Algorithmen und Datenstrukturen

Gierige Algorithmen

Prof. Justus Piater, Ph.D.

29. Mai 2024

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Einführung

In diesem und den folgenden beiden Kapiteln widmen wir uns drei wichtigen algorithmischen Paradigmen, die uns helfen, schwierige Anwendungsprobleme im Lichte gut verstandener algorithmischer Theorien zu formalisieren.

Der Begriff *Paradigma* bedeutet grundsätzliche Denkweise, Erklärungsmuster, oder Herangehensweise. Die drei wichtigsten algorithmischen Paradigmen sind ***gierige*** Algorithmen, ***Teile und Herrsche***, und die ***dynamische Programmierung***. Für viele Anwendungsprobleme eignet sich eines dieser Paradigmen, selten mehr als eines, manchmal keines von diesen.

Gierige Algorithmen wählen iterativ jeweils den größtmöglichen Schritt, was auch immer *größtmöglich* im jeweiligen Problem bedeutet. Ein gieriger Algorithmus ist dann sinnvoll, wenn diese Sequenz gieriger Schritte tatsächlich zu einer optimalen Gesamtlösung führt.

Teile-und-Herrsche-Algorithmen zerlegen ein Problem ein kleinere, *disjunkte* Versionen desselben Problems, und lösen diese rekursiv. Dies ist dann ein gutes Vorgehen, falls die aus den Teillösungen zusammengestzte Gesamtlösung tatsächlich optimal für das Gesamtproblem ist.

Ist ein Problem nicht in voneinander unabhängige Teilprobleme zerlegbar, aber immerhin in gleichartige Unterprobleme, die sich überlappen, dann bietet sich möglicherweise eine Lösung mittels dynamischer Programmierung an. Dynamische Programmierung führt oft zu extrem einfachen und effizienten Algorithmen, die im Wesentlichen systematisch eine Tabelle auf Basis bereits berechneter Einträge ausfüllen.

In diesem Kapitel beginnen wir nun mit *gierigen* Algorithmen.

Algorithmische Paradigmen [Slide 1]

Paradigma: grundsätzliche Denkweise, Muster, Erklärungsmodell, Ansatz, Herangehensweise

Wir werden die folgenden drei algorithmischen Paradigmen näher betrachten:

- ***Gierige*** Algorithmen: Wähle *iterativ* jeweils den *größtmöglichen* Schritt.
- ***Teile und herrsche***: Zerlege das Problem in kleinere, *disjunkte* Versionen desselben Problems, löse diese *rekursiv*, und setze die Gesamtlösung aus den Teillösungen zusammen.
- ***Dynamische Programmierung***: Zerlege das Problem in kleinere, *überlappende* Versionen desselben Problems, und baue die Gesamtlösung *iterativ* auf Lösungen der Teilprobleme auf, indem eine *Tabelle* systematisch ausgefüllt wird.

1 Münzrückgabe

Video 1 beginnt hier.

Definition

Viele Probleme lassen sich dadurch schrittweise lösen, dass man bei jedem Schritt möglichst weit Richtung Ziel vorangeht. Solche Algorithmen bezeichnet man als ***gierig***, oder auf englisch, *greedy*.

Münzrückgabe [Slide 2]

Problem: Jemand schuldet dir x Euro. Er gibt dir $y \geq x$ Euro. Gib Wechselgeld in einer minimalen Anzahl von Münzen der Werte $S = \{s_0, \ldots, s_{N-1}\}$.

Lösung für $S = \{1, 2, 5, 10, 20, 50\}$ Cent: Wähle die Münze mit dem größten Wert kleiner oder gleich der verbleibenden Differenz, bis diese Differenz verschwindet.

Funktioniert dies auch für $S = \{1, 20, 50\}$ Cent?

E.g., gib 60 Cent zurück.

Beispiel

Ein einfaches Beispiel ist das Ausgeben von Wechselgeld. Nehmen wir an, Sie kaufen sich ein Eis für 3€40. Die Verkäuferin überreicht Ihnen das Waffelhörnchen, innen mit in Schokolade getränkten Erdnusssplittern bestrichen und mit Eiscreme überquellend gefüllt, obenauf ein wenig Schlagsahne und Schokosplitter.

Sie reichen der Verkäuferin einen 5€-Schein, und möchten so wenig wie möglich Münzen in Ihrer Hosentasche herumtragen. Die Kasse enthält einen praktisch unbegrenzten Vorrat an Münzen im Wert von 1, 2, 5, 10, 20, und 50 Cent und 1 und 2 Euro. Wie ermittelt die Verkäuferin die minimale Menge Münzen im Wert Ihres Wechselgelds von 1€60?

