Huffman-Algorithmus
Formalisierung und Optimalitätsbeweis

Maximilian Schäffeler

13.01.2017

Dies ist die Ausarbeitung zum Seminarvortrag in der Vorlesung „Perlen der Informatik 3“. Es wird ein formaler Beweis des Huffman-Algorithmus von J.C. Blanchette vorgestellt. Dieser wird in der Syntax des interaktiven Theorembeweisers Isabelle geführt. Schließlich werden kurz Anwendungen des Algorithmus aufgezeigt.

Inhaltsverzeichnis

1 Einleitung **2**

2 Huffman-Algorithmus **2**

3 Funktionale Implementierung **3**

4 Hilfsfunktionen und Lemmata **4**
4.1 Eigenschaften von Binärbäumen 5
4.2 swapSyms und swapFourSyms 6

5 Optimalitätsbeweis **7**

6 Anwendungen **9**

1 Einleitung

Der Huffman-Algorithmus stellt eine Methode zur verlustfreien Komprimierung von Daten dar. Er wurde 1952 von David A. Huffman entwickelt [1]. Der Algorithmus erzeugt aus Symbolen und deren Verteilung im zu komprimierenden Text, Binärbäume, aus welchen präfixfreie Kodierungen variabler Länge (Huffman-Codes) gewonnen werden. Diese kodieren den Text mit minimalem Speicherplatz.

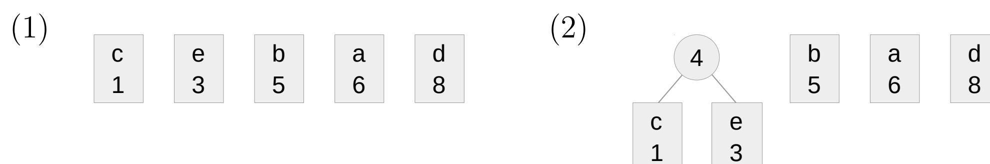
Huffman lieferte in seiner Arbeit keinen Beweis für die Optimalität der erzeugten Kodierungen, es finden sich jedoch in der Literatur bei Cormen [3] und Knuth [4] Beweise bzw. Beweisskizzen. Aufbauend auf diesen Beweisen wird in der Arbeit *Proof Pearl: Mechanizing the Textbook Proof of Huffman's Algorithm* von J. C. Blanchette der Algorithmus und dessen Beweis in Isabelle formalisiert [2]. Die folgenden Kapitel 2 bis 5 basieren auf dieser Arbeit.

2 Huffman-Algorithmus

Als Eingabe erwartet der Huffman-Algorithmus Symbole mit zugeordneten Gewichten. In der Praxis entsprechen diese fast immer den Häufigkeiten, mit denen die Symbole in den zu komprimierenden Daten vorkommen. Der Algorithmus konstruiert dann einen Binärbaum, dessen Blätter die eingegebenen Symbole mit Gewichten sind. Der entstandene Baum hat minimale gewichtete Pfadlänge. Diese ist für ein einzelnes Symbol das Produkt aus Gewicht und Tiefe eines Symbols im Baum. Die Summe der gewichteten Pfadlängen aller Symbole ergibt schließlich die gewichtete Pfadlänge des Baumes.

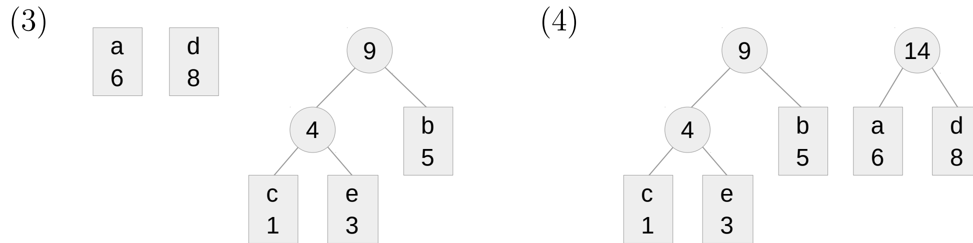
Im Folgenden wird die Vorgehensweise des Algorithmus beschrieben: In jedem Schritt wählt der Huffman-Algorithmus aus einer Liste von Bäumen jeweils zwei mit minimalem Gewicht an der Wurzel aus. Diese werden kombiniert, indem ein neuer Knoten erstellt wird, der als Gewicht die Summe der Gewichte der beiden Bäume erhält. Linker und rechter Teilbaum dieses Knotens sind die beiden Bäume, die kombiniert werden sollen.

