Informationsgehalt oder auch der Unsicherheit einer Wahrscheinlichkeitsverteilung P.

$$H(P) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Das Maximum $H(P) = \log_2 N$ wird für eine Gleichverteilung mit N Ereignissen und das Minimum $H(P) = 0$ für eine Verteilung mit einem sicheren Ereignis, d.h. $p_j = 1$, angenommen. Es kann gezeigt werden, dass kein eindeutig dekodierbarer Code erzeugt werden kann, der eine geringere erwartete Länge hat als $H(P)$. Formal: $H(P) \leq E(C, P)$, wobei $E(C, P)$ der Erwartungswert (= die erwartete Länge der Codeworte) für die Kodierung einer Quelle mit Verteilung P unter Verwendung von Code C ist. Erfüllt ein Code die Gleichheit, so spricht man von einem *zero-redundancy* Code [MT02].

2.2 Kraft-MacMillan Ungleichung

Eine weitere Schranke für die Codewortlänge gibt die Kraft-MacMillan Ungleichung. Diese beantwortet die Frage, wie kurz die Codewörter eines Codes sein dürfen, dass sie noch eindeutig dekodierbar sind (vgl. [Say00]). Sei C ein Code mit N Codewörtern der Länge l_1, l_2, ..., l_N. C ist genau dann eindeutig dekodierbar, wenn gilt:

$$K(C) = \sum_{i=1}^{N} 2^{-l_i} \leq 1$$

Umso länger die Codeworte sind, umso kleiner werden deren Beiträge zur Summe. Da nur die Symbole 0 & 1 erlaubt sind, wird mit 2^{-l_i} gearbeitet. Bei mehreren Symbolen müsste man dies entsprechend anpassen.

MacMillan erweiterte diese Aussage und zeigte, dass jeder Code, der diese Ungleichung erfüllt, in einen präfixfreien Code gleicher Länge umgeschrieben werden kann [Say00].

2.3 Präfixfreier Code

Ein Code ist genau dann präfixfrei, wenn kein Codewort Präfix eines anderen Codeworts ist, d.h. $\forall c_i \in C \nexists c_j \in C : \ c_i = c_j\{0,1\}^+$. Eine alternative Definition kann mit Hilfe von *Codebäumen* (siehe Abb. 1) gegeben werden. Dabei entspricht die Wurzel des Baumes dem leeren Codewort und jede Verzweigung einem Symbol der Grundmenge. Meistens, wie auch in dieser Arbeit, wird mit der Grundmenge {0,1} gearbeitet, was zu binären Codebäumen führt. Die Knoten können Codewörtern entsprechen. Es gilt: Sind alle Codewörter eines Codes Blätter (äußere Knoten) im Codebaum, so ist der Code präfixfrei. Die Rückrichtung gilt ebenso [Say00]. Für den präfixfreien

Abbildung 1. Codebäume für präfixfreien Code (links) und nicht-präfixfreien Code (rechts).

Code kann so ein Baum direkt zum Dekodieren während des Empfangens verwendet werden. Man startet bei der Wurzel und folgt solange den entsprechenden Kanten bis ein Blatt erreicht ist. Dann gibt man das Codewort aus und beginnt erneut an der Wurzel. Beim nicht-präfixfreien Code kann man ein Codewort nicht direkt ausgeben, wenn man einen entsprechenden Knoten erreicht hat, da man sich nicht sicher sein kann, ob dieses Codewort gemeint war oder es nur dem Präfix eines anderen Codewortes entspricht. Man kann die Wörter erst dekodieren, wenn die komplette Nachricht übertragen wurde - falls der Code überhaupt eindeutig dekodierbar ist. Dies ist ein deutlicher Nachteil des nicht-präfixfreien Codes.

2.4 Kopflose Darstellung

Wir definieren die kopflose Darstellung einer Zahl x wie folgt: Nehme die Binärdarstellung von x und entferne das höchstwertigste Bit. Es wird lediglich mit den $\lfloor \log_2(x) \rfloor$ restlichen Bits gearbeitet. Dies ist möglich, da die Binärdarstellung aller Zahlen > 0 eine führende 1 haben. Man spart 1 Bit. Notation: $\mathrm{bin_h}(x)$.

2.5 Self-delimiting Codes

Eine Möglichkeit, Codes in Klassen zu trennen, kann anhand der Art und Weise erfolgen, wie einzelne Codeworte im Datenstrom auseinander gehalten werden (vgl. [Mac03]). Die erste Klasse ist die der self-delimiting Codes. Bei dieser Art von Code wird zuerst die Länge des Codewortes für den Integer übertragen und anschließend das Codewort selbst, wobei die Länge ebenfalls eine positive Integerzahl ist. Es besteht nun wiederum das Problem, dass diese zwei Zahlen auseinander gehalten werden müssen. Dazu muss ein Code mit 'end of file' Symbol für die Kodierung der Länge verwendet werden. Da diese Art von Codes eine schlechtere Komprimierung ermöglichen, besteht nun der Vorteil der self-delimiting Codes darin, dass die Länge im Normalfall eine deutlich kleinere Zahl ist und sich der Overhead, dass die Länge mit übertragen werden muss, durch den Gewinn der Komprimierung der Zahl ausgleicht, bzw. man insgesamt eine Einsparung erreicht.

