23. Gierige (Greedy) Algorithmen

Gebrochenes Rucksack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

Das Gebrochene Rucksackproblem

Menge von $n \in \mathbb{N}$ Gegenständen $\{1, \ldots, n\}$ gegeben. Jeder Gegenstand i hat Nutzwert $v_i \in \mathbb{N}$ und Gewicht $w_i \in \mathbb{N}$. Das Maximalgewicht ist gegeben als $W \in \mathbb{N}$. Bezeichnen die Eingabe mit $E = (v_i, w_i)_{i=1,\ldots,n}$.

Gesucht: Anteile $0 \leq q_i \leq 1$ $(1 \leq i \leq n)$ die die Summe $\sum_{i=1}^{n} q_i \cdot v_i$ maximieren unter $\sum_{i=1}^{n} q_i \cdot w_i \leq W$.

Gierige (Greedy) Heuristik

Sortiere die Gegenstände absteigend nach Nutzen pro Gewicht v_i/w_i.
Annahme $v_i/w_i \geq v_{i+1}/w_{i+1}$
Sei $j = \max\{0 \leq k \leq n : \sum_{i=1}^{k} w_i \leq W\}$. Setze

- $q_i = 1$ für alle $1 \leq i \leq j$.
- $q_{j+1} = \frac{W - \sum_{i=1}^{j} w_i}{w_{j+1}}$.
- $q_i = 0$ für alle $i > j + 1$.

Das ist schnell: $\Theta(n \log n)$ für Sortieren und $\Theta(n)$ für die Berechnung der q_i.

Korrektheit

Annahme: Optimale Lösung (r_i) $(1 \leq i \leq n)$.
Der Rucksack wird immer ganz gefüllt: $\sum_i r_i \cdot w_i = \sum_i q_i \cdot w_i = W$.
Betrachte k: kleinstes i mit $r_i \neq q_i$. Die gierige Heuristik nimmt per Definition so viel wie möglich: $q_k > r_k$. Sei $x = q_k - r_k > 0$.
Konstruiere eine neue Lösung (r'_i): $r'_i = r_i \forall i < k$. $r'_k = q_k$. Entferne Gewicht $\sum_{i=k+1}^{n} \delta_i = x \cdot w_k$ von den Gegenständen $k+1$ bis n. Das geht, denn $\sum_{i=k}^{n} r_i \cdot w_i = \sum_{i=k}^{n} q_i \cdot w_i$.

Korrektheit

$$\begin{aligned}\sum_{i=k}^{n} r'_i v_i &= r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} (r_i w_i - \delta_i) \frac{v_i}{w_i} \\ &\geq r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} r_i w_i \frac{v_i}{w_i} - \delta_i \frac{v_k}{w_k} \\ &= r_k v_k + x w_k \frac{v_k}{w_k} - x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^{n} r_i w_i \frac{v_i}{w_i} = \sum_{i=k}^{n} r_i v_i.\end{aligned}$$

Also ist (r'_i) auch optimal. Iterative Anwendung dieser Idee erzeugt die Lösung (q_i).

Huffman-Codierungen

Ziel: Speicherplatzeffizientes Speichern einer Folge von Zeichen mit einem binären **Zeichencode** aus **Codewörtern**.

Huffman-Codierungen

Ziel: Speicherplatzeffizientes Speichern einer Folge von Zeichen mit einem binären **Zeichencode** aus **Codewörtern**.

Beispiel

File aus 100.000 Buchstaben aus dem Alphabet $\{a, \ldots, f\}$

	a	b	c	d	e	f
Häufigkeit (Tausend)	45	13	12	16	9	5
Codewort fester Länge	000	001	010	011	100	101
Codewort variabler Länge	0	101	100	111	1101	1100

Huffman-Codierungen

Ziel: Speicherplatzeffizientes Speichern einer Folge von Zeichen mit einem binären **Zeichencode** aus **Codewörtern**.

