Informatik für Physiker - Zusammenfassung

Vorlesung: Prof. Dr. Süße
Zusammenfassung: Fabian Stutzki

19. Juli 2006

Die Zusammenfassung bezieht sich auf Informatik für Physiker (SS06).

Inhaltsverzeichnis

1 Programmiersprachen

- herkömmliche: Assembler, Pascal, C, Fortran
- objektorientierte: Simula, C++, Javam C#
- funktionale: Lisp, Coter
- relationale / regelorientierte: Prolog (Ki-Programmierung)
- algebraisch: Maple, Mathcat
- spezielle Fachsprachen: SQL, PHP ...

Syntax

- Backus-Nauer-Form (BNF)
- Syntaxdiagramme (N. Wirth (Pascal))

2 Kodierungen

- Datensicherheit (Fehlerkkorrektur, Geheimhaltung, ...)
- Andere Informationsrepräsentation

3 Datenkompression

Entropie eines Zustandes: (Maßeinheit: bit, Wahrscheinlichkeit p_i des i-ten Zustandes):

$$n = H = \log_2 m = -\log_2 \frac{1}{m} = -\log_2 p_i$$

$m = 2^n$ Zustände, n binäre Variabeln

mittlere Entropie des Systems: Grenze für Kompression von Daten

$$H_0 = -\sum_{i=1}^{m} p_i \log_2 p_i$$

- maximal für $p_i = \frac{1}{m}$ (gleichverteilt) $\Rightarrow H_0 = \log_2 m$
- minimal für Null-Eins-Verteilung $p_j = 1$ und $p_i = 0\ \forall i \neq j \Rightarrow H_0 = -p_j \log_2 p_j = 0$

$$0 \leqslant H_0 \leqslant \log_2 m$$

einfarbiges Bild, maximale Sicherheit $\leftrightarrow$ jedes Bild der Welt, maximale Unsicherheit

3.1 ohne Informationsverlust

3.1.1 Run length encoding

3.1.2 Flächenkodierung (Quadtress)

3.1.3 statische Kodierung

gegebenes Quellenalphabet $A = \{a_1, a_2, \ldots a_n\}$ und Kodealphabet $B = \{b_1, \ldots b_k\}$, zu kodierende Symbolkette besteht aus Buchstaben $a_1 a_3 a_{10} a_4 \ldots$

I-Code: Code mit variabler Kodewortlänge, Unterscheidung zwischen präfixfrei $(1, 01, 001, \ldots)$ und präfixbehaftet $(1, 10, 100, \ldots)$

Ungleichung von Kraft: Für eindeutig decodierbaren Code gilt (d_i Länge des i-ten Codewortes):

$$k^{-d_1} + k^{-d_2} + \ldots + k^{-d_n} \leqslant 1$$

(wenn Gleichung nicht erfüllt, dann Code nicht eindeutig)
Bsp: ACSII $k = 2$, $n = 256$, $d_i = 8 \Rightarrow 2^{-8} \cdot 256 \leqslant 1$
Mittlere Länge L aller Kodewörter

$$L = \sum p_i d_i$$

Huffman-Code: I-Code mit kürzester mittleren Wortlänge L_{min}

$$S^k \to L_{min}/k$$

Entropie untere Schranke

$$H(S) = -\sum_i p_i \log_2 p_i$$

, Huffman-Code gut bei hoher Entropie

Shannon-Noiseless-Theorem: für beliebige Informationsquelle (nur für Statistiken höherer Ordnung zu unterschreiten)

$$H(S^k) \leqslant \frac{L_{min}(S^k}{k}) \leqslant H(S) + \frac{1}{k}$$

für gedächnisfreie Informationsquelle gilt:

$$P(a_{i1} \dots a_{in}) = P(a_{i1} \dots P(a_{in}$$

$$H(S^k) = kH(S)$$

Shannon-Fano-Code: fast Huffman-Code, aber einfacher zu konstruieren

3.1.4 Wörterbuchverfahren

Aufbau während Übertragung

3.1.5 Arithmetische Kodierungen

Symbolkette als ganzes ist das Codewort, gute Kompression für kleine Entropie

3.2 mit Informationsverlust

z.B. JPEG und MPEG

- Transfrmation: bei JPEG früher Discrete Cosinus Trafo, bei JPEG heute Wavlet Trafo
- Vektorquantisierung: Clusteranalyse (Blöcke verschieben)
- Fraktale Methoden
- Approximation