Anschließend wird der entstandene Baum wieder in die Liste eingefügt. Der Algorithmus terminiert, sobald in der Liste nur noch ein einzelner Baum übrig ist. Dieser Baum ist die Ausgabe des Algorithmus, aus ihm können nun Kodierungen für die Symbole gewonnen werden. Das folgende Beispiel soll den Algorithmus anschaulich verständlich machen:

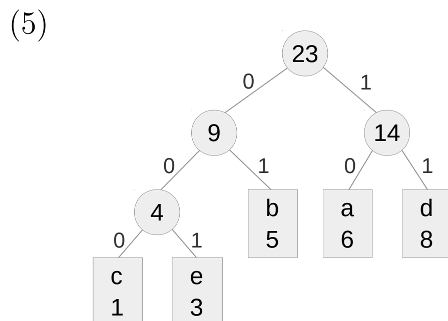


In diesem Beispiel sind die beiden Zeichen mit den geringsten Gewichten c

und e. Der aus den beiden Symbolen entstandene Baum hat Gewicht 4, wird also wieder am Beginn der Liste einsortiert (2). Im Schritt von (2) auf (3) wird der Baum mit den Symbolen c und e mit dem Symbol b verbunden, da diese Bäume die geringsten Gewichte an der Wurzel tragen. In den folgenden Schritten wird dieses Vorgehen wiederholt.



Der so entstandene Binärbaum ist optimal bzgl. gewichteter Pfadlänge, aus ihm können nun Kodierungen der Symbole gewonnen werden, die die zugehörigen Daten optimal komprimieren: Jede Kante zu einem linken Teilbaum wird mit einer 0 annotiert, jede Kante zu einem rechten Teilbaum mit einer 1 (Abbildung (5)). Die Kodierung des Symbols a ergibt sich nun beispielsweise durch Konkatenation der Annotationen auf dem Pfad von der Wurzel zu a, in diesem Fall 10.



3 Funktionale Implementierung

Die funktionale Implementierung und der Beweis folgt der Arbeit von J.C. Blanchette und verwendet die Syntax des Theorembeweisers Isabelle.

Binärbäume werden durch einen eigenen Datentyp *tree* dargestellt. Jedes Blatt (*Leaf*) und jeder innere Knoten (*InnerNode*) haben ein Attribut vom Typ *nat*, welches ganzzahlige Werte speichert, hier das Gewicht des Teilbaumes. Das Attribut vom Typ α in *Leaf* beinhaltet die Symbole, die in den Daten vorkommen und für welche Kodierungen gefunden werden sollen. Innere Knoten besitzen zusätzlich einen linken und einen rechten Teilbaum, wiederum vom Typ α *tree*.

$$\textit{datatype}\ \alpha\ \textit{tree} = \textit{Leaf nat}\ \alpha \mid \textit{InnerNode nat}\ (\alpha\ \textit{tree})\ (\alpha\ \textit{tree})$$

In jedem Schritt des Huffman-Algorithmus müssen zwei Bäume vereinigt werden, hierzu wird ein neuer Knoten eingeführt, der als linken und rechten Teilbaum diese beiden Bäume besitzt. Das Gewicht der Wurzel des neuen Baums ergibt sich aus der Summe der Gewichte der Wurzeln der beiden übergebenen Bäume.