Beispiel

File aus 100.000 Buchstaben aus dem Alphabet $\{a, \ldots, f\}$

	a	b	c	d	e	f
Häufigkeit (Tausend)	45	13	12	16	9	5
Codewort fester Länge	000	001	010	011	100	101
Codewort variabler Länge	0	101	100	111	1101	1100

Speichergrösse (Code fixe Länge): 300.000 bits.
Speichergrösse (Code variabler Länge): 224.000 bits.

Huffman-Codierungen

- Betrachten **Präfixcodes**: kein Codewort kann mit einem anderen Codewort beginnen.

Huffman-Codierungen

- Betrachten **Präfixcodes**: kein Codewort kann mit einem anderen Codewort beginnen.
- Präfixcodes können im Vergleich mit allen Codes die optimale **Datenkompression** erreichen (hier ohne Beweis).

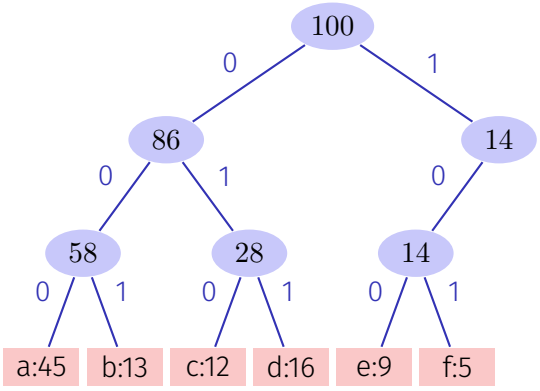
Huffman-Codierungen

- Betrachten **Präfixcodes**: kein Codewort kann mit einem anderen Codewort beginnen.
- Präfixcodes können im Vergleich mit allen Codes die optimale **Datenkompression** erreichen (hier ohne Beweis).
- Codierung: Verkettung der Codewörter ohne Zwischenzeichen (Unterschied zum Morsen!)
  $affe \to 0 \cdot 1100 \cdot 1100 \cdot 1101 \to 0110011001101$

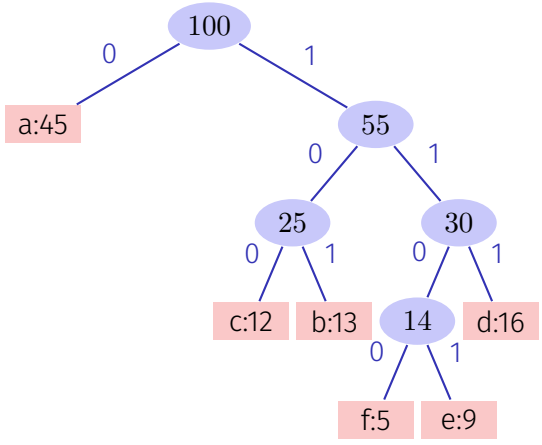
Huffman-Codierungen

- Betrachten **Präfixcodes**: kein Codewort kann mit einem anderen Codewort beginnen.
- Präfixcodes können im Vergleich mit allen Codes die optimale **Datenkompression** erreichen (hier ohne Beweis).
- Codierung: Verkettung der Codewörter ohne Zwischenzeichen (Unterschied zum Morsen!)
  $affe \to 0 \cdot 1100 \cdot 1100 \cdot 1101 \to 0110011001101$
- Decodierung einfach da Präfixcode
  $0110011001101 \to 0 \cdot 1100 \cdot 1100 \cdot 1101 \to affe$

Codebäume



Codewörter fixer Länge



Codewörter variabler Länge

Eigenschaften der Codebäume

- Optimale Codierung eines Files wird immer durch vollständigen binären Baum dargestellt: jeder innere Knoten hat zwei Kinder.