Sie folgt einem einfachen, gierigen Algorithmus: Sie nimmt die größte Münze, deren Wert das verbleibende Wechselgeld nicht übersteigt, nämlich eine 1€-Münze. Diesen Wert subtrahiert sie von den 1€60; es verbleiben also 60 Cent Wechselgeld. Sie nimmt wiederum die größte Münze, deren Wert diese 60 Cent nicht übersteigt, nämlich eine 50-Cent-Münze. Damit verbleibt ein Wechselgeld von 10 Cent, das sie mit einer 10-Cent-Münze auf Null reduziert.

Sie hat also die zurückzugebenden Münzen bestimmt, indem sie Schritt für Schritt jeweils die *größte* Münze gewählt hat, deren Wert das verbleibende Wechselgeld nicht übersteigt. Jeder dieser Schritte war *gierig* im Sinne gieriger Algorithmen.

Das Ergebnis von einer 1€, einer 50-Cent- und einer 10-Cent-Münze ist tatsächlich minimal unter allen Münzmengen, deren Wert dem gesamten Wechselgeld von 1€60 entspricht. Dies zu beweisen ist jedoch überraschend nicht-trivial.

Nehmen wir an, die Europäische Zentralbank führt eine 80-Cent-Münze ein, speziell für Eisfans wie uns. Damit wird mit zwei 80-Cent-Münzen eine bessere Lösung Ihres Wechselgeldproblems ermöglicht. Der gierige Algorithmus würde allerdings dennoch als erstes die 1-€-Münze wählen, weil sie das verbleibende Wechselgeld am stärksten reduziert, und anschließend jeweils eine 50- und 10-Cent-Münze.

Folglich würden Sie sich bei der Eisverkäuferin darüber beschweren, dass ihr Wechselgeld Ihre Hosentasche unnötig beschwert, weil ihr gieriger Algorithmus das optimale Rückgeld von zwei 80-Cent-Münzen nicht findet. Um Ihnen ein optimales Rückgeld zu garantieren, müsste sie allerdings im Allgemeinen ein rechnerisch sehr komplexes Problem lösen. Darauf möchte allerdings der nächste Kunde nicht warten!

Unser einfacher, gieriger Algorithmus findet bei unserem Münzwertsystem ohne die 80-Cent-Münze tatsächlich immer eine optimale Lösung. Münzwertsysteme mit dieser Eigenschaft heißen *kanonisch*. Die meisten realen Münzwertsysteme sind aus diesem Grunde kanonisch.

2 *Huffman Coding*: Einführung

Video 2 beginnt hier.

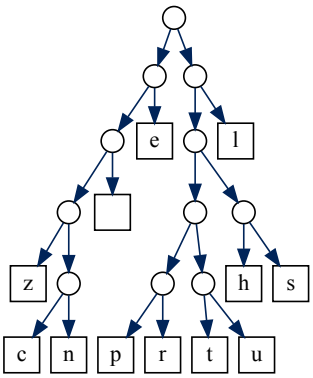
Text-Kompression [Slide 3]

Ziel: Repräsentiere eine Zeichenkette mit möglichst wenigen Bits.

Idee:

- Verwende kurze Codes für häufige Zeichen, und umgekehrt.
- Repräsentiere den Code als Binärbaum, wobei linke und rechte Kanten jeweils für 0 bzw. 1 stehen, und Blätter für Zeichen.

  ⇒ Kein Code ist ein Präfix eines anderen Code.



Beispiel

```
helle pelle schnell zur stelle
```

30 Zeichen = 210 Bits in 7-Bit ASCII Code oder 120 Bits in 4-Bit Code

12 verschiedene Zeichen

92 Bits in Huffman Code:

```
10100111110100110000011111010011011000101010000110111110010000100111000100110111001001111
```

```
  001    c 00010   e 01      h 1010
l 11     n 00011   p 10000   r 10001
s 1011   t 10010   u 10011   z 0000
```

Erklärung

Betrachten wir nun einen etwas weniger trivialen, gierigen Algorithmus. Unsere Anwendung ist hier die Text-Kompression: Wir möchten eine Zeichenkette mit möglichst wenigen Bits repräsentieren, um Speicherplatz oder Bandbreite zu sparen.

Zeichenketten werden meist als UTF-8-codierte Unicode-Codepoints repräsentiert. Die ersten 128 Codepoints entsprechen dem historischen 7-Bit-ASCII-Zeichensatz. Die nächsten 128 Codepoints, also 7 Bits mit dem 8. Bit gleich 1, wurden für verschiedene Sätze nationaler Sonderzeichen genutzt. Daher rührt die traditionelle Bezeichnung einer 8 Bit breiten Variablen als *character*, also Zeichen.