4 Kryptographie

mit geheimem Schlüssel: u Message, k Key, $\oplus$ XOR $\Rightarrow$ Kodierung $v = u \oplus k$ und Dekodierung: $v \oplus k = v \oplus k \oplus k = u$

mit öffentlichem Schlüssel: Einwegfunktion $f(u) = u'$ mit schwieriger Invertierung, z.B. Passwortverwaltung (mit Zufallsergänzung oder salt), RSA-Algortihmus (um private Key ergänzt, elektronische Unterschrift (mit private Key kodieren)

5 Rekursivität

Grundkonzept: Teile und hersche

```
Modul A(D)
IF A(D) == triviales Problem
THEN triviale Loesung explizit angeben
ELSE
teile
herrsche
```

6 Bewertung von Algorithmen

asymptotisches Verhalten entscheidend, f Vergleichsfunktion, g Alorithmus

- $g \in O(f)$: g hat höchsten die Zeitkomplexität f
- $g \in \Omega(f)$: g mindestens von der Zeitkomplexität f
- $g \in \Theta(f)$: g genau von der Zeitkomplexität f

übliche Vergleichsfunktionen

Funktion	Laufzeit	Beispiel
1	konstant	Array-Zugriff
$\log n$	logarithmisch	
n	linear	Skalarprodukt, Distribution Sort,
$n \log n$	fast linear	Quick Sort, Heap Sort, Merge Sort
n^2	quadratisch	Bubble Sort (best case n), Insert Sort
n^k	polynominal	n^3 Matrixmultiplikation
a^n	exponentiel	

Bsp: Selection Sort beim ersten Durchlauf n Operationen, beim zweiten $n-1$ Operationen $\Rightarrow g(n) = \sum^n i = \frac{n(n+1)}{2} \in O(n^2)$

7 Sortieralgorithmen

Distribution Sort: auch Counting Sort oder Histogrammmethode, sehr schnell, benötigt viel Speicher $\Rightarrow$ nur bei kleinem Zustandsraum sinnvoll
Bsp: Bild mit 256 Grauwerten, alle Bildpunkte in Histogramm eintragen, im Histogramm bis zur halben Anzahl durchzählen, auslesen, fertig

Selection Sort: Maximum aus einer Zeichenkette heraussuchen, nach hinten stellen, Zeichenkette um eins verkürzen

$$\dot{Z}ADBE \rightarrow ADB\dot{E}\underline{Z} \rightarrow A\dot{D}B\underline{EZ} \rightarrow A\dot{B}\underline{DEZ} \rightarrow \dots$$

Insert Sort: nächstes Zeichen in sortierten Teil der Zeichenkette einfügen:

$$\underline{Z}\dot{A}DBE \rightarrow \underline{AZ}\dot{D}BE \rightarrow \underline{ADZ}\dot{B}E \rightarrow \dots$$

Bubble Sort: Benachbarte Elemente vertauschen, String mehrmals durchgehen:

$$\underbrace{ZA}DBE \rightarrow A\underbrace{ZD}BE \rightarrow AD\underbrace{ZB}E \rightarrow \dots$$

Quick Sort: Rekursiver Algorithmus (im Durchschnitt $O(n \cdot \log n)$, worst case $O(n^2)$ wird mit Introsort verbessert): Zufällig ein Element heraussuchen, Liste in eine Hälfte mit kleineren und eine mit größeren Elemenenten aufteilen (von links zu großes und von rechts zu kleines Element finden und vertauschen, weiter bis beide Zeiger zusammentreffen)

Merge Sort: Rekursiver Algorithmus: Sortiere linke und dann rechte Listenhälfte, anschließend mische (= sortiere vorsortierte Listenhälften) zur gesamten Liste

Tree Sort: Baum nach Schema aufbauen (links kleineres, rechts größeres Element), Ausgabe: linker Teilbaum, Knoten und rechter Teilbaum

Heap Sort: Vollständiger Baum mit Heap-Eigenschaft (z.B. Nachfolger $\leqslant$ Vorgänger)

8 C

8.1 Bit-Verarbeitung

`unsigned int` oder `unsigned char`
Operatoren

- Verschiebung: shift `<<` und `>>`
- unäre Negation: $\sim$
- bitweise and `&`, or |, xor `^`

