Seminar über Algorithmen, SS2004

Textkompression

von Christian Grümme und Robert Hartmann

1. Einleitung

Textkompression wird zur Verringerung des Speicherbedarfs und der Übertragungskapazität von allgemeinen Daten verwendet. Im folgenden werden vier Algorithmen vorgestellt, die Daten informationsverlustfrei komprimieren. Die Leistung der Verfahren misst man sowohl anhand ihrer Kompressionsrate also auch der Laufzeit und des Speicherbedarfs. Die komprimierte Datei hat eine Größe von 30% bis 50% der originalen Datei. Beispielhafte Statistiken der ausgewählter Algorithmen befinden sich in der Zusammenfassung in Kapitel 6.

2. Statische Huffman Kompression

2.1Kodierung

Die Kodierung ist in 3 Teile unterteilt:

- Häufigkeit jedes Zeichens zählen
- Präfixcode erzeugen
- Text kodieren

Um ein doppeltes Durchlaufen des Textes im 1. Schritt zu vermeiden, kann man auch Statistiken zu Rate zählen. Dabei sollte man achten, dass diese auf die gleiche Art von Text basiert. (gleiche Sprache, Programmcode, Wörterbuch). Lässt sich keine passende Statistik auffinden oder will man eine optimale Grundlage, muss man die Zeichenhäufigkeit aber selbst ermitteln.
Der 2. Schritt erzeugt einen Baum, in welchem der Präfixcode gespeichert ist. Sei dafür *freq(a)* die aus einer Statistik oder anhand des Zählens ermittelte Häufigkeit des Zeichens *a*. Zuerst wird für jedes *a* ein einblättriger Baum T mit *weight(T)=freq(a)* und *label(T)=a* erzeugt. Danach sucht man die Bäume *T1, T2* mit den geringsten Gewichten und vereinigt sie zu einem neuen Baum *T* mit *weight(T)=weight(T1)+weight(T2)* und *leftChild(T)=T1*, *rightChild(T)=T2*. Das wiederholt man solange, bis nur noch ein Baum übrig ist. Um aus diesem dann den Präfix für ein Zeichen *a* abzulesen, sucht man von der Wurzel *r* beginnend das Blatt mit Zeichen *a*, und merkt sich bei jedem Schritt eine 0, falls man zu einem linken Kind gegangen ist und eine 1, falls man zu einem rechten Kind gegangen ist. Für die Implementierung muss man allerdings bei *a* beginnend die Elternzeiger bis zur Wurzel verfolgen und erhält dann den Präfixcode in der umgekehrten Reihenfolge.
Im 3. Schritt wird dann der Quelltext durchlaufen und für jedes Zeichen a sein Codewort in die Zieldatei geschrieben. Da man zur Dekodierung aber den Codebaum braucht, muss dieser im Kopf der Datei gepseichert werden. Das erhöht zwar die Anzahl der zu speichernden Bits, ist aber unumgänglich.

2.2Dekodierung

Zuerst wird der im Kopf der Datei kodierte Codebaum ausgelesen. Danach parst man den Rest der Datei ab, und geht beim Lesen einer 0 zum linken Kind und bei einer 1 zum rechten Kind des aktuellen Knotens. Begonnen wird bei der Wurzel. Erreicht man ein Blatt *l*, wird *label(l)* ausgegeben und mit dem nächsten Bit wieder bei der Wurzel

begonnen.

2.3Laufzeit

Sei n=|Text|, s=|Alphabet|. Dann dauert das Durchlaufen des Textes zur Bestimmung der Häufigkeit jedes Zeichens *O(n)*. Das Erzeugen des Baums dauert *O(s)* für das Erstellen der s Startbäume und auch *O(s)* für das schrittweise Vereinigen. (In jedem Schritt ein Baum weniger => s-1 Schritte). Das Ablesen der Codewörter dauert im schlimmsten Fall *O(s^2)*. Das eigentliche Kodieren braucht *O(n)* Schritte, da wieder der gesamte Text durchlaufen wird und bei jedem Zeichen konstant viel Operationen ausgeführt werden.
Nimmt man jetzt nur n als Eingabegröße, haben wir *O(n).* Ansonsten *O(n + s^2).*

3. Dynamische Huffman Kompression

Die statische Huffman Kompression hat zwei Nachteile:

- Man kennt die Häufigkeit der einzelnen Zeichen nicht und muss deswegen den Text zweimal durchgehen.
- Man muß den Codebaum in die Datei einfügen.