In unserem Beispiel verwenden wir ASCII-Zeichen. In der traditionellen Darstellung repräsentiert also jedes Byte ein Zeichen, und das 8. Bit ist immer 0. Eine solche Sequenz aus 7-Bit-ASCII-Zeichen können wir also trivial auf 7/8 ihrer Länge komprimieren, indem wir das 8. Bit jedes Zeichens eliminieren.

Hier ist ein Beispiel-Text, der ein Werbespruch für einen Paket-Zustelldienst sein könnte, zur Illustration komplett in Kleinbuchstaben: *helle pelle schnell zur stelle*. Das sind 30 Zeichen, die also aus insgesamt 30×8=240 Bits bestehen. Ohne die nicht verwendeten 8. Bits können wir diesen Text also leicht auf 30×7=210 Bits komprimieren.

Es gibt viele Methoden, einen solchen Text noch weiter zu komprimieren. Beim Huffman-Coding besteht die zentrale Beobachtung darin, dass alle Zeichen 7 Bit breit sind. Wenn

wir stattdessen häufig auftretende Zeichen mit weniger Bits repräsentieren, dann können wir meist die Gesamtlänge der Repräsentation verkürzen, auch wenn wir für selten auftretende Zeichen mehr als 7 Bits aufwenden müssen.

Eines müssen wir allerdings beachten: Wenn Zeichencodes verschieden lang sein können, dann muss trotzdem für den Decoder klar sein, wo ein Zeichencode endet und der nächste beginnt. Wir haben ja keine Leerzeichen dazwischen, sondern einfach nur eine lückenlose Sequenz von Nullen und Einsen.

Eine naheliegende Lösung ist, die Zeichen so zu codieren, dass kein Zeichencode ein Präfix eines anderen Zeichencodes ist. Dann muss der Decoder lediglich für jedes Zeichen so viele Bits konsumieren, bis ein gültiger Zeichencode eingelesen ist, und kann sicher sein, dass das Konsumieren weiterer Bits nicht zu einem längeren, gültigen Zeichencode führen kann.

Einen solchen Code können wir als Binärbaum darstellen, wie hier für unseren Beispielspruch gezeigt. Jedes Blatt enthält ein Zeichen. Der Code eines gegebenen Zeichens ergibt sich aus dem Pfad von der Wurzel bis zu seinem Blatt: Eine Traversierung einer linken Kante entspricht einer binären 0, und eine rechte Kante einer binären 1.

Der Pfad von der Wurzel zum Buchstaben `e` führt beispielsweise zuerst über eine linke und dann eine rechte Kante. Damit besteht sein Code aus der Bitsequenz 01. Die Kanten des Pfades zum Buchstaben `r` sind rechts-links-links-links-rechts; sein Code ist also die Bitsequenz 10001.

Da jeder Pfad von der Wurzel zu einem Blatt einen längstmöglichen gerichteten Pfad darstellt, ist es unmöglich, dass ein Pfad (und damit Zeichencode) mit dem Präfix eines anderen Pfades (also Zeichencodes) identisch ist.

Den Code jedes Zeichens können wir auf diese Weise einfach aus diesem Baum auslesen. Damit können wir unseren Beispielspruch codieren: `h` ist 1010, gefolgt von `e`, also 01, gefolgt von zwei `l`, also zweimal 11, und so weiter.

So erstellen wir die hier gezeigte Bitsequenz. Sie umfasst lediglich 92 Bits, also weniger als die Hälfte der 7-Bit-ASCII-Codierung desselben Texts.

Unser kurzer Beispieltext enthält allerdings bloß 12 verschiedene Zeichen. Um diese 12 verschiedenen Zeichen mit einer fixen Anzahl Bits zu codieren, benötigen wir lediglich 4 Bits, nicht 7. Mit diesem Code muss sich unser Huffman-Code also messen. Er gewinnt jedoch auch hier, denn um 30 Zeichen zu codieren, benötigt unser 4-Bit-Code $30 \times 4 = 120$ Bits. Es zahlt sich hier also aus, die häufigsten Zeichen in weniger als 4 Bits zu codieren, obwohl 6 unserer 12 verschiedenen Zeichen jeweils 5 Bits erfordern.

Wollen wir den Code verwenden, um diesen einen Text zu übertragen, dann genügt es nicht, ausschließlich den codierten Text zu übertragen; wir müssen auch den Code mitübertragen, damit der Empfänger ihn wieder decodieren kann. Dies würde die Platzeffizienz entsprechend reduzieren. Möchten wir jedoch viel Text übertragen, dann zahlt sich die Codierung aus, da ihr Platzbedarf für einen gegebenen Zeichensatz konstant ist.

