Seminar

Codierungstheorie

Lempel-Ziv-Verfahren

Informatik

Sommersemester 2005

Inhalt

Wörterbuchmethoden

Wörterbuchmethoden sind einfache Methoden der Komprimierung. Auch die Lempel-Ziv-Codierung, auf die hier insbesondere eingegangen wird, zählt zu den Wörterbuchmethoden.
Wörterbuchmethoden verwenden, wie der Name schon sagt ein Wörterbuch. Grundsätzlich unterscheidet man zwischen zwei Arten von Wörterbuchmethoden: *statische* und *dynamische Wörterbuchmethoden*. Bei statischen Wörterbuchmethoden ist das Wörterbuch schon vor der Kompression fest vorgegeben und wird nicht mehr verändert. Bei dynamischen Wörterbuchmethoden wird das Wörterbuch erst im Laufe der Kompression aus den zu komprimierenden Daten erstellt.
Die Kompression einer Wörterbuchmethode erfolgt dadurch, dass Zeichenfolgen, die häufig in den zu komprimierenden Daten vorkommen in dem Wörterbuch gespeichert werden, und die Ausgabe der Codierung dann statt aus den Daten aus Zeigern auf die Wörterbucheinträge besteht.

Statische Wörterbuchmethoden

Da bei statischen Wörterbuchmethoden das Wörterbuch schon vor der Komprimierung feststeht muss man nicht nur das Quellalphabet kennen, sondern auch eine Wahrscheinlichkeitsverteilung über diesem Alphabet. Diese Wahrscheinlichkeitsverteilung gibt an, welche Zeichenfolgen statistisch am häufigsten vorkommen.
Da man Wörterbuchmethoden häufig bei der Komprimierung bei Texten verwendet werden die nächsten Beispiele eine Quellalphabet $A=\{0,1\}^b$ verwenden.
Die Wahrscheinlichkeitsverteilung sei gegeben durch: $p(0)=p_0$ und $p(1)=p_1$ auf $\{0,1\}$.
Das Wörterbuch W hat 2^c Einträge wobei $c<b$.
Mit Hilfe der Wahrscheinlichkeitsverteilung werden also die 2^c häufigsten Folgen ermittelt und im Wörterbuch gespeichert.
Um nun eine Folge $a \in A$ zu codieren wird überprüft, ob diese Folge im Wörterbuch vorkommt, also ob $a=W_i$ für $i=0,\ldots,2^c-1$:
→ ja, dann wird dir Folge mit $0(i)_2$ codiert.
→ nein, dann wird die Folge mit 1a codiert.
Das vorangestellte Bit ist eine so genannte *flag*, die angibt ob die darauf folgende Binärfolge eine unkomprimierte Teilfolge oder ein Index des Wörterbuchs ist. Der Index i soll immer in c Bits angegeben werden.

Beispiel:

$A=\{0,1\}^5$
$|W|=2^2$
$p(0)=¾$
$p(1)=¼$
Bestimme zuerst die vier häufigsten Folgen und speichere diese im Wörterbuch:
$W_0=00000=0^5 \Rightarrow p(W_0)=\frac{243}{1024}$
W_1, W_2, W_3 sind Folgen mit vier Nullen und einer 1. Hiervon gibt es insgesamt 5 Stück. Es passen nur noch 3 ins Wörterbuch. Welche man hier nimmt ist nicht fest vorgelegt:

$W_1=10^4=10000$

$W_2=01000$

$W_3=00100$

Es sei nun die Folge 00000 00001 10010 00100 gegeben, die komprimiert werden soll.
W_0 $\notin W$ $\notin W$ W_3

Ausgabe:

000 100001 110010 011
1+*c* 1+*b* 1+*b* 1+*c*

Eingabebits: 20

Ausgabebits: $2 \cdot (1+c) + 2 \cdot (1+b) = 18$

Bemerkung:

Man erzielt allerdings nicht immer eine Komprimierung. Wenn man bei obigem Beispiel die Eingabefolge 00001 00001 00010 00010 hätte, welche die gleiche Wahrscheinlichkeitsverteilung und die gleiche Eingabelänge hat, wird die Folge bei der Kompression nicht kleiner sondern größer. Es kommen noch 4 Bits (die flags) hinzu, da keine Teilfolge im Wörterbuch vorhanden ist.
=> Diese Codierungsmethode ist nicht optimal.

