Dr. Johannes Mayer
Axel Blumenstock
Christian Ehrhardt

Abteilung Angewandte Informationsverarbeitung

23. November 2005
Blatt 5

Allgemeine Informatik III / Systemnahe Software I (WS 2005/2006)

Abgabetermin: 29. November 2005

Fasse Dich kurz… (10 Punkte)

Die zentrale Idee bei Verfahren der verlustfreien Datenkompression besteht darin, häufig vorkommende Zeichen (allgemeiner: Sequenzen) durch kurze Bitfolgen zu repräsentieren, und seltene durch lange. Sind die Zeichen in der Ursprungsdatei tatsächlich sehr ungleich verteilt, ergibt sich so in der Summe eine Reduktion des Speicheraufwandes.
Wir möchten ein Verfahren implementieren, das

- einerseits nicht vorab darauf festgelegt ist, welche Zeichen wie oft vorkommen, sondern sich an beliebige Verteilungen anpassen kann;
- andererseits soll es die zu komprimierenden Daten nicht komplett im Speicher halten oder zweimal lesen müssen, sondern gleich beim ersten Lesen – „online" – komprimieren.

Unser (einfacher) Ansatz soll jedes Zeichen aus der Eingabedatei in drei Schritten verarbeiten:

- Zunächst werden die Zeichen in eine Folge von Zahlencodes zwischen 0 und 255 übersetzt, so dass darin kleine Zahlen möglichst häufig sind.
- Dann werden kleine Codes auf kurze Bitsequenzen, große in längere Bitsequenzen abgebildet.
- Diese Bitsequenzen werden byteweise in die Ausgabe geschrieben.

Der erste Schritt ist auch als *Move-To-Front-Codierung* (MTF) bekannt und Teil des sehr leistungsfähigen *bzip2*. Wir initialisieren ein Array *a* der Größe 256 so, dass *a[i] = i*. Müssen wir nun ein bestimmtes Zeichen *x* codieren, geben wir aus, an welcher Stelle es in diesem Array steht (mit anderen Worten: den Index *i*, so dass *a[i] = x*). Gleichzeitig holen wir es an die Position 0 des Arrays und verschieben alle Zeichen zwischen Position 0 und *i* um eins nach hinten.

Zu Illustrationszwecken verwenden wir hier nur ein Array der Größe zwölf, das mit den Zeichen A (an Index 0) bis L (Index 11) gefüllt ist.

A	B	C	D	E	F	G	H	I	J	K	L

Codieren wir damit das Wort „ALLE": Das Zeichen 'A' steht an Position 0, also geben wir 0 aus. Da es bereits vorne im Array steht, müssen wir nichts weiter tun. Das nächste Zeichen 'L' steht an Position 11, also geben wir 11 aus. Holen wir es nach vorne, sieht das Array wie folgt aus:

L	A	B	C	D	E	F	G	H	I	J	K

Es folgt ein weiteres 'L', seine Position und damit unsere Ausgabe ist jetzt 0. Auch in diesem Schritt ändert sich das Array nicht. Für das 'E' müssen wir nun die Position 5 ausgeben, und es ergibt sich folgende Situation:

E	L	A	B	C	D	F	G	H	I	J	K

Wir haben also die Zeichenfolge „ALLE" zur Codefolge 0, 11, 0, 5 übersetzt. Es ist klar, dass häufig vorkommende Zeichen im Laufe der Zeit vorne im Array stehen werden und daher kleine Codes erhalten.
Im zweiten Schritt müssen wir nun die Zahlencodes in Bitsequenzen umwandeln, so dass kleine Codes kurze Sequenzen erhalten. Gleichzeitig muss gewährleistet sein, den Bitstrom beim Decodieren wieder in die einzelnen Bitsequenzen zerlegen zu können. Dies gelingt mit folgendem *präfixfreien* Code:

- Vor die Binärdarstellung der Codes 0 bis 7 (binär `000` bis `111`) setzen wir einfach eine 0. Das heißt, 0 wird als die Bitsequenz `0000` repräsentiert, 1 als `0001`, bis zur 7 als `0111`.
- Von den Codes 8 bis 39 ziehen wir zunächst 8 ab und hängen die fünfstellige Binärdarstellung des Ergebnisses (`00000` bis `11111`) an eine `10` an.
- Von allen übrigen Codes ziehen wir 40 ab und hängen das Ergebnis als achtstellige Binärzahl an eine `11` an.