$$\mathit{uniteTrees}\ t_1\ t_2 = \mathit{InnerNode}\ (\mathit{cachedWeight}\ t_1 + \mathit{cachedWeight}\ t_2)\ t_1\ t_2$$

cachedWeight gibt hier einfach das Gewicht des Teilbaumes oder Blattes zurück.

$$\mathit{cachedWeight}\ (\mathit{Leaf}\ w\ a) = w$$
$$\mathit{cachedWeight}\ (\mathit{InnerNode}\ w\ t_1\ t_2) = w$$

Die Funktion *insortTree* fügt einen Binärbaum an der passenden Stelle in eine aufsteigend nach Gewichten sortierte Liste von Bäumen ein. Die Sortierung der Liste vereinfacht die Suche nach Bäumen mit minimalen Gewichten an der Wurzel.

$$\mathit{insortTree}\ u\ [\,] = [u]$$
$$\begin{aligned}\mathit{insortTree}\ u\ (t \cdot ts) = (&\text{if } \mathit{cachedWeight}\ u \leq \mathit{cachedWeight}\ t\\ &\text{then } u \cdot t \cdot ts\\ &\text{else } \ t \cdot \mathit{insortTree}\ u\ ts)\end{aligned}$$

huffman vereinigt schließlich jeweils die zwei ersten Elemente in der Liste von Bäumen, dies sind die Bäume mit minimalem Gewicht an der Wurzel. Die Funktion fügt den so entstandenen Baum sortiert wieder in die Liste ein und wird rekursiv mit der neuen Liste aufgerufen, bis diese nur noch ein Element enthält. Dieses Vorgehen entspricht genau dem Huffman-Algorithmus.

$$\mathit{huffman}\ [t] = t$$
$$\mathit{huffman}\ (t_1 \cdot t_2 \cdot ts) = \mathit{huffman}\ (\mathit{insortTree}\ (\mathit{uniteTrees}\ t_1\ t_2)\ ts)$$

Folgende Anforderungen werden an die Eingaben gestellt, damit der Algorithmus eine optimale Lösung findet: Die Eingabeliste muss aufsteigend nach Gewichten sortiert sein und darf nur aus Blättern bestehen. Außerdem darf jedes Symbol nur einmal in der Liste vorhanden sein.

4 Hilfsfunktionen und Lemmata

Um die Korrektheit des Algorithmus zu beweisen, werden zunächst Funktionen definiert, die Aussagen über die Eigenschaften und Korrektheit von Eingaben, Binärbäumen und die Optimalität der Ausgaben treffen lassen.

4.1 Eigenschaften von Binärbäumen

Das Alphabet (*alphabet*) eines Baums ist die Menge aller Symbole, die den Blattknoten im Baum zugeordnet sind.

$$alphabet(Leaf\ w\ a) = a$$
$$alphabet(InnerNode\ w\ t_1\ t_2) = alphabet\, t_1 \cup alphabet\, t_2$$

Zusätzlich muss gewährleistet sein, dass jedes Symbol nur einmal im Baum vorkommt, d.h. dass das Alphabet des rechten und des linken Teilbaumes sind disjunkt sind. Der Huffman-Algorithmus arbeitet nur mit diesen konsistenten Bäumen korrekt.

$$consistent(Leaf\ w\ a) = True$$
$$\begin{aligned} consistent(InnerNode\ w\ t_1\ t_2) = (&consistent\, t_1 \wedge consistent\, t_2 \\ &\wedge alphabet\, t_1 \cap alphabet\, t_2 = \emptyset) \end{aligned}$$

Die Tiefe eines Symbols (nur für konsistente Bäume definiert) ist die Länge des Pfads von diesem zur Wurzel. Die spätere Kodierungslänge entspricht der Tiefe eines Symbols im Baum. Ein Symbol, das nicht im Baum vorkommt, erhält Tiefe 0.

