

Codierungstheorie - Optimale Codes

Chrisian Nawroth

16. Mai 2003

1 Optimale Codes

1.1 Motivation

Bei der Übertragung und Speicherung von Datenmengen ist es sinnvoll, diese Daten in möglichst optimierter Form zu vorzuhalten. Dadurch können kostenintensive Ressourcen wie Speicherplatz und Übertragungsbandbreite eingespart werden. Ein Nachteil der Code-Optimierung ist allerdings, dass für das Codieren und Decodieren von Codes unter Umständen ein hoher Rechenaufwand nötig ist, was sich negativ auf die Geschwindigkeit eines Systems auswirken kann. Daher ist immer genau zu prüfen, welches Codierungsverfahren gewählt wird und ob eine Codeoptimierung überhaupt sinnvoll ist. Im folgenden wird das HUFFMAN-Verfahren vorgestellt, ein einfach zu implementierendes und zu verstehendes Optimierungsverfahren.

1.2 Huffman-Codierung

1.2.1 Grundlegendes Prinzip

Die Idee des Huffman-Verfahrens liegt darin, häufig wiederkehrende Codewörter mit möglichst optimalen "Abkürzungen" zu codieren. Am besten lässt sich dieses Vorgehen anhand eines durchschnittlichen deutschen Textes verdeutlichen (angenommen wird ein Alphabet mit 26 Buchstaben): In einem normalverteilten deutschen Text kommt der Buchstabe 'e' am häufigsten vor. Daher ist es sinnvoll, diesen Buchstaben mit einer möglichst kurzen Bit-Folge abzuspeichern. Man könnte das 'e' z.B. mit einer binären '0' codieren. Dem Buchstaben 'y' kommt im Deutschen am seltensten vor, und daher ist für ihn das längste Codewort zu wählen. Wählt man eine derartige Codierung, ist allerdings strikt darauf zu achten, dass die FANO-Bedingung erfüllt ist:

Satz: *Wenn kein Codewort Anfangswort eines anderen Codewortes ist, dann ist jede codierte Zeichenfolge eindeutig dekodierbar*

Für das genannte Beispiel ('e' := 0) würde dies bedeuten, dass es keine weiteren Codes geben kann, die mit einer 0 beginnen, da in diesem Fall nicht mehr eindeutig festzustellen wäre, ob es sich um ein 'e' und einen weiteren Buchstaben handelt, oder ob es ein einziger von 'e' verschiedener Buchstabe ist. Der Huffman-Algorithmus berücksichtigt die Einhaltung der FANO-Bedingung

1.2.2 Algorithmus

Eingabe: Eindeutiger Code, der die FANO-Bdingung erfüllt (z.B. Text)

Ausgabe: Binärer Baum mit der Knotenmakierung p und der Kantenmakierung h

Vorgehen:

- Erzeuge für jedes Symbol x , das im zu codierenden Text t vorkommt, und makiere diesen Knoten mit der Häufigkeit mit der x im Text vorkommt.
- Suche zwei Knoten u v mit minimaler Makierung $p(u)$ und $p(v)$, zu denen noch keine Kante hinführt.
- Erzeuge einen neuen Knoten w , und verbinde diesen mit u und v . Markiere die eine Knate mit 0, die andere Kante mit 1. Markiere den Knoten w mit $p(u)+p(v)$
- Wiederhole die letzten beiden Schritte solange, bis es keinen Knoten gibt, zu dem keine Kante mehr hinführt.