2.6 Codes mit 'end of file' Symbolen

Bei dieser Codeklasse wird eine Bitfolge bestimmt, die das Ende jedes einzelnen Codeworts definiert. Es ist ersichtlich, dass diese Bitfolge in keinem der Codeworte selbst als Teilfolge enthalten sein darf.

2.7 Universal Codes

Betrachtet man die asymptotische Codewortlänge und klassifiziert damit, erhält man weitere Codeklassen. Eine davon ist die der Universal Codes. Diese haben mehrere, interessante Eigenschaften. Eine davon ist, dass jeder Univeral Code unbeschränkt ist, d.h. damit können beliebig große Zahlen kodiert werden. Eine andere Eigenschaft begrenzt das Wachstum der Codewortlänge in Abhängigkeit der Eingabe. Für Universal Codes gilt:

$$\text{Length}(\text{encode}_{\text{universal}}(x)) \leq c_1 \cdot (H + c_2)$$

wobei c_1 und c_2 Konstanten sind. Dies bedeutet, dass die Länge der Codeworte sich lediglich um einen konstanten Faktor und einen additiven Term von der optimalen Codewortlänge, welche von der Entropie H gegeben ist, unterscheidet. Gilt $c_1 = 1$, so nähern sich die Werte mit zunehmender Entropie immer mehr aneinander an und man sagt, dass der Code *asymptotisch optimal* ist. Als *universal* werden diese Art von Codes deshalb bezeichnet, da diese Eigenschaften für jede monoton abnehmende Wahrscheinlichkeitsverteilung gelten und die Codes somit beweisbar 'nicht so schlecht' sind [MT02].

3 Verschiedene Codes

In diesem Kapitel werden acht verschiedene Codes vorgestellt und beschrieben.

3.1 Unary Code

Der einfachste eindeutig rekonstruierbare Code für Ganzzahlen ist der Unary Code. Eine natürliche Zahl x wird als $x-1$ Nullen, gefolgt von einer Eins codiert. Alternativ wird auch oft die Codierung in $x-1$ Einsen, gefolgt von einer Null verwendet.

$$\text{encode}_{\text{unary}}(x) = 0^{x-1}1$$

Folglich ist dies der einfachste Code mit 'end of file' ('eof') Symbol. Beispiel: $\text{encode}_{\text{unary}}(13) =$ 0000000000001. Die Dekodierung erfolgt durch Zählen der Bits bis eine Eins auftritt. Dieses 'eof'-Symbol wird mitgezählt. Der Code ist optimal für eine Wahrscheinlichkeitsverteilung von $P(x) = 2^{-x}$, da er für diesen Fall identisch zu dem entsprechenden Huffman-Code ist, von welchem bekannt ist, dass er optimal ist. Die Codewortlänge für eine Zahl x ist genau x Bits lang, d.h. liegt in $O(x)$. An den genannten zwei Punkten ist zu erkennen, dass dieser Code vorallem für kleine Integer x geeignet ist. Praktische Anwedung findet er teils in anderen Codierungsverfahren, die im Folgenden erläutert werden.

3.2 Elias Gamma

Der Elias Gamma Code ist ein erster, einfacher Universal Code einer ganzen Codefamilie von Peter Elias von 1975. Sei x eine positive, ganze Zahl und n die Anzahl der Bits ihrer Binärdarstellung. Dann erfolgt die Kodierung so, dass der Zahl $n-1$ Nullen vorangestellt werden und dann die Binärdarstellung von x selbst geschrieben wird. Eine äquivalente Definition wäre, dass zuerst die Länge der Zahl im oben definierten Unary Code kodiert geschrieben wird und die kopflose Darstellung der Zahl angefügt wird. Es ergibt sich die Darstellung:

$$\text{encode}_\gamma(x) = 0^{n-1} bin(x) = \text{encode}_{\text{unary}}(n)\ \text{bin}_\text{h}(x)$$