$$depth(Leaf\ w\ b)\ a = 0$$
$$\begin{aligned} depth(InnerNode\ w\ t_1\ t_2)\ a = (&\text{if } a \in alphabet\, t_1 \text{ then } depth\, t_1\ a + 1 \\ &\text{else if } a \in alphabet\, t_2 \text{ then } depth\, t_2\ a + 1 \\ &\text{else } 0) \end{aligned}$$

Die Höhe des Baumes entspricht der maximalen Tiefe eines Symbols:

$$height(Leaf\ w\ a) = 0$$
$$height(InnerNode\ w\ t_1\ t_2) = max(height\, t_1)(height\, t_2) + 1$$

Die Häufigkeit (*frequency*) eines Symbols ist die Summe der Häufigkeit dieses Symbols im linken und rechten Teilbaum. In einem konsistenten Baum, ist jeweils höchstens ein Summand von 0 verschieden.

$$freq(Leaf\ w\ b)\ a = (\text{if } a = b \text{ then } w \text{ else } 0)$$
$$freq(InnerNode\ w\ t_1\ t_2)\ a = freq\ t_1\ a + freq\ t_2\ a$$

Außerdem wird trotz der Möglichkeit, das Gewicht direkt aus einem Knoten auszulesen, eine Funktion zur Berechnung des Gewichts definiert. Dies vereinfacht den späteren Beweis, da die Korrektheit des aktuell in den Knoten gespeicherten Gewichtes nicht gezeigt werden muss.

$$weight(Leaf\ w\ a) = w$$
$$weight(InnerNode\ w\ t_1\ t_2) = weight\, t_1 + weight\, t_2$$

Die gewichtete Pfadlänge eines Baumes ist:
$\sum_{a \in alphabet\, t} freq\, t\, a * depth\, t\, a = weight\, t + \sum_{a \in alphabet\, t} freq\, t\, a * (depth\, t\, a - 1)$.
Daraus ergibt sich die Rekursionsgleichung

$$cost(Leaf\ w\ a) = 0$$
$$cost(InnerNode\ w\ t_1\ t_2) = weight\, t_1 + cost\, t_1 + weight\, t_2 + cost\, t_2$$

Das Prädikat *optimum t* testet, ob t die minimale gewichtete Pfadlänge aller Binärbäume mit gleichem Symbolen und zugehörigen Gewichten hat. Dies ist genau dann der Fall, wenn kein solcher Baum einen geringeren Wert bzgl. der Funktion cost aufweist.

$$optimum\, t = (\forall u.\ consistent\, u \wedge alphabet\, t = alphabet\, u \wedge freq\, t = freq\, u \longrightarrow cost\ t \leq cost\ u)$$

Alle in diesem Abschnitt vorgestellten Funktionen sind auch in einer Version für Listen von Bäumen verfügbar. Diese Funktionen werden elementweise angewendet und erhalten im Funktionsnamen als Index zusätzlich ein F.

4.2 swapSyms und swapFourSyms

swapFourSyms t a b c d tauscht die Symbole a und b auf die Positionen von c und d, zuvor muss $a \neq b$ und $c \neq d$ gelten. Hierfür wird die Hilfsfunktion *swapSyms* benötigt, die 2 benachbarte Symbole tauscht. Benachbarte Symbole sind Blätter im Baum, die am selben inneren Knoten hängen.

$$\begin{aligned} swapFourSyms\, t\ a\ b\ c\ d = (&\text{if } a = d \text{ then } swapSyms\, t\ b\ c \\ &\text{else if } b = c \text{ then } swapSyms\ t\ a\ d \\ &\text{else } swapSyms(swapSyms\, t\ a\ c)\ b\ d \end{aligned}$$

Die naive Definition (nur der else-Fall) würde in den beiden zuvor abgefangenen Fällen fehlschlagen. Das Problem ist, dass hierbei nicht alle Symbole ihre Position tauschen.