Die dynamische Huffman Kompression hat diese Nachteile nicht. Sie erzeugt einen Baum der jedes mal verändert wird, wenn ein neues Zeichen gelesen wird.

3.1 Kodierung

Der binäre Baum, der beim Enkodieren erzeugt wird, der Huffman Baum, hat die bis dahin eingelesenen Zeichen als Label in seinen Blättern gespeichert und die Häufigkeit als dessen Gewicht. Es existiert ein spezieller Knoten mit dem Label „ART“ der als Platzhalter für noch nicht eingelesene Zeichen dient. Außerdem sind die Knoten entsprechend ihrer Position durchnummeriert, so dass zu jedem ungeradem i, i+1 sein Geschwisterknoten ist. Der Baum wird mit dem Knoten „ART“ als Wurzel mit dem Gewicht 1 und der Nummer 0 initialisiert. Jedes mal wenn ein Zeichen eingelesen wird, wird es im Baum gesucht und den Weg dahin (1 links, 0 rechts) gemerkt, der wird in die Datei geschrieben und die Huffman Baum Eigenschaften wiederhergestellt (*DynHufTree-Update*). Falls das Zeichen noch nicht im Baum vorkommt, wird sich der Weg zum Knoten „ART“ gemerkt und wird in die Datei geschrieben. Zusätzlich wird der ASCII-Code des neuen Zeichens in die Datei geschrieben, in den Baum wird anstelle des „ART“-Knoten eine neuer Knoten mit dem Gewicht 1 gefügt, der als linkes Kind einen neuen Knoten, mit dem Gewicht 0 und das neue Zeichen als Label hat, und als rechtes Kind den Knoten „ART“ mit dem Gewicht 1. Dann werden wieder die Huffman Baum Eigenschaften wiederhergestellt. Dazu wird der Knoten mit dem gerade eingelesenen Zeichen betrachtet. Sein Gewicht wird um eins erhöht. Dann wird nacheinander das Gewicht eines jeden Knoten mit dem Gewicht von dem betrachteten Knoten verglichen. Sobald einer ein kleineres Gewicht hat, werden die Knoten vertauscht und nun wird der Vater-Knoten des betrachteten Knotens der neue betrachtete Knoten. Diese Prozedur wird wiederholt bis die Wurzel der betrachtete Knoten ist.

3.2 Dekodierung

Die Dekodierung läuft symmetrisch zu der Encodierung, deswegen werden hier die gleichen Einfüge- und Update-Methoden wie bei der Encodierung benutzt. Es wird beim Einlesen der Zeichen derselbe Baum erzeugt wie beim Encodieren. Sobald der eingelesene Weg zum „ART“-Knoten führt, werden die nächsten 8-bit eingelesen und in

das zugehörige ASCII Zeichen umgewandelt und auf die selbe Art in den Baum eingefügt wie bei der Encodierung.

3.3 Laufzeit

Die Initialisierung ist konstant, danach wird dann die Schleife ausgeführt, in der einzeln die Zeichen (|Zeichen| = n) eingelesen und kodiert werden und jedes mal noch der Huffman Baum aktualisiert. Dass kodieren dauert bei einem entarteten Baum O(n), also bei einen ganzen Durchlauf. Bei Aktualisierung des Huffman Baumes wird der gesamte bis hierhin entstandene Baum durchlaufen => O(n). Zusammen mit der äußeren Schleife => $O(n^2)$.
Da die Decodierung symmetrisch zur Encodierung ist, gilt die selbe Laufzeit auch für die Decodierung.