Beispiel: $\text{encode}_\gamma(13) = 000\ 1101 = 0001\ 101$. Das Leerzeichen dient lediglich der Lesbarkeit und würde in einem echten Datenstrom entfallen. Die Dekodierung kann, wie auch die Kodierung, unterschiedlich beschrieben werden. Der erste Ansatz lautet: Zähle die Nullen bis eine Eins auftritt und diese Anzahl sei m. Interpretiere die Eins als 2^m. Addiere die Zahl, deren Binärdarstellung aus den nächsten m Bits besteht und erhalte somit die ursprüngliche Zahl x. Die zweite Formulierung wäre: Dekodiere die Länge n mit Hilfe des Dekoders für den Unary Code, lese die nächsten $n-1$ Bits und interpretiere diese als kopflose Binärdarstellung der Zahl x. Beide Verfahren sind identisch und können die kodierte Zahl eindeutig rekonstruieren. Das erste Verfahren z.B., da die Binärdarstellung jeder Zahl $x > 0$ mit einer Eins beginnt und somit die Längeninformation und die Binärdarstellung der Zahl auseinandergehalten werden können. Die Codewortlänge für eine Zahl x beträgt $1 + \lfloor \log_2 x \rfloor$ für die Längeninformation plus $\lfloor \log_2 x \rfloor$ für die kopflose Darstellung. Somit beträgt die Länge des Codeworts für eine Zahl x: $1 + 2\lfloor \log_2 x \rfloor = O(\log_2 x)$.

3.3 Elias Delta

Der zweite Code der Codefamilie ist der Elias Delta Code. Auch dieser besteht aus einer Längeninformation und einer Darstellung der zu kodierenden Zahl. Jedoch wird hier für den ersten Teil, an Stelle des Unary Codes, der Elias Gamma Code verwendet. Ein Integer x mit einer Binärdarstellung der Länge n wird demnach wie folgt kodiert:

$$\text{encode}_\delta(x) = \text{encode}_\gamma(n)\ \text{bin}_\text{h}(x)$$

Beispiel: $\text{encode}_\delta(13) = \text{encode}_\gamma(4)\ \text{bin}_\text{h}(13) = 00100\ 101$. Mit dieser Kodierung werden $\lfloor \log_2 x \rfloor$ Bits für die kopflose Darstellung und $1 + 2\lfloor \log_2 n \rfloor = 1 + 2\lfloor \log_2(\lfloor \log_2 x \rfloor + 1) \rfloor$ Bits für den Rest benötigt. Der hintere Term kann auch als $1 + 2\lfloor \log_2(\log_2 2x) \rfloor$ geschrieben werden und das ergibt zusammen: $1 + 2\lfloor \log_2(\log_2 2x) \rfloor + \lfloor \log_2 x \rfloor = O(\log_2 x)$. Der Elias Gamma Code liefert nur für $x \in \{2, 3, 8, 9, 10, 11, 12, 13, 14, 15\}$ kürzere Codeworte als der Elias Delta Code, ansonsten sind dessen Codeworte gleich lang oder kürzer.
Weitere Codes dieser Familie können rekursiv konstruiert werden, jedoch erzielt man nur für sehr große Eingabezahlen einen Gewinn. In [MT02] wird als Beispiel die Zahl eine Milliarde (10^9) genannt, die mit C_γ 59 Bits, mit C_δ 39 Bits und mit dem nächsten Code ebenfalls 39 Bits benötigen würde. C_γ und C_δ benötigen jeweils 9 Bits um die Längeninformation $1 + \lfloor \log_2 10^9 \rfloor = 30$ zu kodieren. D.h. in der Praxis genügt der Elias Delta Code meist aus.

3.4 Levenstein

Ein weiterer Universal Code ist der 1968 von Vladimir Levenshtein entwickelte Levenshtein oder auch Levenstein Code. In [Sal07] wird die Kodierung eines Integers x damit wie folgt beschrieben:

1. Wenn $x = 0$, schreibe 0 und überspringe restliche Schritte.
2. Setze Zählvariable C auf 1. Initialisiere das Codewort mit dem leerem String.
3. Nehme Binärdarstellung von x ohne führende 1 und stelle sie dem bisherigen Codewort voran.
4. Sei M die Anzahl der in Schritt 3 angefügten Bits.
5. Wenn $M \neq 0$, inkrementiere C um 1, gehe zu Schritt 3 mit M an der Stelle von x.
6. Wenn $M = 0$, stelle dem bisherigen Code C 1er gefolgt von einer 0 voran und stoppe.