Intuitiv sollten Symbole, die häufiger in den zu komprimierenden Daten vorkommen mit weniger Bits kodiert werden, d.h. eine geringere Tiefe im Binärbaum aufweisen. Dies kann mithilfe von Isabelle gezeigt werden.

Lemma 1
Falls *consistent t*, $a \in alphabet\ t$, $b \in alphabet\, t$,
$freq\, t\ a \leq freq\, t\ b$ und $depth\ t\ a \leq depth\ t\ b$,
dann $cost\ (swapSyms\ t\ a\ b) \leq cost\ t$.

Dieses Lemma kann auf die Funktionsanwendung *swapFourSyms t a b c d* erweitert werden: wenn a und b Blätter mit minimalem Gewicht sind (Prädikat *minima*), c und d auf maximaler Tiefe im Baum liegen, dann erhöhen sich die Kosten nach Anwendung der Funktion nicht.

Lemma 2
Falls $\mathit{consistent}\, t$, $\mathit{minima}\, t\, a\, b$, $c \in \mathit{alphabet}\, t$, $d \in \mathit{alphabet}\, t$,
$\mathit{depth}\, t\, c = \mathit{height}\, t$, $\mathit{depth}\, t\, d = \mathit{height}\, t$ und $c \neq d$,
dann $\mathit{cost}\ (\mathit{swapFourSyms}\, t\, a\, b\, c\, d) \leq \mathit{cost}\, t$.

Der Beweis des Lemmas unterscheidet die Fälle a = c, a = d, b = c und b = d und wendet dann das obige Lemma 1 zu *swapSyms* an.

Da der Huffman-Algorithmus Bäume zusammenfügt, existiert eine Funktion *mergeSibling*, die zwei Blätter des Baumes verschmilzt, das so entstehende Blatt hat als Gewicht die Summe der Gewichte der beiden Blätter. Diese Funktion reduziert die Kosten des Baumes um genau diese Summe. *mergeSibling* erlaubt es, die Kombination zweier Bäume genauer zu betrachten.

Lemma 3
Falls $\mathit{consistent}\ t$ und $\mathit{sibling}\ t\ a \neq a$,
dann $\mathit{cost}(\mathit{mergeSibling}\ t\ a) + \mathit{freq}\ t\ a + \mathit{freq}\ t\ (\mathit{sibling}\ t\ a) = \mathit{cost}\ t.$

Analog wird eine Funktion definiert, die ein Blatt in zwei Blätter spaltet, die sich das Gewicht des ursprünglichen Blattes teilen.
So spaltet $\mathit{splitLeaf}\, t\, w_a\, a\, w_b\, b$ das Blatt a in zwei Blätter a und b auf, die die Gewichte w_a bzw. w_b erhalten.

Auch für diese Funktion findet sich ein Kostenlemma, die Anwendung der Funktion erhöht die Kosten des Baums um $w_a + w_b$.

Lemma 4
Falls $\mathit{consistent}\, t$, $a \in \mathit{alphabet}\, t$ und $\mathit{freq}\, t\, a = w_a + w_b$,
dann $\mathit{cost}\ (\mathit{splitLeaf}\, t\, w_a\, a\, w_b\, b) = \mathit{cost}\, t + w_a + w_b$

5 Optimalitätsbeweis

Mit den Erkenntnissen des vorigen Abschnitts kann nun die Optimalität bzgl. gewichteter Pfadlänge der vom Algorithmus erzeugten Binärbäume gezeigt werden. Es ist leicht zu zeigen, dass das Alphabet, die Konsistenz und die Häufigkeiten der Symbole vom Huffman-Algorithmus unverändert bleiben.

Um den Beweis führen zu können, muss noch eine weitere Eigenschaft von *splitLeaf* im Zusammenhang mit optimalen Bäumen gezeigt werden:
Es wird $\mathit{splitLeaf}\, t\ w_a\ a\ w_b\ b$ auf den Baum angewandt. Wenn t optimal ist, w_a und w_b minimale Gewichte im neuen Baum sind und das Gewicht von a auf w_a und w_b verteilt wird, so bleibt auch der entstandene Baum optimal.