Dynamische Wörterbuchmethoden – Das LZ-Verfahren

Die Lempel-Ziv-Codierung hat einige wesentliche Vorteile gegenüber dem eben kennengelernten Verfahren der Kompression der Daten:

- Die LZ-Codierung benötigt keine Wahrscheinlichkeitsverteilung
- Das Wörterbuch wird dynamisch angelegt

Es gibt viele verschiedene Varianten der LZ-Codierung. Diese Varianten stammen alle von zwei Komprimierungsverfahren ab, dem **LZ77** und dem **LZ78**. Diese Algorithmen wurden von Abraham Lempel und Jacob Ziv in den Jahren 1977 und 1978 entwickelt.
Die bekannteste Abwandlung ist die 1984 von Terry Welch entwickelte LZW-Codierung.

Der LZ78

Auch hier kann man von einem binären Quellalphabet $A = \{0, 1\}$ ausgehen. Da die LZ-Codierung allerdings keine feste Blocklänge benötigt, wie eine Codierung mit einem statischen Wörterbuch, werden also Folgen der Form $\{0, 1\}^*$ codiert.

Definition (Parsing):

Ein *Parsing* einer Folge $x \in \{0, 1\}^*$ ist eine Aufteilung von x in Teilfolgen $y_1, \ldots, y_c$. Das Parsing heißt disjunkt, wenn die y_i paarweise verschieden sind.

Das Lempel-Ziv-Parsing ist ein disjunktes Parsing.

Das Lempel-Ziv-Parsing ist ein disjunktes Parsing mit folgender Definition:
Eingabe: $x = x_1, \ldots, x_n$
Ausgabe: $y_1 = x_1$
$y_2, \ldots, y_c$ ist die kürzeste Teilfolge von x, die nicht schon im Parsing vorkommt.

Beispiel (Parsing):

x=1011010100010

Das Parsing für diese Eingabefolge wäre: 1, 0, 11, 01, 010, 00, 10
y_1 y_2 …… y_7

Lemma

Es folgt aus der Definition des Lempel-Ziv-Parsings, dass es zu jeder Folge y_i ein Bit b und einen eindeutigen Index $0 \le j \le i$-1 gibt, so dass: $y_i = y_j b$

Beispiel: $y_5 = 010 = y_4 0$
$y_7 = 10 = y_1 0$

Der Algorithmus

1. Erzeuge das LZ-Parsing der zu codierenden Daten.
2. Jedes y_i wird durch (j,b) codiert. Wobei j durch $\lceil \mathrm{ld}(c) \rceil$ Bits codiert wird.

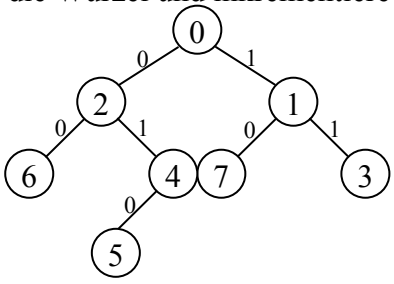
Beispiel mit der Eingabefolge von oben:

Ausgabe	Wörterbuch	
	Index	Eintrag
(000,1)	1	1
(000,0)	2	0
(001,1)	3	11
(010,1)	4	01
(100,0)	5	010
(010,0)	6	00
(001,0)	7	10

Der Algorithmus wird mit Hilfe eines *Binärbaums* abgearbeitet. Am Anfang existiert nur die Wurzel mit Null als Eintrag. Weiter gibt es einen Zeiger, der zu Beginn auf die Wurzel zeigt und einen Zähler z:=0, der jeweils den größten Knoteneintrag speichert.