Verwendet man die alternative Unary Kodierung für eine Zahl n (n-1 1er gefolgt von einer 0), so kann die Kodierung auch mit folgendem Pseudocode durchgeführt werden:

encode$_{\text{Lev}}(x)$ =

```
C = 1;
M = x;
while (M > 0) {
      prepend bin_h(M);
      M = floor(log2(M));
      C++;
}
prepend unary_alternative(C);
```

Es gilt:

- bin_h(x) ist die kopflose Binärdarstellung
- floor(y) ist die Abrundungsfunktion: um ganze Zahlen zu erhalten, falls $M \neq 2^k$
- log2(z) ist der Logarithmus zur Basis 2
- prepend meint das Voranstellen vor das bisherige Codewort

Beispiel: encode$_{\text{Lev}}(13) = 1110\ 1\ 101$. Ausführlich:

- $C = 1$, $M = 13 \rightarrow$ prepend $\text{bin}_{\text{h}}(13)$ $(= 101)$ $\rightarrow M = \lfloor \log_2(13) \rfloor = 3$, $C++$
- $C = 2$, $M = 3 \rightarrow$ prepend $\text{bin}_{\text{h}}(3)$ $(= 1)$ $\rightarrow M = \lfloor \log_2(3) \rfloor = 1$, $C++$
- $C = 3$, $M = 1 \rightarrow$ prepend $\text{bin}_{\text{h}}(1)$ $(\equiv$ leeres Wort$)$ $\rightarrow M = \lfloor \log_2(1) \rfloor = 0$, $C++$
- prepend $\text{encode}_{\text{alternative-unary}}(C) = \text{encode}_{\text{alternative-unary}}(4) = 1110$

Der erste Teil beschreibt in der alternativen Unarykodierung die Anzahl der noch folgenden Bits. Mit dem Levenstein Code kann auch die Null kodiert werden.

Um zu zeigen, dass die Dekodierung eindeutig erfolgen kann, wird ein Verfahren dafür angegeben, vgl. [Sal07]:

1. Setze Zähler C auf die Anzahl aufeinanderfolgender 1er und entferne erste 0.
2. Wenn $C = 0$, gib 0 aus und überspringe restliche Schritte.
3. Setze $N = 1$ und wiederhole Schritt 4 $(C - 1)$ Mal.
4. Lese N Bits, stelle diesen '1' voran und weise resultierenden Wert N zu.

Der Wert, der am Schluss in N steht, ist das dekodierte Ergebnis.

Beispiel: decode$_{\text{Lev}}(11101101) = 13$. Ausführlich:

- $y = 11101101$ zu dekodierendes Codewort
- Lese 1er und entferne eine $0 \rightarrow$ $C = 3$, $y = 1101$
- Setze $N = 1$ und wiederhole nächsten Schritt $C - 1$ $(= 2)$ Mal $\rightarrow$ $N = 1$
- Lese N $(= 1)$ Bit, stelle diesen '1' voran und weise Wert N zu $\rightarrow$ $N = 11 = 3$, $y = 101$
- Lese N $(= 3)$ Bits, stelle diesen '1' voran und weise Wert N zu $\rightarrow$ $N = 1101 = 13$, $y =$ $''$
- $\Rightarrow$ decode$_{\text{Lev}}(11101101) = 13$

3.5 Fibonacci

Ein Vertreter der Codes mit 'eof' Symbol ist der Fibonacci Code. Dieser gehört ebenfalls zu den Universal Codes [Hir]. Definiert wurde er von Apostolico und Fraenkel 1985 [AF87] und er basiert auf der Darstellung eines Integers x als Summe von Fibonacci-Zahlen. Eine möglicher Kodierungsalgorithmus ist:

encode$_{\text{Fib}}$ (x) =

```
o Generiere alle Fibonacci−Zahlen
        F(1), F(2), .., F(m) <= x, F(m+1) > x.
o Initialisiere Codewort mit 1.
for (i = m; i > 1; i−−) {
    if (x − F(i) >= 0) {
        x = x − F(i);
        prepend(1);
    } else {
        prepend(0);
    }
}
```

Wobei F(i) die i-te Fibonacci-Zahl ist und diese mit dem Index 1 beginnt, d.h. es gilt:

F(1) = F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
...

Da F(1) und F(2) identisch und somit redundant sind, wird F(1) weggelassen und man beginnt mit F(2) bei der Kodierung. Beispiel: encode$_{\text{Fib}}(13)$ = 0000011. Die Dekodierung ist einfach über die Summe der Fibonacci-Zahlen zu berechnen, deren entsprechenden Bits gesetzt sind: $x = \sum_{i=0}^{m-2} d(i)F(i+2)$. Im Beispiel sind nur die letzten 2 Bits '1', da 13 die 7. Fibonacci-Zahl ist und somit keine weiteren addiert werden müssen. Das letzte Bit wird beim Dekodieren ignoriert und dient nur als 'eof' Symbol.
Es kann gezeigt werden, dass für keine Darstellung einer natürlichen Zahl in dieser Form, d.h. als so eine Summe von Fibonacci-Zahlen, jemals zwei Einsen hintereinander auftreten [Sal00]. Die Idee ist die, dass wenn zwei aufeinanderfolgende Fibonacci-Zahlen vorkommen, dann müsste bereits die nächstgrößere Fibonacci-Zahl (als Summe der zwei) vorkommen und dies wird induktiv fortgesetzt. Somit ist es möglich, durch die Eins für die größte vorkommende Fibonacci-Zahl und die künstlich angehängte Eins, die Zahlen in einem Datenstrom eindeutig zu dekodieren.
Der Fibonacci Code ist nicht asymptotisch optimal, (genauer wurde sogar gezeigt, dass der Fibonacci Code von Grad 2 die Konstanten $c_1 = 2$ und $c_2 = 3$ besitzt [Hir]) jedoch erzeugt er für kleine Zahlen (bis $x = 514.228$) kürzere Codeworte als die Elias Delta Code. Eine weitere interessante Eigenschaft ist, dass der Fibonacci Code robuster gegenüber Bitfehlern ist, da ein fehlerhafter Datenstrom nur für wenige Zahlen falsche Ergebnisse liefert und weiter dekodiert werden kann [Wikb]. Genauer gibt es 3 Fälle:

1. 000 wird zu 010 oder umgekehrt: Dann wird nur die aktuelle Zahl beeinflusst.
2. 0010 wird zu 0110 (Delimiter wird eingefügt): Dabei wird ursprünglich eine Zahl durch 11 zu zwei Zahlen getrennt. Das nächste Auftreten von 11 beendet jedoch diesen fehlerhaften Block und alle folgenden Zahlen werden korrekt dekodiert.
3. 0110 wird zu 0100 oder 0010 (Delimiter wird zerstört): Durch das Verschwinden des Trennzeichens verschmelzen zwei Zahlen zu einer. Mit dem Endesymbol 11 der zweiten Zahl endet wiederum der Fehler.

Solch eine Eigenschaft trifft nicht für jeden Universal Code zu.

3.6 Golomb

Der nächste Code, der vorgestellt wird, ist der Golomb Code. Dieser gehört ebenfalls zu den Universal Codes und ist ein parametrisierter Code. In [Sal00], [Say00], [MT02] werden drei leicht unterschiedliche Definitionen genannt. Hier wird mit folgender Definition von [Sal00] gearbeitet:

Sei x die zu kodierende Zahl. Berechne $q = \lfloor \frac{x-1}{b} \rfloor$ und $r = x - qb - 1$, wobei der Code von dem Parameter $b > 0$ abhängt. ($\Rightarrow q \in [0, \infty)$, $r \in [0, b-1]$)

$$\text{encode}_{\text{Gol}}(x) = \text{encode}_{\text{unary}}(q+1)\ \text{encode}_{\text{minimal}}(r)$$

Da nach Wahl des Parameters b der Wertebereich für r festgelegt ist, kann dafür ein fester minimaler Code generiert werden, was das $\text{encode}_{\text{minimal}}(r)$ hier macht. Entweder verwendet man dazu eine Huffman Kodierung oder ein einfaches statisches Schema: Kodiere die ersten $2^{\lceil \log_2(b) \rceil} - b$ Zahlen mit $\lfloor \log_2(b) \rfloor$ Bits und die restlichen mit $\lfloor \log_2(b) \rfloor + 1$ Bits. Beispiel: $\text{encode}_{\text{Gol}}(13)$ für $b = 3$. $\lfloor \log_2(3) \rfloor = 1$, $\lceil \log_2(3) \rceil = 2$, $2^2 - 3 = 1$. D.h. ein Element von r wird mit 1 Bit kodiert und die restlichen mit 2 Bit, z.B. $r = 0 \rightarrow 0$, $r = 1 \rightarrow 10$, $r = 2 \rightarrow 11$. $q = \lfloor \frac{13-1}{3} \rfloor = 4$, $r = 13 - 4 \cdot 3 - 1 = 0 \Rightarrow \text{encode}_{\text{Gol}}(13) = 00001\ 0$ (für $b = 3$).
Mit dieser Definition kann $x = 0$ nicht direkt kodiert werden, jedoch wird in [Say00] eine alternative Definition verwendet, die dies ermöglicht.
Der Golomb Code hat eine sehr nützliche Eigenschaft. Er ist optimal für eine Geometrische Verteilung mit einer Wahrscheinlichkeit p für Erfolg, wenn $b = \lceil \frac{\log_e(2-p)}{-\log_e(1-p)} \rceil \approx (\log_e 2)\frac{1}{p} \approx 0.69 \cdot \frac{1}{p}$ gewählt wird [MT02].

