Rheinisch Westfälische Technische Hochschule
Institut für Geometrie und Praktische Mathematik

# Mathematisches Praktikum (MaPra) — SS 2001

Prof. Dr. Wolfgang Dahmen — Alexander Voß — Stephan Niewersch

## Aufgabe 7

**Bearbeitungszeit:** bis xxx
**Mathematischer Hintergrund:** Informationstheorie, LZW Komprimierung
**Elemente von C++:** Maps

---

### Aufgabenstellung

Vervollständigen Sie ein Programm, mit dessen Hilfe Sie Kanten in einem *GIF-Bild* detektieren können. Implementieren Sie dazu in den hierfür vorgesehenen Routinen den *LZW*-[1]Komprimierungsalgorithmus zum Laden und Speichern des Bildes und den Algorithmus zur Kantenerkennung.

### Mathematischer Hintergrund: LZW-Komprimierung

Die LZW-Komprimierung kann zunächst genauso wie der Huffman-Code der letzten Übung zur Komprimierung von Daten aller Art verwendet werden. In dem bekannten *GIF-Bildformat* wird jedoch genau diese Kompressionsart benutzt und es liegt nahe, direkt Routinen zum Laden und Speichern von GIF-Bildern zu implementieren.
Der LZW-Algorithmus wurde 1977 zum ersten Mal veröffentlicht und die grundliegende Idee ist, kurz gesagt, sich wiederholende Zeichenfolgen durch kurze Codes zu ersetzen. Zu diesem Zweck werden Tabellen angelegt, die Buch darüber führen, welche längeren Strings (Zeichenfolgen) mit welchen Codes ersetzt werden. Im Gegensatz zum Huffman-Code braucht man diese Tabellen jedoch nicht mit abzulegen, sie können aus dem komprimierten Code wieder erzeugt werden.

Schema zur LZW-Komprimierung:

```
S := hole naechsten char aus unkomprimierten Daten
WHILE (es gibt noch chars in unkomprimierten Daten) DO
  C := hole naechsten char aus unkomprimierten Daten
  IF (S+C ist in der Code-Tabelle) THEN
    S := S + C
  ELSE
    gib den Code fuer S aus
    fuege S+C in die Code-Tabelle ein
    S := C
  END IF
END WHILE
gebe den Code fuer S aus
```