Wie erstellen wir einen solchen Huffman-Code? Die Grundidee des Huffman-Algorithmus ist es, Pfade von der Wurzel zu häufig auftretenden Zeichen kürzer zu halten als Pfade zu selteneren Zeichen.

Erstellung eines Huffman-Codes [Slide 4]

```
helle pelle schnell zur stelle
```

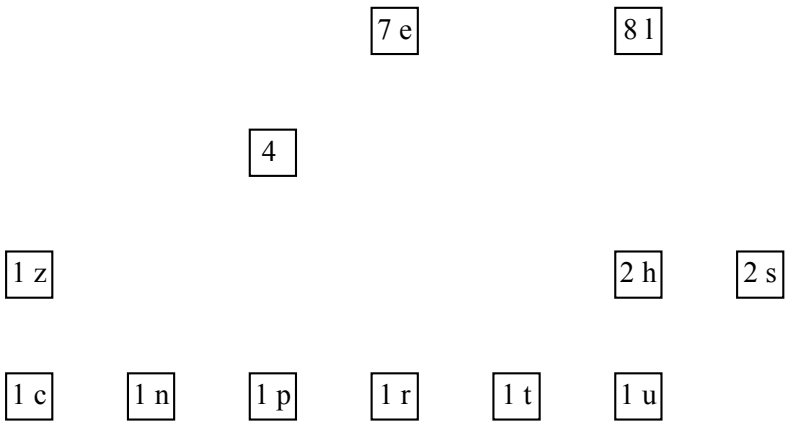
Zeichen		c	e	h	l	n	p	r	s	t	u	z
Häufigkeit	4	1	7	2	8	1	1	1	2	1	1	1

Erklärung

Wir beginnen also damit, die Vorkommen der Zeichen in unserem zu codierenden Text zu zählen. Das Ergebnis zeigt diese Tabelle: Das Leerzeichen kommt viermal vor, das `l` ist das häufigste Zeichen mit 8 Vorkommen, und 7 verschiedene Buchstaben treten jeweils nur einmal auf.

Nun konstruieren wir einen Binärbaum, dessen Blätter für seltene Zeichen eine größere Tiefe haben als solche für häufigere Zeichen. Dies tun wir, indem wir den Baum von unten nach oben aufbauen, beginnend mit den seltensten Zeichen.

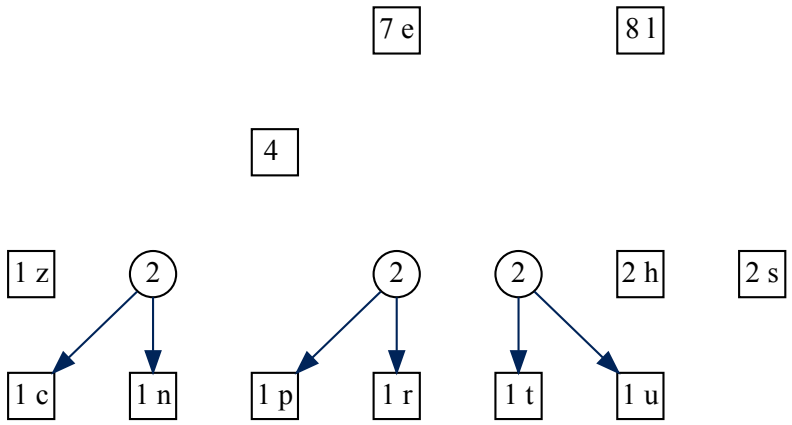
Erstellung eines Huffman-Codes [Slide 5]



Erklärung

Zu Beginn erstellen wir d triviale Bäume, die jeweils nur aus einem Blatt bestehen. Jedes dieser Blätter enthält jeweils eines der d unterschiedlichen Zeichen unseres Texts. Jedem dieser Bäume weisen wir ein Gewicht zu, nämlich die Häufigkeit des durch ihn repräsentierten Zeichens.