3.7 Rice

Rice Codes (mit Parameter k) sind eine spezielle Art von Golomb Codes, wobei der Golomb Parameter $b = 2^k$ gewählt wird. $k = 0$ ist dabei auch erlaubt, was einem $b = 1$ entspricht und identisch zum Unary Code ist. Ein Vorteil des Rice Codes ist die effiziente Implementierung. Die Division kann durch einen Shift um k Stellen nach rechts durchgeführt werden. Der Wert, der sich dabei ergibt, wird um 1 inkrementiert und mit dem Unary Code kodiert. Die ursprünglichen niederwertigen k Bits, werden direkt als k-stellige Binärzahlen kodiert.
Beispiel: $\text{encode}_{\text{Rice}}(13) = 000101$ mit $k = 2$. Dies kommt folgendermaßen zustande:
$\text{bin}(13) = 1101$ wird um k (= 2) Stellen nach rechts geschoben $\rightarrow$ 11 bleibt übrig. Um 1 inkrementiert erhält man 100, was dezimal 4 entspricht. Die herausgeschobenen Bits waren 01. Somit ist $\text{encode}_{\text{Rice}}(13) = \text{encode}_{\text{unary}}(4)\ 01 = 0001\ 01$.

3.8 Exponential-Golomb

Ein weiterer parametrisierter Universal Code ist der Exponential-Golomb Code oder auch Exp-Golomb Code. Die Kodierung eines Integers x mit dem Exp-Golomb Code mit Paramter k kann wie folgt beschrieben werden [Wika]:

1. Nehme Binärdarstellung von x ohne letzten k Bits, addiere 1 hinzu und schreibe diese Zahl.
2. Zähle Anzahl geschriebener Bits, subtrahiere 1 & stelle bisherigem Code so viele Nullen voran.
3. Schreibe die letzten k Bits von x in Binärdarstellung.

Mit den bisher definierten Codes, kann man die Kodierung eines Integers x mit dem Exp-Golomb Code mit Parameter k auch anders schreiben. Sei dazu $M = \lfloor \frac{x}{2^k} \rfloor + 1$, dann gilt:

$$\text{encode}_{\text{ExpGol}}(x) = \text{encode}_{\text{unary}}(\lfloor \log_2 M \rfloor + 1)\ \text{bin}_{\text{h}}(M)\ \text{k-letzteBits}(\text{bin}(x))$$

Aus dieser Definition geht allerdings nicht hervor, was geschrieben werden muss, wenn die Länge von $\text{bin}(x) < k$. Für den hier beschriebenen Dekoder müssen führende Nullen in x eingefügt werden, sodass die Länge mindestens k beträgt. Meist wird $k = 0$ verwendet, was diesen Fall ausschließt und keine Information darüber liefert. Zudem ist für $k = 0$ der Exp-Golomb Code von x identisch mit dem Elias Gamma Code von $x + 1$, was ermöglicht $x = 0$ zu kodieren.
Beispiel: $\text{encode}_{\text{ExpGol}}(13)$ für $k = 3$. $M = \lfloor \frac{13}{2^3} \rfloor + 1 = \lfloor \frac{13}{8} \rfloor + 1 = 2$.

$$\text{encode}_{\text{ExpGol}}(13) = \text{encode}_{\text{unary}}(\lfloor \log_2 2 \rfloor + 1)\ \text{bin}_{\text{h}}(2)\ \text{3-letzteBits}(\text{bin}(x))$$
$$\text{encode}_{\text{ExpGol}}(13) = \text{encode}_{\text{unary}}(2)\ \text{bin}_{\text{h}}(2)\ \text{3-letzteBits}(1101)$$
$$\text{encode}_{\text{ExpGol}}(13) = 01\ 0\ 101$$

Ein möglicher Dekodierer für $k \neq 0$ ist:

1. Zähle Anzahl führender Nullen und speichere in n.
2. Lies die nächsten $n + 1$ Bits und speichere in x.
3. Lies die nächsten k Bits und speichere in y.
4. Dekrementiere x um 1.
5. Schiebe x um k Stellen nach links (= multipliziere x mit 2^k) und addiere y.