[1] Lempel-Ziv-Welch

**Mathematischer Hintergrund: Komprimierung, Entropie**

Zwar ist Festplattenplatz heutzutage einigermaßen billig, dennoch fallen beispielsweise bei der Bildverarbeitung oder in Datenbanken leicht Datenmengen an, die auch große Platten schnell füllen: Ein Farb-Bild im Format 1024×768 Punkte mit 24 bit Farbtiefe (d. h. 256 Helligkeitsstufen pro Grundfarbe) beispielsweise verschlingt unkomprimiert 2.25 MB Speicher. Daher möchte man Daten komprimieren, um sie möglichst platzsparend abzulegen. Auch bei der (manchmal recht langsamen) Übertragung von Daten über Netze möchte man nicht Zeit(=Geld) für unnötig lange Dateien verschwenden.
Grundsätzlich sind dabei zwei Kategorien der Kompression zu unterscheiden: *verlustfreie* und *verlustbehaftete Komprimierung.* Bei der ersten Variante kann man die ursprünglichen Daten wieder genau rekonstruieren (ausführbare Programme sind z.B. recht empfindlich gegen kleine Änderungen). Prinzipiell lassen sich diese Verfahren auf beliebige Daten anwenden, eine Komprimierung erzielt man jedoch meist nur, wenn die Daten gewisse statistische Annahmen erfüllen, wie zum Beispiel: einige (Grau-)werte bzw. Zeichen in Texten kommen häufiger vor als andere (oder benachbarte Daten weichen nur wenig voneinander ab). Mit dieser Kategorie befaßt sich die vorliegende Aufgabe.
Bei der zweiten Variante lassen sich Daten nur näherungsweise rekonstruieren, was z.B. speziell bei Bildern sinnvoll sein kann. Je geringer die Anforderungen an die Treue zum Original dabei sind, desto größer kann offensichtlich die Kompressionsrate werden. Typische Vertreter dieser Kategorie sind das JPEG-Format und die im neuen JPEG enthaltene Wavelet-Kompression. Hier müssen Annahmen darüber getroffen werden, welche Teile eines Bildes verändert werden können, ohne daß das Bild beeinträchtigt wird. Meist sind es die hochfrequenten Anteile eines Bildes, die nur ungenügend wiedergegeben werden. Ob man bei einem Bild wirklich auf diese Information verzichten kann, ist allerdings vom Kontext (d. h. der späteren Interpretation der Daten) abhängig: Bei einem eingescannten Foto eines Autos (mit großen glatten Flächen) werden einzelne Bildpunkte, die sich stark von ihren Nachbarn unterscheiden, hauptsächlich von Bildfehlern herrühren, die man getrost wegretuschieren kann — bei einem Foto des Nachthimmels würde man mit derselben Strategie allerdings sämtliche Sterne aus dem Bild entfernen.
Die Möglichkeit zur Kompression von Daten ist wie schon angedeutet untrennbar mit ihrem Inhalt verbunden. Es stellt sich die Frage nach einem geeigneten Maß für Komprimierbarkeit, oder wie wir noch sehen werden, für Information bzw. Ungewißheit.
Beispiel: Man betrachte ein Experiment, welches als Ergebnis entweder A oder B liefert. Dann würde ein Bit ausreichen um das Ergebnis zu beschreiben. Allgemein würden

$$\log_2(M) = -\log_2(P), \quad \text{mit} \quad P = \frac{1}{M}$$

Bits ausreichen, wenn man $M$ Symbole hätte, die alle mit der gleichen Wahrscheinlichkeit $P$ auftreten würden.
Erweitert man die Situation auf den Fall, daß verschiedene Symbole $S_i$ verschiedene Wahrscheinlichkeiten $P_i$ haben, so definiert

$$u_i = -\log_2(P_i)$$

den *Überraschungswert* ("surprisal" nach [T]), ein solches Symbol zu bekommen[2]. Wir nehmen dabei immer an, daß $\sum P_i = 1$ gilt.

Anders ausgedrückt: in einer Reihe von $N$ Ergebnissen eines Experiments, bei dem $N_i$ die Häufigkeit des $i$'ten Symbols beschreibt, ist

$$\sum_{i=1}^{M} \frac{N_i}{N} u_i$$

der mittlere Überraschungswert für die $N$ Symbole. Mit dem Gesetz großer Zahlen konvergiert bei einem Experiment mit unendlich vielen Ergebnissen $\frac{N_i}{N}$ gegen die Wahrscheinlichkeit $P_i$ und der *mittlere*

[2]Ist ein $P_i = 0$, so wäre man *ziemlich* überrascht dieses Symbol $S_i$ zu sehen, wo es doch gar nicht auftreten kann. Umgekehrt ist die Überraschung gleich Null, wenn man das Ergebnis mit Sicherheit, also $P_i = 1$, kennt.

*Überraschungswert* ist

$$H = \sum_{i=1}^{M} P_i u_i = -\sum_{i=1}^{M} P_i \log_2(P_i) = \sum_{i=1}^{M} P_i \log_2\left(\frac{1}{P_i}\right),$$

genannt *die Entropie*.
Für die Entropie gilt:

$$H \geq 0 \quad \text{und} \quad H = 0 \quad \Longleftrightarrow \quad P_i = 1 \quad \text{für ein } i,$$

$$H \leq \log_2(M) \quad \text{und} \quad H = \log_2(M) \quad \Longleftrightarrow \quad P_i = \frac{1}{M} \quad \text{für alle } i.$$