Formal drückt dies das folgende Lemma aus:

Lemma 5
Falls $\mathit{consistent}\, t$, $\mathit{optimum}\, t$,
$a \in \mathit{alphabet}\, t$, b $\notin \mathit{alphabet}\, t$, $\mathit{freq}\, t\, a = w_a + w_b$,
$\forall c \in \mathit{alphabet}\, t.\ w_a \leq \mathit{freq}\, t\, c \wedge w_b \leq \mathit{freq}\, t\, c$, und $w_a \leq w_b$,
dann $\mathit{optimum}\ (\mathit{splitLeaf}\, t\, w_a\, a\, w_b\, b)$

Der Beweis zeigt, dass wenn t optimal und damit günstiger als alle vergleichbaren Bäume v ist, dann auch der entstandene Baum günstiger als alle vergleichbaren Bäume u ist. In u gibt es zwei Symbole c und d, die auf der tiefsten Ebene des Baumes liegen.
Wenn *mergeSibling (swapFourSyms u a b c d) a* ausgehend vom Baum u angewandt wird, werden zunächst a und b Nachbarn, die anschließend verschmolzen werden. Es entsteht ein mit t vergleichbarer Baum v. Mit Lemma 3 kann die Ungleichung weiter vereinfacht werden, indem $mergeSibling$ sowie $w_a + w_b$ eliminiert werden. Der letzte Schritt ist die Anwendung von Lemma 2:

$$\begin{aligned}
&\mathit{cost}(\mathit{splitLeaf}\, t\, a\, w_a\, b\, w_b)\\
&= \mathit{cost}\, t + w_a + w_b\\
&\leq \mathit{cost}\, v + w_a + w_b\\
&= \mathit{cost}\ (mergeSibling\ (\mathit{swapFourSyms}\, u\, a\, b\, c\, d)\ a) + w_a + w_b\\
&= \mathit{cost}(\mathit{swapFourSyms}\, u\, a\, b\, c\, d)\\
&\leq \mathit{cost}\, u
\end{aligned}$$

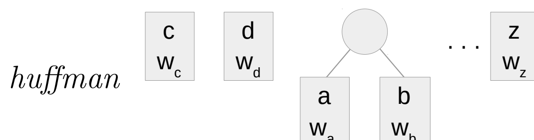
Nun kann die Optimalität des Huffman-Algorithmus bewiesen werden.

Satz
Falls $\mathit{consistent}_F\ ts$, $\mathit{height}_F\ ts = 0$, $\mathit{sortedByWeight}\, ts$, und $ts \neq []$,
dann $\mathit{optimum}\ (\mathit{huffman}\ ts)$

Der Beweis beruht auf Induktion über die Größe des Waldes ts. Als Eingabe wird eine Liste von Blättern erwartet, die aufsteigend nach Gewichten sortiert sind.

Induktionsbasis
Falls ts aus einem einzigen Blatt besteht, wird dieses zurückgegeben. Es hat Kosten 0 und ist damit optimal.

Induktionsschritt
Falls ts aus zwei oder mehreren Knoten besteht, bleibt nach dem ersten Schritt des Algorithmus folgender Term übrig:

huffman


Der neue Baum, der hier an dritter Stelle eingefügt wurde, kann im allgemeinen an jeder Position in der Liste stehen. Die so entstandene Liste entspricht

$$\mathit{huffman}\ (\mathit{splitLeaf}\ \boxed{\begin{matrix}c\\ w_c\end{matrix}}\ \boxed{\begin{matrix}d\\ w_d\end{matrix}}\ \boxed{\begin{matrix}a\\ w_a+w_b\end{matrix}}\ \cdots\ \boxed{\begin{matrix}z\\ w_z\end{matrix}}\ \ w_a\ a\ w_b\ b).$$