Durchführung: Die Bits des Eingabestroms werden nach und nach eingelesen und es wird jedes Mal die Frage gestellt: „*Führt aus dem Knoten auf den der Zeiger zeigt eine Kante, die mit* x_i *gelabelt ist*“:

- ➔ Ja: setze Zeiger auf den Knoten, zu dem die Kante führt.
- ➔ Nein:
  - füge einen neuen Knoten mit Label z+1 ein
  - Label die Kante zwischen den Knoten mit x_i
  - Sei j Index des Knotens mit dem aktuellen Zeiger, dann gebe (j, x_i) aus, setze den Zeiger auf die Wurzel und inkrementiere den Zähler um 1.



Der LZ77

Bei der LZ77-Komprimierung wird der eingelesene Text als Tabelle genutzt. Der Algorithmus arbeitet mit zwei Fenstern:

- ➔ dem *Search-Buffer*: Der Search-Buffer enthält bereits codierte Zeichen. Hier wird nach Übereinstimmungen zu den noch zu codierenden Zeichen gesucht.
- ➔ dem *Look-Ahead-Buffer*: Der Look-Ahead-Buffer schließt sich unmittelbar an den Search-Buffer an und enthält die als nächstes zu codierenden Zeichen.

Der Search-Buffer ist im Allgemeinen viel größer als der Look-Ahead-Buffer.

Beispiel:

Search-Buffer	Look-Ahead-Buffer
`sir_sid_eastman_easily_t`	`eases_sea_sick_seals`

Der Algorithmus

1. Durchsuche den Search-Buffer von rechts nach dem String, bzw. den ersten Zeichen des Look-Ahead-Buffers und versuche eine möglichst lange Übereinstimmung zu finden. („`e`“ von `easily` => Übereinstimmung 3 Zeichen lang: „`eas`“) Sollten mehrere Übereinstimmungen der gleichen Länge auftreten wird die zuletzt gefundene verwendet. (nochmals „`eas`“ von `eastman`)
2. Die Übereinstimmung wird durch (*Offset*, *Länge*, „*nächstes Zeichen*“) codiert. *Offset* ist der Index des ersten Zeichens der Übereinstimmung. (16, 3, „e“)
3. Verschiebe die Fenster um *Länge*+1 Zeichen nach rechts.

Beispiel:

Search-Buffer	Look-Ahead-Buffer	Ausgabe
	AABCAAABCD	(0, 0, „A“)
A	ABCAAABCD	(1, 1, „B“)
AAB	CAAABCD	(0, 0, „C“)
AABC	AAABCD	(4, 2, „A“)
AABCAAA	BCD	(5, 2, „D“)
AABCAAABCD		

Bemerkungen

1. Das Muster, das im Search-Buffer gesucht wird, kann in den Look-Ahead-Buffer hineinlaufen. Beispiel:

Search-Buffer	Look-Ahead-Buffer	Ausgabe
`die katze ruft miau`	`miaumiaumiau`	(4, 11, „u“)

   Sei *l* die Länge des Look-Ahead-Buffers, dann ist die maximale Länge des gefundenen Strings *l*-1.

2. Man erhält natürlich eine größere Kompression, wenn man den Search-Buffer vergrößert, allerdings wird der die Suche dann natürlich auch aufwendiger.

Dekodieren mit dem LZ77

Das Dekodieren erfolgt entgegengesetzt der Codierung. Man verwendet die gleichen Fenster und kopiert die Teilfolgen an die Stelle des Offsets.
Anschließend fügt man noch das nächste unkomprimierte Zeichen an und führt die Verschiebung durch.