4. Arithmetisches Komprimieren

Das Verfahren der arithmetischen Kodierung weicht von der Idee der Huffman Codierung ab, Symbole durch eine Anzahl von Bits zu ersetzen. Es erzeugt einen Code für den gesamten Datenstrom und erreicht daher eine sehr hohe Effizienz. Ein weiterer Unterschied ist, dass ein Teil des Codes nur dekodiert werden kann, wenn der gesamte vorherige Teil bekannt ist. Bei Huffman hingegen, kann an verschiedenen Stellen angesetzt werden.

4.1Kodierung

Der Algorithmus weißt dem Text eine Zahl zwischen 0 und 1 zu. Bevor man mit dem eigentlichen Kodieren beginnen kann, muss allen Symbolen a_i des zugrundeliegenden Alphabets ein Intervall $I(a_i)=[l_i, h_i[$ zugewiesen werden. Dabei ist ein Intervall $I(a_i)$ länger, je häufiger a_i im Text vorkommt. Auch hier kann wieder auf schon bekannte Statistiken zur Initialisierung zurückgegriffen werden. Weiterhin gilt $h_i=l_{i+1}$, $l_0=0$ und $h_{|Alphabet|}=1$.
Man beginnt mit dem Intervall *[low, high[= [0,1[*. Wird ein Symbol a_i gelesen, aktualisiert man *low* und *high* folgenderweise:

$low = low + (high-low) * l_i$

$high = low + (high-low) * h_i$

Danach speichert man den Nachkommateil in binärer Darstellung. Zur Dekodierung wird allerdings nur *low* gebraucht.

4.2Dekodierung

Bei gegebenem *low* sucht man zuerst das Intervall $I(a_i)$ in dem *low* liegt, gibt a_i aus und setzt $low = (low-l_i) / (h_i-l_i)$. Gilt $low=0$ ist der Prozess beendet.

4.3Implementierung

Das Problem bei der Implementierung ist die endliche Genauigkeit mit der Computer arbeiten. Bei sehr großen Texten kann es daher zu Rundungsfehlern und somit Informationsverlust kommen. Zur Lösung bildet man das Intervall [0,1[auf $[0, 2^N-1[$ mit festem N ab. Das Codeword in diesem Fall entspricht dem gemeinsamen Präfix der beiden Intervallgrenzen in jedem Schritt. Weiterhin werden

freq[i] = Häufigkeit von a_i im Text +1 und

cum-freq[i] = Summe der Häufigkeiten von a_{i+1} bis $a_{|Alphabet|}$

gebraucht. Um eine Reihenfolge der a_i festzulegen, werden sie nach Häufigkeit im bisher gelesenen Textpräfix absteigend sortiert und müssen nach jedem Schritt neu

geordnet werden. Beachte, dass *cum-freq[0]* -|∑| die Anzahl aller bisher gelesenen Symbole im Text ist. Nachdem ein Symbol a_i gelesen wird, werden die Intervallgrenzen *low* und *high* neu berechnet,

*low = low + ((high-low+1)*cum-freq[i]) / cum-freq[0]*

*high = low + ((high-low+1)*cum-freq[i-1]) / cum-freq[0] - 1*

und bei zu kleinem Intervallbetrag skalliert. Wurde der gemeinsame Präfix gesendet, shiftet man *low* und *high* nach links und füllt sie mit 0'en bzw. 1'en.
Die Dekodierung funktioniert in genau der umgekehrten Reihenfolge. Es werden N Bit gelesen, für den diesen entsprechende Wert *value* ein passender Index *i* für *cum-freq[i]* gesucht, a_i ausgegeben, die Intervalle angepasst und die Tabelle mit *freq, cum-freq* neu geordnet. Die beim Kodieren geschriebenen Präfixe der Intervallgrenzen werden zur Berechnung von *value* benutzt.