Da w_a und w_b minimal sind und somit im ersten Schritt a und b kombiniert werden, sind die Funktionen *huffman* und *splitLeaf* in diesem Fall kommutativ.

$$\mathit{splitLeaf}\ (\mathit{huffman}\ \boxed{\begin{matrix}c\\ w_c\end{matrix}}\ \boxed{\begin{matrix}d\\ w_d\end{matrix}}\ \boxed{\begin{matrix}a\\ w_a+w_b\end{matrix}}\ \cdots\ \boxed{\begin{matrix}z\\ w_z\end{matrix}}\ )\ w_a\ a\ w_b\ b$$

Nach Lemma 5 erhält *splitLeaf* hier die Optimalität, d.h. es muss nur die Optimalität des rekursiven Aufrufs der *huffman*-Funktion gezeigt werden. Diese folgt direkt aus der Induktionshypothese.

Somit wurde gezeigt, dass der Huffman-Algorithmus zu Binärbäumen mit minimaler gewichteter Pfadlänge führt. Der konkrete Beweis in Isabelle benötigt noch einige zusätzliche Hilfslemmata und Aussagen über die Eigenschaften der entstehenden Binärbäume, folgt aber der hier vorgestellten Struktur.

6 Anwendungen

Der Algorithmus kann in der Praxis Einsparungen der Datenmenge von 20 - 90% erzielen [3, S. 385], ist dabei jedoch sehr abhängig von der Verteilung der Häufigkeiten der einzelnen Symbole.

Die Verarbeitung von Kodierungen mit variablen Längen ist auf Computern, die meist auf festen Byte- bzw. Wort-Größen arbeiten komplizierter. Dies ist bei der Verarbeitung von Text problematisch.

Huffman-Codes werden heute häufig als verlustfreier Schritt in Komprimierungsverfahren wie z.B. JPEG oder MP3 eingesetzt. Außerdem kommen sie in vielen Anwendungen als letzter Schritt vor, um die zu vor komprimierten Daten auf möglichst optimale Art abzuspeichern.

Da der Huffman-Algorithmus die Häufigkeiten der im Text enthaltenen Zeichen als Eingabe benötigt, ist keine Echtzeit-Komprimierung möglich. Es existiert jedoch eine dynamische Variante des Huffman-Algorithmus, die auch mit einem Pass eine gute Annäherung an die optimalen Huffman-Codes liefert [5].

Huffman-Codes bestehen immer aus einer ganzzahligen Anzahl Bits. Deshalb besteht die Möglichkeit, mit Arithmetischem Kodieren, wobei Daten als ein einziger Bruch zwischen 0 und 1 mit variabler Genauigkeit dargestellt werden, Daten noch effizienter zu kodieren [6]. Huffman-Codes können als Spezialfall dieser Methode gesehen werden.

Literatur

[1] Huffman, D.A.: *A method for the construction of minimum-redundancy codes*. In: Proc. IRE 40(9), S. 1098–1101, 1952.

[2] Blanchette, Jasmin C.: *Proof Pearl: Mechanizing the Textbook Proof of Huffman's Algorithm*. Journal of Automated Reasoning 43(1), S. 1–18, 2009.

[3] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L.: *Introduction To Algorithms*. 2. Aufl.. Cambridge: MIT Press, 2001.

[4] Knuth, Donald E.: *The Art of Computer Programming : Volume 1: Fundamental Algorithms*. Boston: Addison-Wesley Professional, 1997.

[5] Vitter, Jeffrey S.: *Design and Analysis of Dynamic Huffman Codes*. Journal of the ACM, 34(4), Oktober 1987.

[6] Blelloch, Guy E.: *Introduction to Data Compression*, 2013, unter `http://www.cs.cmu.edu/~guyb/realworld/compression.pdf` (abgerufen am 16.01.2017).