Die Entropie ist also ein Maß für den Informationsgehalt der Daten. Sie gibt Auskunft darüber, wie *komprimierbar* Daten sind, denn sind sie z.B. gleichermaßig verteilt, so muß man schon $\log(M)$ Bits spendieren, anderenfalls kommt man vielleicht mit weniger aus.
Beispiel: Eine Textdatei enthält:

$$ABACDAAB \quad .$$

Je vorkommendem Symbol ergeben sich die Häufigkeiten: $N_A = 4$, $N_B = 2$, $N_C = 1$, $N_D = 1$ mit $N = 8$. Ordnet man den Symbolen die Wahrscheinlichkeiten $\frac{N_i}{N}$

$$P_A = \frac{4}{8} = \frac{1}{2}, \quad P_B = \frac{2}{8} = \frac{1}{4}, \quad P_C = \frac{1}{8}, \quad P_D = \frac{1}{8}$$

zu, so ergibt sich

$$u_A = 1 \quad \text{Bit}, \quad u_B = 2 \quad \text{Bits}, \quad , u_C = 3 \quad \text{Bits}, \quad u_D = 3 \quad \text{Bits}$$

und als Entropie

$$H = \frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + \frac{1}{8} * 3 = 1.75,$$

d.h. die mittlere Anzahl von Bits, die zur Kodierung nötig sind ist 1.75.
Angenommen, man ordnet den Symbolen die folgenden Bitkodierungen zu:

$$A = 1, \quad B = 01, \quad C = 001 \quad \text{und} \quad D = 000,$$

so würde der obige Text kodiert durch die Bitfolge

$$10110010001101.$$

Das sind genau 14 Bits für 8 Symbole, also im Mittel 1.75 je Symbol.

Es stellt sich nun die Frage: existiert zu gegebenen Daten eine (praktikable) Kodierung, so daß die mittlere Länge pro Symbol in $[H, H + 1)$ liegt?
Die Antwort ist *ja* und der *Huffman-Algorithmus* erzeugt gerade eine solche optimale Kodierung.

**Huffman-Code**

Der *Huffman-Code* basiert darauf, für einzelne Symbole die häufig vorkommen kurze Bitfolgen und für seltenere längere Bitfolgen zu verwenden, siehe auch das obige Beispiel. Bei der Auswahl der Codes muß allerdings darauf geachtet werden, daß die Bitfolgen „präfixfrei“ sind: Angenommen, man hätte für die drei Werte A,B,C Codes „1“, „0“ und „10“. Dann könnte man nicht entscheiden, ob mit der Bitfolge „10“ die Symbole A und B oder nur das Symbol C gemeint ist. Es darf daher nicht vorkommen, daß ein Anfangsteil (Präfix) eines Codes — hier die „1“ von „10“ für C — bereits einen anderen Code darstellt. Genau dies leistet der Huffman-Code, von dem sich sogar nachweisen läßt, daß er derjenige präfixfreie Code ist, der die höchste Kompresionsrate liefert.
Im Folgenden betrachten wir Daten, also die Symbole, als Bytes oder Chars (ohne Vorzeichen), d.h.

aus dem Wertebereich 0 bis 255. Genausogut kann man sie als Grauwerte interpretieren, was die folgenden Diagramme lesbarer macht. Diese Interpretation ändert an der Kodierung nichts.
Um den Huffman-Code zu bestimmen, benötigt man zunächst die Häufigkeitsverteilung der Symbole (Grauwerte), das sogenannte *Histogramm*. Aus ihm wird von den Blättern zur Wurzel ein Baum konstruiert, bei dem jedes Blatt einem Symbol entspricht: Dazu ordnet man dem Histogramm der $n$ Symbole zunächst einen Wald von $n$ Bäumen zu. Jeder Baum besteht dabei nur aus einem einzigen Knoten, der das Symbol und seine Häufigkeit enthält. Nun sucht man sich die zwei Bäume mit der kleinsten Häufigkeit (jeweils an der Wurzel) heraus und vereint sie durch Anhängen eines gemeinsamen Vaterknotens an den Wurzeln zu einem neuen Baum. Der Vaterknoten (die Wurzel des neuen Baums) bekommt als Häufigkeit die Summe der beiden Häufigkeiten der Kinder zugewiesen und als (Pseudo)symbol das kleinere der beiden Kinder[3]. Die entstehenden Kanten bekommen die Markierung „0“ und „1“. Aus den nun $n-1$ Bäumen sucht man nun wieder diejenigen mit den kleinsten Häufigkeiten an der Wurzel etc. Die folgende Skizze verdeutlicht diesen Prozeß (der Grauwert sei das Symbol):