4.4Laufzeit

Die erste Version läuft den Text genau einmal durch, führt konstant viele Operationen aus und gibt am Ende nur eine Zahl zurück. Die Laufzeit ist daher *O(n)*, wenn n die Länge des Textes ist.
Die in der Praxis verwendete Version läuft den Text auch genau einmal durch, führt jedoch nicht bei jedem Zeichen gleich viele Operationen aus. Das Verändern der Intervallgrenzen und Senden des gemeinsamen Präfixes tritt höchstens *2*N* auf. Dazu konstant viele Operationen. Das Updaten der Tabellen für *freq* und *cum-freq* dauert maximal *|Alphabet|,* das Sortieren der Tabelle geht in *|Alphabet|*log(|Alphabet|)*. Nimmt man die Größe des Alphabets als konstant an, fällt diese für sehr große Intervalle nicht mehr ins Gewicht. Daher auch hier *O(n).*

5. Lempel-Ziv-Welsh Komprimierung

Die Grundidee all dieses Verfahren ist es eine Zeichenkette, die schon einmal gesendet/gespeichert wurde und nun wieder auftaucht, durch einen Zeiger auf die Stelle wo sie eher auftauchte oder durch einen Zeiger auf ein Wörterbuch zu ersetzen. Damit soll die Länge der zu codierenden Zeichenkette beträchtlich gekürzt werden.

5.1Kodierung

Zuerst wird ein Wörterbuch (Hier: eine Hash-Tabelle) angelegt, indem man schon mal alle einzelnen ASCII – Zeichen einfügt (256). Dann wird immer einzeln ein Zeichen *a* gelesen und geschaut, ob es schon, zusammen mit der vorherigen Zeichenkette *w* (am Anfang ein leerer String), im Wörterbuch vorkommt. Wenn *wa* nicht vorkommt, wird der Index (bei der Hash-Tabelle der Schlüssel) von *w* in die Datei geschrieben und *wa* in das Wörterbuch eingefügt.

5.2 Dekodierung

Die Decodierung ist Symmetrisch zur Encodierung. Erst wird wieder ein Wörterbuch mit den ASCII-Zeichen (256) angelegt. Zuerst wird ein Schlüssel aus der Datei gelesen und das zugehörige Zeichen *w*, in die neue Datei geschrieben. Dann wird der nächste Schlüssel gelesen. Falls er kleiner ist, als das Wörterbuch, dann wird der zugehörige String *a* geschrieben und *w+firstChar(a)* in das Wörterbuch eingefügt. Nun wird *w = a* gesetzt und das Nächste Zeichen gelesen. Wenn nicht, dann tritt ein Sonderfall ein, da das gelesene Zeichen noch nicht im Wörterbuch steht. Dieser Fall tritt nur auf, wenn *w+a*, die Form *aXa*. Jetzt wird einfach *w+first(w)* in das Wörterbuch eingefügt,*w = first (w)* gesetzt, w geschrieben und das Nächste Zeichen gelesen.

5.3 Laufzeit

Die Initialisierung des Wörterbuches erfolgt in konstanter Zeit. Die while - Schleife wird n mal durchlaufen (|Zeichen| = n). Alles in der Schleife braucht konstante Zeit, bis auf das Suchen des Indexes zu einem Zeichen. Dieses braucht O(n). => $O(n^2)$ für die Enkodierung und auch für die Dekodierung, da sie symmetrisch sind.

6. Zusammenfassung

In einer Implementierung der Verfahren in Java ergaben sich folgende Kompressionsraten:

	Statische Huffman	LZW
Text1: vorher	187248	187248
nachher	161829	73386
komprimiert auf	86,42%	39,19%
Text2: vorher	4669040	4669040
nachher	3966527	2932158
komprimiert auf	84,95%	62.80%
Text3: vorher	8231784	8231784
nachher	7025153	5120993
komprimiert auf	85,34%	62,21%

Text1 besteht aus Tagebucheinträgen auf Deutsch, Text2 enthält den Text Gulliver's Travels (engl.) und Text3 ist des erste Buch der „Herr der Ringe“- Saga (engl.).