Folgende Tabelle zeigt ein paar Beispiele hierzu (Leerzeichen nur zwecks Lesbarkeit):

Zahlencode	Bitsequenz
0	`0 000`
1	`0 001`
7	`0 111`
8	`10 00000`
11	`10 00011`
32	`10 11000`
38	`10 11110`
39	`10 11111`
40	`11 00000000`
41	`11 00000001`
255	`11 11010111`

Unsere Zahlenfolge 0, 11, 0, 5 würde also so aussehen: `0000 1000011 0000 0101`.
Der dritte Schritt besteht nur mehr darin, diese Bitsequenz byteweise in die Ausgabe zu schreiben. Wir füllen dazu die Bytes von links nach rechts. Sofern das letzte Byte nicht komplett mit gültigen Bits belegt ist, wird es mit Einsen aufgefüllt, was auf keinen Fall mehr als gültige Bitsequenz interpretiert werden kann. Unsere obige Bitsequenz ergibt also die drei folgenden Bytes, wobei fünf Einsen angefügt werden mussten:

```
00001000
01100000
10111111
```

Dieses Verfahren ist umkehrbar. Sehen wir im Bitstrom zunächst eine `0`, wissen wir, dass die nächsten drei Bits als Zahlencode zwischen 0 und 7 interpretiert werden müssen. Sehen wir eine `10` bzw. eine `11`, so gehören die nächsten fünf bzw. acht Bits zusammen. Und initialisieren und modifizieren wir unser Array wie beim Komprimieren, lässt sich aus der Zahlenfolge 0, 11, 0, 5 auch wieder der ursprüngliche Text „ALLE“ gewinnen.
Die Aufgabe: Schreiben Sie zwei Programme – *mtf-encode* und *mtf-decode* – die von der Standardeingabe lesen, auf die Standardausgabe schreiben und dabei nach oben beschriebenem Verfahren komprimieren bzw. wieder dekomprimieren. Wenn Ihre Programme funktionieren, brauchen Sie Ihrem Tutor die nachfolgend beschriebenen Zwischenstufen nicht vorzuführen, sie dienen lediglich als Anleitung.

(a) Legen Sie sich als globale Variable ein int-Array der Größe 256 an und initialisieren Sie es. Schreiben Sie eine Funktion

```
int getIndex(int character)
```

welche das gegebene Zeichen *character* in dem Array sucht und seinen Index zurückgibt, nachdem sie es im Array nach vorn geholt hat.

(b) Schreiben Sie eine kurze *main()*, die Zeichen von der Standardeingabe entgegennimmt und die Zahlencodes mittels `printf("%d ", ...)` an der Standardausgabe ausgibt. Im ftp-Verzeichnis zu diesem Blatt finden Sie Beispieltexte und die zugehörigen, auf diese Weise zahlencodierten Dateien (`*.num`).

(c) Erstellen Sie nun bereits die erste Fassung von *mtf-decode*, das solche Zahlensequenzen mit `scanf("%d", ...)` von der Standardeingabe liest und dies mit demselben Array wie oben zu dem Klartext decodiert.

(d) Ergänzen Sie Ihr Programm *mtf-encode* um eine Funktion wie

```
void writeBits(int code)
```

welche für den gegebenen Zahlencode die zugehörige Bitsequenz erzeugt (siehe obige Tabelle) und in Bytes verpackt. Dazu können Sie beispielsweise eine globale Puffervariable vom Typ int verwenden, in die Sie die Bitsequenzen hineinschieben; und sobald mindestens acht Bits enthalten sind, schreiben Sie diese als Zeichen in die Standardausgabe. Sehen Sie ferner eine Funktion

```
void flushBuffer()
```

vor, die zum Schluss aufzurufen ist und, falls noch Bits im Puffer verblieben sind, ihn mit Einsen auffüllt und als letztes Zeichen ausgibt.

Sie können diese Funktionen testen, indem Sie mit ihnen die Codes 0 bis 255 in Bitsequenzen abbilden und das Ergebnis mit der Datei `0-255.mtf` aus dem ftp-Verzeichnis vergleichen.

(e) Schreiben Sie im Dekomprimierprogramm eine Funktion, die aus dieser Datei wieder die Codes 0 bis 255 extrahieren kann. Hilfreich ist dabei eine Funktion wie

```
int getBits(int howMany)
```

die (ebenfalls mittels einer Puffervariable) die nächsten *howMany* Bits aus der Standardeingabe holt.

(f) Haben Sie auch das erfolgreich bewerkstelligt, müssen Sie die Funktionen nur noch zusammensetzen.

Bemerkung: Dass unsere Implementierung keine sonderlich beeindruckenden Kompressionsraten erreicht, hat zwei Gründe:

- Zum einen müssen für jedes Zeichen mindestens vier Bit ausgegeben werden. Kompressionsraten von unter 50 % sind hier also gar nicht möglich. Es gibt Verfahren, die für eine gegebene Verteilung der Zeichen eine optimale Abbildung auf Bitsequenzen ermitteln, die unser einfaches dreistufiges Schema (0xxx, 10xxxxx, 11xxxxxxxx) ersetzen würde.

- Unser Verfahren betrachtet jedes Zeichen für sich. Dadurch gleicht das Decodieren einem Zufallsexperiment, bei dem wir für die Vorhersage des nächsten Zeichens nur die Häufigkeitsverteilung der einzelnen Zeichen zur Verfügung haben, wodurch die Unsicherheit (und damit der Informationsbedarf in bit) sehr groß ist. Würden wir hingegen den Kontext berücksichtigen (beispielsweise: welches Zeichen folgt auf „berücksic_“), wäre der Informationsbedarf wesentlich geringer.

Viel Erfolg!