Zum Schluß erhält man einen einzigen Baum, aus dem man nun den Huffman-Code eines Symbols ablesen kann: Jedes Symbol läßt sich durch einen eindeutigen Weg von der Wurzel aus erreichen. Die Markierungen der Kanten, die man dabei durchläuft, ergeben dann den Code.
Wenn man diesen Algorithmus genau betrachtet, erkennt man, daß der Code nicht eindeutig bestimmt ist: Erstens spielt es keine Rolle, welche der beiden Kanten die Aufschrift „0“ und welche die Aufschrift „1“ bekommt, wenn man zwei Teilbäume vereinigt. Zweitens sind die beiden Bäume mit der niedrigsten Häufigkeit nicht unbedingt eindeutig bestimmt. Es hängt also vom Sortieralgorithmus und der Wahl der Kantenmarkierung ab, wie der Code aussieht. Auf die Länge des komprimierten Bildes hat dies aber keinerlei Einfluß. Man sollte aber aufpassen, daß beim Komprimieren und Dekomprimieren derselbe Code verwendet wird!

**Bemerkungen zum Algorithmus**

- Für Symbole, die im Bild gar nicht vorkommen, sollte kein Code erzeugt werden. Ansonsten würde der Algorithmus nicht nur langsamer; unter Umständen würden die anderen Codes unnötig lang (um die Präfixfreiheit zu wahren).

- Damit der Code eindeutig wird, auch wenn es mehrere Bäume mit der gleichen Häufigkeit an der Wurzel gibt, führen wir ein zweites Sortierkriterium ein: Sind die Häufigkeiten an der Wurzel gleich, so ist derjenige Baum der „kleinste“, dessen Wurzel den niedrigsten Symbolwert trägt. Beim Vereinen zweier Bäume bekommt der neu hinzukommende Vaterknoten als Symbol das Minimum der Symbole seiner beiden Söhne zugewiesen. Ferner bekommt diejenige Kante die Aufschrift „0“ die zum „kleineren“ Knoten (gemäß Häufigkeit bzw. Symbol) führt.

- Es brauchen nicht immer alle Bäume korrekt sortiert zu werden, es reicht, in jedem Schritt die beiden „kleinsten“ zu lokalisieren.

**Kodieren und Dekodieren von Daten**

Wenn man den Code bestimmt hat, kann man jedes Symbol durch seinen Bitcode ersetzen. Sollte die Länge der Bitfolge, die so entsteht, nicht durch 8 teilbar sein, so füllt man noch Nullbits auf, bis eine durch 8 teilbare Länge erreicht ist. Jeweils 8 Bit kann man dann zu einem Byte oder Char zusammenfassen.
Auch das Dekodieren läßt sich am einfachsten mit Hilfe des Baums durchführen: Man startet bei der Wurzel und wählt je nach gelesenem Bit die Kante mit der entsprechenden Markierung, bis man schließlich ein Blatt, d. h. ein Symbol, erreicht. Danach startet man wieder bei der Wurzel, um den nächsten Grauwert zu bestimmen. Da man den Baum wieder zum Dekodieren benötigt, speichern wir einfach das Histogramm anstelle des Baumes ab, aus ihm läßt sich der Baum wieder rekonstruieren[4]

[3] Dieses Pseudosymbol wird nur für die Sortierung benutzt, siehe später.
[4] siehe auch: Hinweise zur Implementation.