8.2 Felder

```
double[100]={0.0}
```

8.2.1 Zeichenketten

```
material[6]={'s','t','e','e','l','\0'};
material[]={"steel"}; // letztes Zeichen immer eine Null
```

letztes Zeichen immer Null

8.3 Pointer

typisierte Adressen (unsigned int)

```
int *p;
int y,x=20;
int p = &y; // Pointer p zeigt auf Adresse von y
int *p = x; // x wird auf den Inhalt von p gespeichert, y = 20
```

Pointerarithmetik: Rechenoperationen wie Addition, Subtraktion oder Multiplikation mit Pointern

```
p++; // eine Adresseinheit des Typs weiter
p--;
y=*p++ + 4; // +4 dann p eine Adresse weiter -> postfix
y = *(++p) + 4; // erst weiter dann 4 addieren -> prefix
```

Pointer als Funktionsparameter: call by reference zu erreichen, ormaler Funktionsaufruf mit **call by value** (Wertübergabe, Kopie wird erzeugt)

```
void double_it(float *ptr)
{
*ptr = 2.0
}

double_it(&x); // Funktionsaufruf
```

Funktionspointer: Pointer auf Funktionen

```
double multi(double x, double y) { return x*y; } // Funktion
double (*f)(double, double); // Funktionspointer
f = multi; // Funktionspointer zeigt auf Funktion multi
```

8.4 Prototypen

Vor 1. Aufruf mit `double sin(double);` initialisieren, default `extern`, aber auch `static` oder `inline` möglich

8.5 Dynamische Speicherplatzverwaltung

Memory Allocation durch malloc, Freigabe durch free

```
int *feld; int n=10;
feld = (int*)malloc(n*sizeof(int));
...
free(feld);
```

8.6 Datentyp Struktur

```
struct point{ float x,y,z, int val;}
point a;
a.y = 10;
a->x = 10:
struct node { int val; struct node* next; }
```

8.7 typedef-Aneisung

```
typedef char* string;
```

oder

```
typedef struct node *Ref; // Ref Synonym für node*
typedef struct node
{ int val; Ref next} NODE; // NODE Synonym für node
```

8.8 Datenorganisation

sequentiell: Sätze bilden lineare Liste ($O(n)$)

binäre Suche: sortierte Liste ($O(\log n)$)

direkt adressierbar: Feld von Sätzen (feste Blöcke), key $\leftrightarrow$ id, nach Hauptordnungsmerkmal (HOM)

- gestreute Speicherung: Pseudoplätze (Adressraum belegen, viele Lücken, $O(1)$), auf Sätze beliebig zugreifen
- gestreute Speicherung mit indirekten Adressen
  Bsp. 20.000 Sätze mit 10 stelliger Materialnummer, Kapazität +20%, Divisor festlegen (möglichst Primzahl), Rest oben speichern $\Rightarrow$ nur eindeutig, daher Kollisionsstrategie notwendig (z.B. sequentielle Suche bis freier Platz)

8.9 Files

8.10 Makros

simple Textersetzung

```
#define QUAD(x) ((x)*(x))
```

9 C++

9.1 Grundlegende Unterschiede

Ein- und Ausgabe: `cin >> a;`

```
cout << "Text" << a << endl;
cerr
ifstream fin("eingabe.txt)"; // mit fin >> a nutzen
ofstream fout("eingabe.txt)"; // mit fout << a nutzen
```

Dynamische Datenobjekte: Operatoren `new` und `delete`, bei Feldern

```
int *pa = new int[14];
delete[] pa;
```

oder die Erzeugung einer einfach verketteten Liste in C++:

```
struct node
{
  int key;
  struct node* next;
};
node* p;
node* q;
node* First = new node;
First->key = 1;
p = First;
for (int i=2;i<=10;i++)
{
  q = new node;
  q->key = i;
  p->next = q;
  p=q;
}
p->next = NULL;
```

Referenzen: call by reference, praktischer als Pointerübergabe

```
void quad(double &a)
{ a = a*a; }
...
quad(b); // Aufruf
```

Überladen von Funktionen: Funktionen mit gleichem Namen aber anderen Parametern überladen werden, intern über Signaturen unterschieden

```
double max (double a, double b) {..}
int max (int a, int b) {...}
```