Eingabe		Ausgabe
(0, 0, „A“)		A
(1, 1, „B“)	A	AB
(0, 0, „C“)	AAB	C
(4, 2, „A“)	AABC	AAA
(5, 2, „D“)	AABCAAA	BCD
	AABCAAABCD	

Fazit LZ77 / LZ78

Der LZ77 wird häufig bei Archiven verwendet und ist hierfür auch gut geeignet, da das Dekodieren erheblich einfacher ist als das Codieren, denn die Suche nach Übereinstimmungen während der Codierung ist sehr aufwendig.
Bei Archiven komprimiert man einmal, erstellt das Archiv und greift später nur noch darauf zu und führt dementsprechend die aufwendige Codierung nur einmal durch.
Man kann beweisen, dass die Lempel-Ziv-Codierung über einem Quellalphabet {0, 1} und einer beliebigen unbekannten Wahrscheinlichkeitsverteilung im Gegensatz zu statischen Wörterbuchmethoden asymptotisch optimal ist, also ein viel besseren Codierungs-Algorithmus liefert.

Die Huffman-Codierung

David A. Huffman war ein amerikanischer Computerpionier und hat 1952 die *Huffman-Codierung* entwickelt. Die Huffman-Codierung ist eine *Entropiekodierung*, die im Gegensatz zum Stringersatzverfahren (LZ77 und LZ78) jedem einzelnen Zeichnen eines Textes eine unterschiedlich lange Folge von Bits zuordnet.
Da eine bestimmte Mindestanzahl von Bits notwendig ist, um alle Zeichen voneinander zu unterscheiden, kann die Anzahl der Bits, die den Zeichen zugeordnet werden, nicht unbegrenzt klein werden.
Entropiekodierer werden häufig mit anderen Kodierern kombiniert. Dies ist der Grund, warum hier davon geredet wird, denn ich werde später erklären, wie man die Huffman-Codierung mit dem LZ77 kombiniert und so die bekannte *Gzip-Komprimierung* erhält.

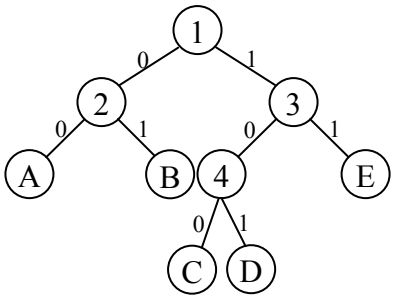
Grundidee

Um eine speichersparende Darstellung zu erhalten müssen die Zeichen mit einer unterschiedlichen Anzahl von Bits kodiert werden. Diese Bitfolgen müssen jedoch *präfixfrei* sein um eine eindeutige Abbildung zu erhalten. Wenn man nämlich „A“ mit 10, „B“ mit 01 und „C“ mit 0 codiert, wird die Zeichenkette „ABC“ kodiert mit „10010“. Diese Folge wird jedoch auch von der Zeichenkette „ACA“ erzeugt.

Präfixfrei bedeutet, dass keine Bitfolge, die für ein einzelnes Zeichen steht am Anfang eines anderen Zeichens stehen darf.
Um nun einen präfixfreien Code zu erzeugen verwendet man einen binären Baum zur Darstellung des Codes. In dem Baum stehen die Blätter für die zu kodierenden Zeichen, während die inneren Knoten für die zu schreibenden oder gelesenen Bits stehen.

Beispiel:

Der Baum rechts ist von der gewünschten Form. Die inneren Knoten sind nur durchnummeriert, um sie benennen zu können.
Um nun zum Beispiel die Zeichenfolge ACDC zu kodieren, werden die den entsprechenden Zeichen zugewiesenen Bitfolgen hintereinander ausgegeben: 00100101100.