4 Vergleiche, Anwendungen & weitere Eigenschaften

4.1 Vergleich: Unary Code & Elias Gamma Code

In [MT02] wird beschrieben, wie man die Anwendung von Codes als Ratespiele verstehen kann. Anhand dessen werden dann der Unary Code und der Elias Gamma Code verglichen.
Das Ratespiel ist folgendermaßen aufgebaut: Eine Partei überlegt sich eine Nummer zwischen 1 und n. Die andere Partei stellt fragen der Form 'Ist die Zahl größer als x?', worauf mit 'Ja' oder 'Nein' geantwortet wird. Ziel ist nun die Anzahl der Fragen, die benötigt werden um die Zahl herauszufinden, zu minimieren. Dabei kann ein Code nun so interpretiert werden, dass jede Antwort auf eine solche Frage genau einem Bit entspricht. Wenn alle Bits verwendet wurden, hat man eine Beschreibung der Zahl $0 \leq x - 1 < n$. Man weiß, dass die optimale Lösung darin besteht, den Suchraum mit jeder Frage zu halbieren. Mit dieser Methode findet man die Lösung nach spätestens $\lceil \log_2(n) \rceil$ Fragen.
Der Unary Code stellt die Fragen 'Ist die Zahl größer als 0?', 'Ist die Zahl größer als 1?', 'Ist die Zahl größer als 2?... Es handelt sich also um eine lineare Suche. Dies ist nicht optimal, aber findet die Lösung nach absehbar vielen Schritten.
Was jedoch, wenn n nicht bekannt ist, bzw. das Ratespiel auf die Form gebracht wird: 'Eine Partei überlegt sich eine positive Zahl...'? Die Anwendung der linearen Suche ist dafür offensichtlich nicht sehr effizient, aber auch die optimale Strategie kann nicht mehr angewendet werden.
Der Elias Gamma Code bietet eine weitere Strategie. Er beginnt ebenfalls mit der Frage 'Ist die Zahl größer als 1?', aber fährt bei Zustimmung fort mit den Fragen 'Ist die Zahl größer als 3?', 'Ist die Zahl größer als 7 $(= 2^3 - 1)$?', ... Irgendwann wird ein Punkt erreicht sein, an dem ein 'Nein' als Antwort geliefert wird. Dabei wird in eine binäre Suche übergegangen und die Zahl so exakt bestimmt.
Auch andere Codes können auf diese Art und Weise interpretiert und analysiert werden.

4.2 Vergleich: Elias Codes & Golomb Code

Ebenfalls in [MT02] wird nochmals eine ähnliche Interpretation genannt. Die Aufgabe besteht wieder darin eine Zahl exakt zu bestimmen. Anstatt nun aber die Suche über die Schrittweite zu beschreiben, wird näher darauf eingegangen wie viele Elemente in den einzelnen Schritten möglich wären.
Der Elias Gamma und Elias Delta Code sind beide Beispiele für eine Gruppe von Codes, die aus einem *Selektor* Teil und aus einem Binärdarstellungsteil bestehen. Der Selektor beschreibt jeweils einen Wertebereich, ein sogenanntes *Bucket* und der Binärdarstellungsteil gibt die genaue Platzierung des Elementes in dem Bucket an.
Unterscheiden kann man nun die Codes mit Hilfe eines Vektors, der die Größe der Buckets beschreibt und dem Code für den Selektor. Für die Elias Codes ist der Vektor

$$(1, 2, 4, 8, \ldots, 2^k, \ldots)$$

identisch. Der Unterschied zwischen beiden besteht darin, dass im Elias Gamma Code der Unary Code für den Selektor verwendet wird und im Elias Delta der Elias Gamma Code.
Der Golomb Code hingegen verwendet zwar auch einen Unary Code Selektor, wie der Elias Gamma Code, jedoch hat dieser eine konstante Bucketgröße von

$$(b, b, b, b, \ldots)$$

Äquivalent dazu haben Rice Codes eine konstante Bucketgröße von

$$(2^k, 2^k, 2^k, 2^k, \dots)$$

Wie auch bei dem Vergleich über die Suchstrategie, kann man auch diese Interpretation auf weitere Codes anwenden um etwas über deren Eigenschaften herauszufinden und Codes anschaulich zu vergleichen.

4.3 Wertebereich erweitern

Viele Codes können die Null nicht direkt kodieren. Benötigt man einen Code, der dies aber kann, so muss man seine Auswahl nicht gleich stark einschränken, sondern kann sich mit sogenannten Mapping oder auch Umbelegungen aushelfen. Anstatt einen Integer x zu kodieren, verwendet man denselben Kodierer stattdessen für $x + 1$ und das resultierende Codewort soll dann das Codewort für x sein. Für manche Codes, wie z.B. den Golomb Code, gibt es auch leicht abgeänderte Definitionen, die es ebenfalls zulassen, die Null zu kodieren. In allen Fällen ist es wichtig, dass auch der Dekodierer entsprechend angepasst wird.
Manchmal möchte man nicht nur positive Zahlen kodieren, sondern auch negative. Auch hierfür kann man ein Mapping erstellen, beispielsweise in der Form $f(x) = \lfloor x/2 \rfloor \cdot (-1)^{x \bmod 2}$, wobei mod der Modulo-Operator ist, der den Rest einer Ganzzahldivision zurückliefert. Es ist zu beachten, dass es bei diesem Beispiel eine positive und negative Null gibt. Logischerweise wächst durch die Verwendung der Mappings die Anzahl der benötigten Bits für die einzelnen positiven Zahlen.