Eigenschaften der Codebäume

- Optimale Codierung eines Files wird immer durch vollständigen binären Baum dargestellt: jeder innere Knoten hat zwei Kinder.
- Sei C die Menge der Codewörter, $f(c)$ die Häufigkeit eines Codeworts c und $d_T(c)$ die Tiefe eines Wortes im Baum T. Definieren die **Kosten** eines Baumes als

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

(Kosten = Anzahl Bits des codierten Files)

Eigenschaften der Codebäume

- Optimale Codierung eines Files wird immer durch vollständigen binären Baum dargestellt: jeder innere Knoten hat zwei Kinder.
- Sei C die Menge der Codewörter, $f(c)$ die Häufigkeit eines Codeworts c und $d_T(c)$ die Tiefe eines Wortes im Baum T. Definieren die **Kosten** eines Baumes als

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

  (Kosten = Anzahl Bits des codierten Files)

Bezeichnen im folgenden einen Codebaum als optimal, wenn er die Kosten minimiert.

Algorithmus Idee

Baum Konstruktion von unten nach oben

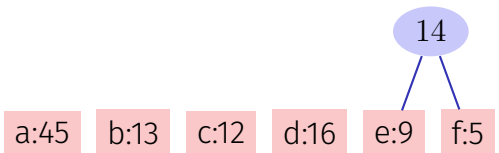
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.

a:45 b:13 c:12 d:16 e:9 f:5

Algorithmus Idee

Baum Konstruktion von unten nach oben

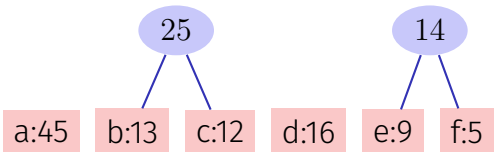
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Idee

Baum Konstruktion von unten nach oben

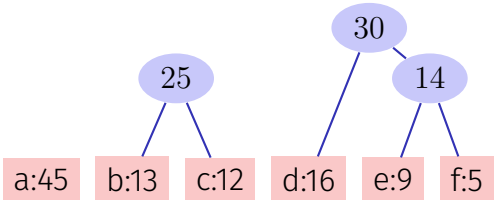
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Idee

Baum Konstruktion von unten nach oben

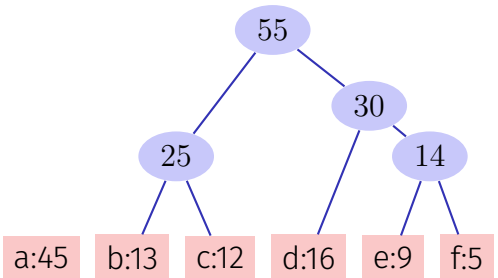
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Idee

Baum Konstruktion von unten nach oben

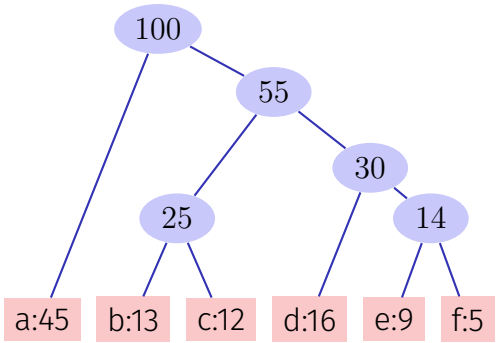
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Idee

Baum Konstruktion von unten nach oben

- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Huffman(C)

Input: Codewörter $c \in C$
Output: Wurzel eines optimalen Codebaums

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
    Alloziere neuen Knoten z
    z.left $\leftarrow$ ExtractMin(Q) // Extrahiere Wort mit minimaler Häufigkeit.
    z.right $\leftarrow$ ExtractMin(Q)
    z.freq $\leftarrow$ z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)

Analyse

Verwendung eines Heaps: Heap bauen in $\mathcal{O}(n)$. Extract-Min in $O(\log n)$ für n Elemente. Somit Laufzeit $O(n \log n)$.