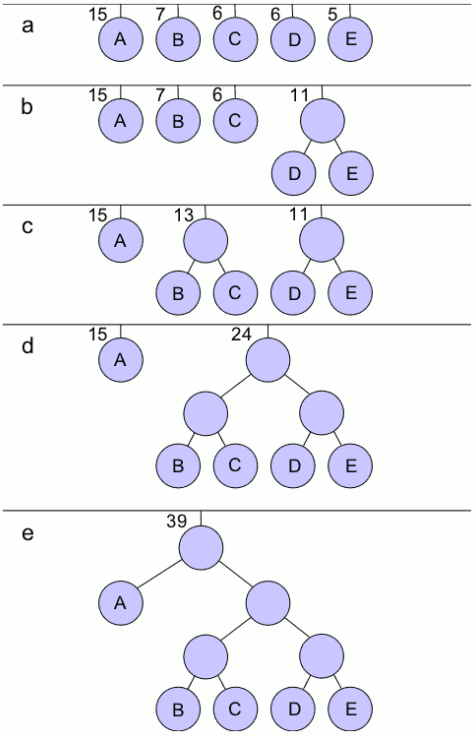
Das verbleibende Problem ist die Erstellung eines Baumes, bei dem das durchschnittliche Ermitteln der erforderlichen Bitfolge eines Zeichens im Mittel möglichst klein ist.
Die Shannon-Fano-Codierung und die Huffman-Codierung sind zwei unterschiedliche Algorithmen zur Konstruktion dieser Bäume.
Da der vom Shannon-Fano-Algorithmus erzeugte Baum jedoch nicht optimal ist, entwickelte Huffman 1952 seinen Algorithmus, welcher in der Tat nachweisbar immer einen optimalen Baum liefert.

Der Algorithmus

1. Erstelle einen "Wald" mit Bäumen, für jedes Zeichen. Diese Bäume enthalten nur einen Knoten: das Zeichen
2. suche die beiden Bäume im Wald, die für die Zeichen mit der kleinsten Wahrscheinlichkeit stehen. Entferne diese Bäume aus dem Wald. Erstelle einen neuen Baum, der die beiden entfernten Bäume als Subbaum hat. Füge diesen Baum in den Wald ein. Benutze dabei die Summe der Wahrscheinlichkeiten der Unterbäume.
3. Wiederhole, bis nur noch ein Baum übrig ist.

Beispiel

Abschnitt a zeigt den ersten Schritt des Algorithmus. Die Zahlen sind die Auftretungswahrscheinlichkeiten der einzelnen Zeichen.
Da C, D, und E die Zeichen mit der niedrigsten Wahrscheinlichkeit sind, werden sie in b entfernt (Schritt 2), und es wird ein neuer Baum mit der Summer der Wahrscheinlichkeiten und D E als Subbäume erstellt.
Dieses Verfahren wird durchgeführt, bis schließlich A mit der größten Wahrscheinlichkeit mit dem anderen Baum zusammengeführt wird. A ist das am häufigsten auftretende Zeichen, und benötigt nun nur ein Bit um es zu kodieren. Die andere Zeichen benötigen jeweils 3 Bit.



LZ77 und Huffman im gzip

gzip ist die Kurzform für GNU ZIP. Dabei handelt es sich um ein Kompressionsprogramm, das auch im Quelltext erhältlich ist. Verwechselt werden darf das Format aber nicht mit dem Zip-Dateiformat unter Windows, welches von gzip unabhängig ist.
gzip ist ein Komprimierungsprogramm, das praktisch für alle Computerbetriebssysteme verfügbar ist. Es bietet einen guten Kompressionsgrad und ist frei von patentierten Algorithmen. Es wurde von *Jean-Loup Gailly* ursprünglich entwickelt um unter Unix das alte Programm *compress* zu ersetzen.
gzip basiert auf dem *deflate*-Algorithmus, der eine Kombination aus LZ77 und Huffman-Codierung ist. Hierbei werden die bei der LZ77-Codierung entstehenden Zeichenlängen und Distanzcodes mit Hilfe der Huffman-Codierung gespeichert.

Quellen

- Datenkompression nach Lempel-Ziv, Christian Näger, 1997
- Implementierung des Packverfahrens im Programm Gzip, Ingo Rohloff, 1996
- http://www.data-compression.de
- http://www.lempel-ziv.de
- http://www.wikipedia.de