1.2.3 Beispiel

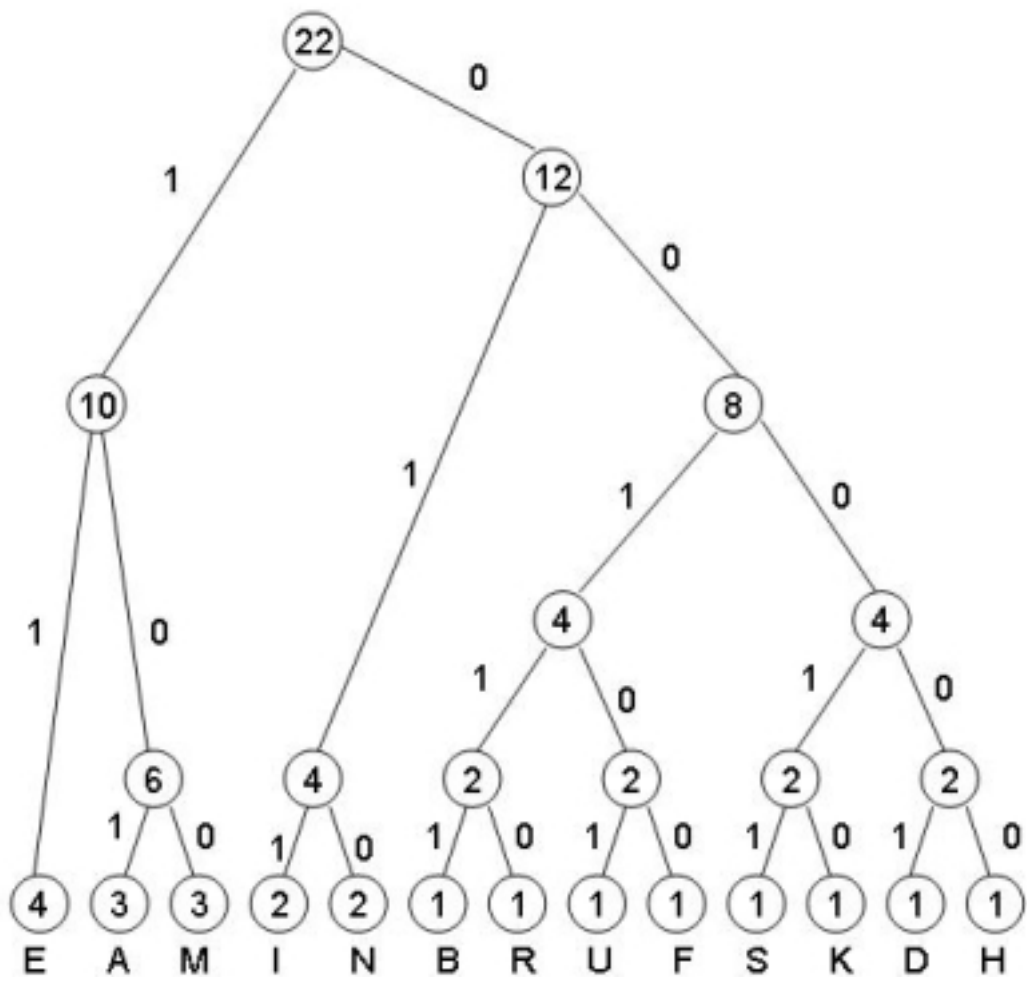
Das folgenden Beispiel soll codiert werden:

BERUFSAKADEMIE MANNHEIM

Die Verteilung der Buchstaben sieht folgendermaen aus:

Buchstabe	Häufigkeit im Text
E	4
A	3
M	3
I	2
B	2
N	2
R	1
U	1
F	1
S	1
K	1
D	1
H	1

Abfolge einiger exemplarischer Schritte der HUFFMAN-Codierung.
Der entglte Baum, mit den Kantenbezeichnungen:



Nach der Huffman-Codierung ergibt sich folgende Tabelle, die sich am Baum ablesen lässt. Im Vergleich dazu eine Standard ASCII-Codierung.

Buchstabe	Hufigkeit im Text	HUFFMAN-Code	HUFFMAN-Bits	ASCII-Bits	Ersparnis
E	4	11	8	32	24
A	3	101	9	24	15
M	3	100	9	24	15
I	2	011	6	16	10
N	2	010	6	16	10
B	1	00111	5	8	3
R	1	00110	5	8	3
U	1	00101	5	8	3
F	1	00100	5	8	3
S	1	00011	5	8	3
K	1	00010	5	8	3
D	1	00001	5	8	3
H	1	00000	5	8	3
Gesamt:	22		78	176	98

Gegenüber einer herkömmlichen ASCII-Codierung spart man also 98 Bits ein, das entspricht 55,7 %.