**Verbesserung der Kompressionsrate**

Je ungleichmäßiger das Histogramm ist, desto höher ist die Kompressionsrate bei der Huffman-Kodierung, weil sich dann der Längenunterschied der einzelnen Codewörter umso stärker bemerkbar macht. Um die Kompressionsrate zu verbessern, kann man versuchen, durch eine invertierbare Transformation das Histogramm zu beeinflussen, z.B. durch Abspeichern von Mittelwerten von Symbolen und Differenzen. Dies soll jedoch nicht mehr Gegenstand dieser Übung sein.

**Aufgabe**

i) Schreiben Sie ein Programm, z.B. `a6`, welches bei Aufruf von der Kommandozeile

```
a6 -c file1 file2
```

die Datei `file1` komprimiert und als Datei `file2` wieder abspeichert

ii) und durch Aufruf

```
a6 -d file2 file3
```

die Datei `file2` dekomprimiert und als Datei `file3` abspeichert.

iii) Benutzen Sie zur Komprimierung den Huffman-Code.

iv) Berechnen Sie beim Komprimieren und Dekomprimieren die Entropie und den Kompressionsfaktor (Verhältnis zwischen komprimierten und unkomprimierten Daten) und geben Sie diese mit aus.

v) Zum Testen existieren Beispieldateien. Wenn Ihr Programm diese komprimieren kann und das Ergebnis wieder dekomprimiert die Originaldatei ergibt, so ist die Aufgabe gelöst. Alternativ können mit dem Programm `loesung6` auch Referenzlösungen erzeugt werden.

**Hinweise zur Implementation**

In `unit6.h` sind zwei Hilfsklassen beschrieben: `charvector` und `bitvector`. Diese sind von `std::vector` abgeleitet und mit einigen Hilfsfunktionen versehen, die Sie benutzen können oder auch nicht, compiliert in `unit6.o`. Eine kurze Beschreibung der Hilfsfunktionen finden Sie im Header-File `unit6.h`.

Eine Datei kann als ein Vektor von chars (die Symbole) betrachtet werden. Dabei ist ein char ein Byte, und besteht aus 8 Bits.

Der Datentyp für einen Knoten eines Baumes könnte wie folgt aussehen:

```
struct Knoten
{
  unsigned char Symbol;
  unsigned int  Haeufigkeit;
  Knoten *P0, *P1;
};
```

Bedenken Sie jedoch, daß für die Knoten für die Symbole die jeweiligen Codes noch abgelegt werden müssen.

Wenn Sie einen Bitvektor als einen Vektor aus chars interpretieren (also immer 8 Bits zusammenfassen), so sind die letzten Bits mit Null aufzufüllen. Möchte man umgekehrt den Bitvektor aus dem Charactervektor erzeugen, so ist nicht klar, wieviele Bits des letzten chars dazugehören. Spendiert man als ersten char die Anzahl von Bits im letzten char (0 für keine), so ist der Bitvektor wieder eindeutig zu rekonstruieren.

Der Aufbau der komprimierten Referenzdateien ist wie folgt:

| Position | Länge in Bytes | Inhalt |
|---|---|---|
| 0–3 | 4 | ID-String "HUFF" |
| 4–7 | 4 | Anzahl M verschiedener chars (Symbole) |
| 8–8+5*M-1 | 5*M | je Symbol 1 Byte Symbol + 4 Bytes Häufigkeit |
| ... | ... | Bitvektor |

Ein Wert vom Typ `unsigned int` besteht in der Inter/AMD+Windows/Linux Welt in der Regel aus 32 Bit, d.h. aus 4 Bytes bzw. chars. Somit kann man also eine Zahl obigen Typs in einem Vektor aus chars ablegen und auch wieder auslesen.

**Literatur**

[T] M. Tribus. Thermostatics and Thermodynamics. D. van Nostrand Company, Inc., Princeton, N. J., 1961

http://www.math.psu.edu/gunesch/Entropy/infcode.html, Linkliste vieler interessanter Artikel, u.a. C. E. Shannon. A Mathematical Theory of Communication.