Das gierige Verfahren ist korrekt

Theorem 21

Seien x, y zwei Symbole mit kleinsten Frequenzen in C und sei $T'(C')$ der optimale Baum zum Alphabet $C' = C - \{x, y\} + \{z\}$ mit neuem Symbol z mit $f(z) = f(x) + f(y)$. Dann ist der Baum $T(C)$ der aus $T'(C')$ entsteht, indem der Knoten z durch einen inneren Knoten mit Kindern x und y ersetzt wird, ein optimaler Codebaum zum Alphabet C.

Beweis

Es gilt
$f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x)+f(y)) \cdot (d_{T'}(z)+1) = f(z) \cdot d_{T'}(x) + f(x) + f(y)$.
Also $B(T') = B(T) - f(x) - f(y)$.

Annahme: T sei nicht optimal. Dann existiert ein optimaler Baum T'' mit $B(T'') < B(T)$. Annahme: x und y Brüder in T''. T''' sei der Baum T'' in dem der innere Knoten mit Kindern x und y gegen z getauscht wird. Dann gilt $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$.
Widerspruch zur Optimalität von T'.

Die Annahme, dass x und y Brüder sind in T'' kann man rechtfertigen, da ein Tausch der Elemente mit kleinster Häufigkeit auf die unterste Ebene den Wert von B höchstens verkleinern kann.

24. C++ vertieft (IV): Ausnahmen (Exceptions)

Was kann schon schiefgehen?

- Öffnen einer Datei zum Lesen oder Schreiben

```
std::ifstream input("myfile.txt");
```

- Parsing

```
int value = std::stoi("12-8");
```

- Speicherallokation

```
std::vector<double> data(ManyMillions);
```

- Invalide Daten

```
int a = b/x; // what if x is zero?
```

Möglichkeiten der Fehlerbehandlung

- Keine (inakzeptabel)
- Globale Fehlervariable (Flags)
- Funktionen, die Fehlercodes zurückgeben
- Objekte, die Fehlercodes speichern
- Ausnahmen

Globale Fehlervariablen

- Typisch für älteren C-Code
- Nebenläufigkeit ist ein Problem
- Fehlerbehandlung nach Belieben. Erfordert grosse Disziplin, sehr gute Dokumentation und übersäht den Code mit scheinbar unzusammenhängenden Checks.

Rückgabe von Fehlercodes

- Jeder Aufruf einer Funktion wird mit Ergebnis quittiert.
- Typisch für grosse APIs (OS Level). Dort oft mit globalen Fehlercodes kombiniert.[42]
- Der Aufrufer kann den Rückgabewert einer Funktion prüfen, um die korrekte Ausführung zu überwachen.

[42] Globaler error code thread-safety durch thread local storage.

Rückgabe von Fehlercodes

Beispiel

```
#include <errno.h>
...

pf = fopen ("notexisting.txt", "r+");
if (pf == NULL) {
  fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
}
else { // ...
  fclose (pf);
}
```

Fehlerstatus im Objekt

- Fehlerzustand eines Objektes intern im Objekt gespeichert.

Beispiel

```
int i;
std::cin >> i;
if (std::cin.good()){// success, continue
  ...
}
```

Exceptions

- Exceptions unterbrechen den normalen Kontrollfluss
- Exceptions können geworfen (throw) und gefangen (catch) werden
- Exceptions können über Funktionengrenzen hinweg agieren.

Beispiel: Exception werfen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main()
{
  f(4);
  return 0;
}
```

Beispiel: Exception werfen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main()
{
  f(4);
  return 0;
}
```

```
terminate called after throwing an instance of 'MyException'
Aborted
```

Beispiel: Exception fangen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main(){
  try{
    f(4);
  }
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

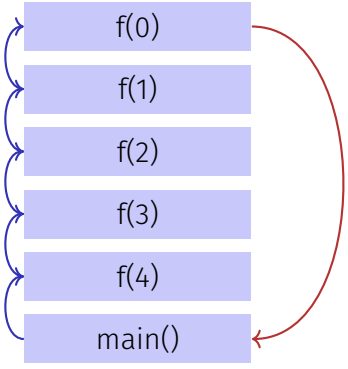
Beispiel: Exception fangen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main(){
  try{
    f(4);
  }
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

exception caught



Resourcen werden geschlossen

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
  if (i==0) throw MyException();
  SomeResource x;
  f(i-1);
}
int main(){
  try{f(5);}
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

Resourcen werden geschlossen

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
  if (i==0) throw MyException();
  SomeResource x;
  f(i-1);
}
int main(){
  try{f(5);}
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

```
closed resource
closed resource
closed resource
closed resource
closed resource
exception caught
```

Wann Exceptions?