Namensbereich: direkt ansprechbarer Namensbereich

```
namespace ABC
{
  int x;
}
main ()
{
  ABC::x = 1.5;
}
```

9.2 Objektorientierung

Abstrakte Datentypen: Klassen = Daten + Funktionen, über die der Nutzer die Daten verändern kann

```
class komplex
{
private: // default
  double re, im;
public:
 // Konstuktoren
  komplex(double r=0, double i=0):
    re(r),im(i) {;}; // inline Typwandl. und Standard
  komplex(const char &); // member-Fkt Typwandl.
  komplex(const komplex &); // Kopier-Konstruktor

// Destruktor
  ~komplex();

// Operator-Überladung
// member-Funktion
  komplex operator=(const komplex &);
  komplex operator+(const komplex &);
```

```
  komplex & operator+=(const komplex &);
  komplex operator*(const komplex &);
  komplex operator*(double);
  komplex & operator*=(double);
  friend komplex operator*(double, complex &);

// Prototyp für Ausgabe mit cout << komplex als Freund
  friend ostream& operator<<(ostream &, const komplex &);

// member-Funktionen als Protoypendeklaration
  double betrag(const komplex &);
  double phi(const komplex &);
}

// Memberfunktionen implementieren
double komplex::betrag(const komplex & z)
{
  return sqrt(z.re*z.re+z.im*z.im);
}

// cout implementieren
ostream & operator<<(ostream & os, const komplex & z)
{
  os << z.re << "+i" << z.im;
  return os;
}
```

Deklaration: Zwei Objekte `heute` und `morgen` der Klasse `datum` deklarieren:
`datum heute, morgen;`

Aufträge: `heute.set_datum();` (nur auf public-Werte möglich)

Konstruktoren: spezielle Memberfunktionen, die Initialisierung durchführen

- (System-) Default-Konstruktoren
- Name identisch mit Klassenname
- kein Return-Wert

```
datum::datum() {...} // belegt Speicherplatz
```

```
datum::datum(int t, int m, int j)
{ tag=t; ...}

datum::datum(int t, int m, int j) // Listen-Init.
tag(t), monat(m), jahr(j)
{...}
```

dann Objekinstanzierung

```
datum heute;
datum morgen(10,9,2006);
```

Kopierkonstruktoren: automatisch vom System erzeugt, bei dynamischer Speicherplatzverwaltung selber schreiben

```
datum morgen=heute;
datum morgen(heute);
```

Destruktoren: Gegenstück zu Konstruktoren, werden vom System bei Verlassen des Gültigkeitsbereichs ausgeführt, Name immer `~classname`, kein Rückgabewert oder Parameter

```
datum::~datum() {;} // korrekt,
```

friend-Klassen und -Funktionen: Explizite Erlaubnis auf interne Daten zuzugreifen, hierfür in der Klassendefinition eintragen

```
friend class beispiel2; // Freundklasse
friend void ShowImage(const Img&); // Freundfunktion
```

C++-Standardklassen: einige wichtige Standardklassen von C++:

string: Klasse für Zeichenketten

```
#include<string>
string bez1, bez2("hallo ");
string bez3(bez2);
bez1 = bez2 + bez3;
bez1 += "ABC";
```

vector: Klasse für beliebige Vektoren

```
#include<vector>
vector<int> vec(10);
vector<double> vec2(n);
int m = v.size(); // Vektorlänge
vec.push_back(wert); // Element am Ende eintragen
vec.pop_back(); // letztes Element löschen
vector<vector<int>> a(m,vector<int>(n)); // Matrix
```

bool: logischer Datentyp

```
#include<bool>
char zeichen;
log = (zeichen >= 'A' && zeichen <= 'Z' );
```

complex: Rechnen mit komplexen Zahlen, Member- und Freundfunktionen implementiert

```
#include<complex>
complex<double> a(1.0,-1.5);
d1 = a.real;
d2 = real(a);
r = a.norm;
phi = a.arg;
b = conj(a);
```

Überladen von Operatoren: Operatoren wie `+`, `-` und `&` können überladen werden:

1. Operator-Memberfunktion: `Klassenname::operator+(Args) {...}`
2. spezielle Operator-Funktion: `operator+(Args) {...}`