4.4 Anwendungen der Codes

Im Folgenden werden nun noch einige möglichen Anwendungsbeispiele für die einzelnen Codes aufgeführt. Diese sind nicht vollständig und werden auch nicht weiter belegt.
Der Unary Code wird wegen seines linearen Wachstums meist nicht direkt, sondern überwiegend in anderen Kodierungen eingesetzt. Beispiele wie Elias Gamma und Golomb Code wurden bereits beschrieben. Der Elias Gamma Code wird, wie der Unary Code, als Teil eines anderen Codes, in diesem Fall im Elias Delta Code, eingesetzt. Er könnte wegen seines besseren Wachstumsverhaltens aber auch für Anwendungsgebiete des Elias Delta Codes eingesetzt werden. Dieser wird manchmal für weitere Codes der Elias Code Familie verwendet, aber auch direkt für die Kodierung von Schlüsseln in Datenbanken. Die Golomb Codes werden, wie auch Rice Codes, oftmals für verlustfreie Datenkompression eingesetzt. Für den Fibonacci und den Levenstein Code wurden keine erwähnten Anwendungsgebiete gefunden, jedoch könnten diese ebenfalls für alle der genannten Aufgaben (mehr oder weniger gut geeignet) verwendet werden.
Der Exponential-Golomb Code wird mit einem Parameter $k = 0$ im 'H.264/MPEG-4 AVC video compression standard' verwendet, wobei man bei $k = 0$ auch sagen könnte, dass ein Elias Gamma Code verwendet wird, der ein Mapping benutzt um die Null zusätzlich kodieren zu können. Teilweise wird im Bereich der Videokodierung auch der Exp-Golomb Code mit einem Mapping verwendet um auch negative Ganzzahlen kodieren zu können.

5 Zusammenfassung

Es wurde ein kurzer Überblick über die geläufigsten, unbeschränkten Codes für Integer gegeben. Im Folgenden ist noch eine Codetabelle zu finden, die die Codeworte der verschiedenen Codes für wenige, kleine Zahlen exemplarisch aufführt. Bei manchen sind die Unterschiede im Wachstumsverhalten bereits zu erkennen, bei anderen wäre dies erst für sehr große Zahlen der Fall. Die Tabelle soll auch der Überprüfung der Verfahren dienen, wobei bei Golomb und Rice Codes auf die Kodierung der Reste geachtet werden muss.

Literatur

[AF87] Alberto Apostolico and Aviezri S. Fraenkel. Robust transmission of unbounded strings using fibonacci representations. *IEEE Transactions on Information Theory*, 33(2):238–245, 1987.

[Hir] Dan Hirschberg. Universal codes and representation of integers. http://www.ics.uci.edu/∼dan/pubs/DC-Sec3.html. Stand: 1. Juni 2012.

[Mac03] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. http://www.inference.phy.cam.ac.uk/mackay/itila/.

[MT02] Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithms*. Kluwer, 2002.

[Oja] Pasi Ojala. Pucrunch. http://www.cs.tut.fi/∼albert/Dev/pucrunch/. Stand: 1. Juni 2012.

[Sal00] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2nd edition, 2000.

[Sal07] David Salomon. *Variable-length Codes for Data Compression*. Springer-Verlag, 2007.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2000.

[Wika] Wikipedia. Exponential-Golomb coding. https://en.wikipedia.org/wiki/Exponential-Golomb_coding. Stand: 1. Juni 2012.

[Wikb] Wikipedia. Fibonacci coding. https://en.wikipedia.org/wiki/Fibonacci_coding. Stand: 1. Juni 2012.

Tabelle 1. Codetabelle: (* nicht ersichtlich, ob *0*en eingefügt werden müssen)

x	bin(x)	Unary	Elias Gamma	Elias Delta	Levenstein	Fibonacci	Golomb ($b = 3$)	Rice ($k = 2$)	Exp-Golomb ($k = 3$)
0	0	-	-	-	0	-	-	-	1*00*0 *
1	1	1	1	1	10	11	10	11	1*00*1 *
2	10	01	010	0100	1100	011	110	110	1*0*10 *
3	11	001	011	0101	1101	0011	111	111	1*0*11 *
4	100	0001	00100	01100	1110000	1011	010	0100	1100
5	101	00001	00101	01101	1110001	00011	0110	0101	1101
6	110	000001	00110	01110	1110010	10011	0111	0110	1110
7	111	0000001	00111	01111	1110011	01011	0010	0111	1111
8	1000	00000001	0001000	00100000	11101000	000011	00110	00100	010000
9	1001	000000001	0001001	00100001	11101001	100011	00111	00101	010001
10	1010	0000000001	0001010	00100010	11101010	010011	00010	00110	010010
11	1011	00000000001	0001011	00100011	11101011	001011	000110	00111	010011
12	1100	000000000001	0001100	00100100	11101100	101011	000111	000100	010100
13	1101	0000000000001	0001101	00100101	11101101	0000011	000010	000101	010101
14	1110	00000000000001	0001110	00100110	11101110	1000011	0000110	000110	010110
15	1111	000000000000001	0001111	00100111	11101111	0100011	0000111	000111	010111
16	10000	0000000000000001	000010000	001010000	111100000000	0010011	0000010	0000100	011000
17	...	...	...	...	...	...	...	...	...