Exceptions werden für die **Behandlung von Fehlern** benutzt.

- Verwende **`throw`** nur, um einen Fehler zu signalisieren, welcher die Postcondition einer Funktion verletzt oder das Fortfahren des Codes unmöglich macht
- Verwende **`catch`** nur, wenn klar ist, wie man den Fehler behandeln kann (u.U. mit erneutem Werfen der Exception)
- Verwende **`throw`** **nicht** um einen Programmierfehler oder eine Verletzung von Invarianten anzuzeigen (benutze stattdessen **`assert`**)
- Verwende Exceptions **nicht** um den Kontrollfluss zu ändern. Throw ist **nicht** ein besseres return.

Warum Exceptions?

Das:

```
int ret = f();
if (ret == 0) {
  // ...
} else {
  // ...code that handles the error...
}
```

sieht auf den ersten Blick vielleicht besser / einfacher aus als das:

```
try {
  f();
  // ...
} catch (std::exception& e) {
  // ...code that handles the error...
}
```

Warum Exceptions?

Die Wahrheit ist, dass Einfachstbeispiele den Kern der Sache nicht immer treffen.

Return-Codes zur Fehlerbehandlung übersähen grössere Codestücke entweder mit Checks oder die Fehlerbehandlung bleibt auf der Strecke.

Darum Exceptions

Beispiel 1: Evaluation von Ausdrücken (Expression Parser, Vorlesung Informatik I)

Eingabe: `1 + (3 * 6 / (/ 7 ))`

Fehler tief in der Rekursionshierarchie. Wie kann ich eine vernünftige Fehlermeldung produzieren (und weiterfahren)? Müsste den Fehlercode über alle Rekursionsstufen zurückgeben. Das übersäht den Code mit checks.

Beispiel 2

Wertetyp mit Garantie: Werte im gegebenen Bereich.

```
template <typename T, T min, T max>
class Range{
public:
  Range(){}
  Range (const T& v) : value (v) {
    if (value < min) throw Underflow ();
    if (value > max) throw Overflow ();
  }
  operator const T& () const {return value;}
private:
  T value;
};
```

Fehlerbehandlung im Konstruktor!

Fehlertypen, hierarchisch

```
class RangeException {};
class Overflow : public RangeException {};
class Underflow : public RangeException {};
class DivisionByZero: public RangeException {};
class FormatError: public RangeException {};
```

Operatoren

```
template <typename T, T min, T max>
Range<T, min, max> operator/ (const Range<T, min, max>& a,
                              const Range<T, min, max>& b){
  if (b == 0) throw DivisionByZero();
  return T (a) * T(b);
}

template <typename T, T min, T max>
std::istream& operator >> (std::istream& is, Range<T, min, max>& a){
  T value;
  if (!(is >> value)) throw FormatError();
  a = value;
  return is;
}
```

Fehlerbehandlung im Operator!

Fehlerbehandlung (zentral)

```
Range<int,-10,10> a,b,c;
try{
  std::cin >> a;
  std::cin >> b;
  std::cin >> c;
  a = a / b + 4 * (b - c);
  std::cout << a;
}
catch(FormatError& e){ std::cout << "Format error\n"; }
catch(Underflow& e){ std::cout << "Underflow\n"; }
catch(Overflow& e){ std::cout << "Overflow\n"; }
catch(DivisionByZero& e){ std::cout << "Divison By Zero\n"; }
```