Erstellung eines Huffman-Codes [Slide 6]

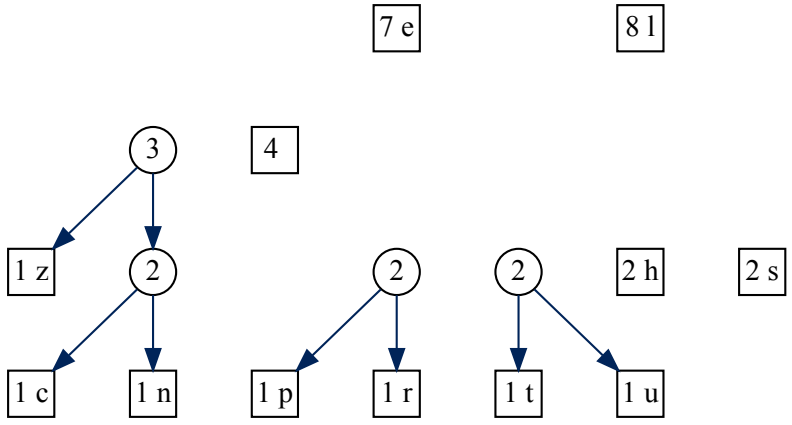


Erklärung

Nun fassen wir iterativ jeweils zwei Bäume niedrigsten Gewichts durch Hinzufügen einer gemeinsamen Wurzel zusammen. Diesen neuen Baum gewichten wir mit der Summe der Gewichte seiner beiden Unterbäume.

In diesem Beispiel fassen wir die Bäume mit den Zeichen `c` und `n` zu einem neuen Baum mit Gewicht 2 zusammen, ebenso die Bäume mit den Zeichen `p` und `r` sowie die beiden Bäume mit den Zeichen `t` und `u`.

Erstellung eines Huffman-Codes [Slide 7]



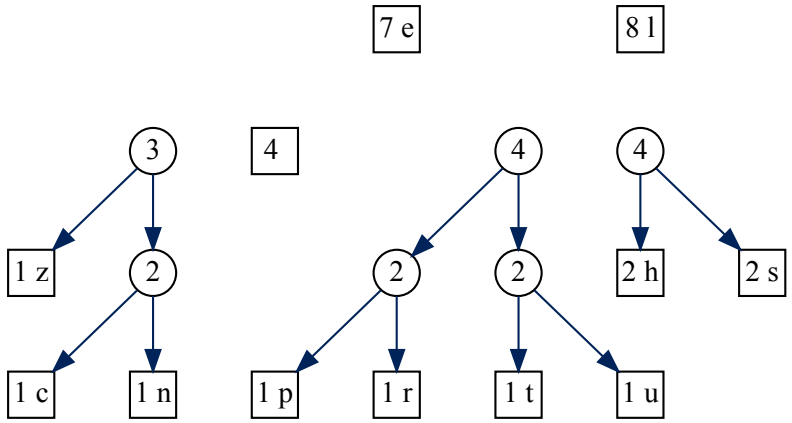
Erklärung

Nun haben wir nur noch einen Baum mit Gewicht 1 übrig, den mit dem `z`. Diesen fassen wir als nächstes mit einem beliebigen Baum mit Gewicht 2 zu einem Baum mit Gewicht 3 zusammen.

Wir sehen, dass es oft viele äquivalente Möglichkeiten gibt, aus Bäumen gleichen Gewichts einen auszuwählen. Damit ist klar, dass es i.A. viele verschiedene, äquivalente Huffman-Codes für einen gegebenen Text gibt.

An dieser Stelle haben die leichtesten Bäume ein Gewicht von 2, von denen wir vier haben, nämlich die beiden soeben erstellten Bäume mit `p` und `r` und mit `t` und `u`, und die beiden Ein-Blatt-Bäume mit `h` und `s`.

Erstellung eines Huffman-Codes [Slide 8]

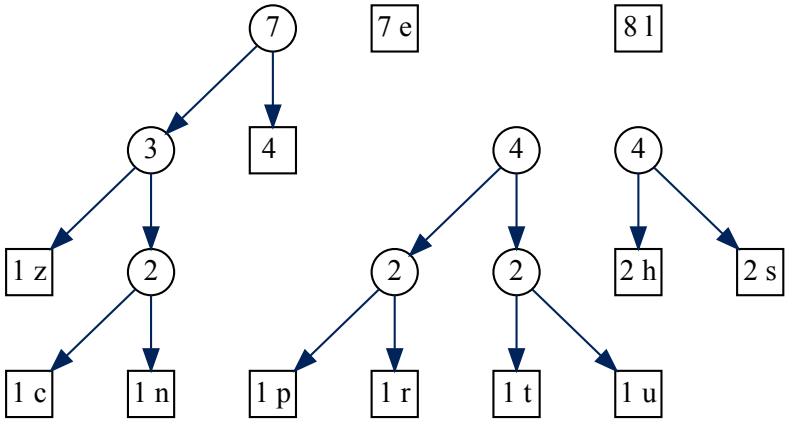


Erklärung

Hier kombinieren wir die beiden erstgenannten und auch die beiden letztgenannten jeweils zu einem neuen Baum mit Gewicht 4.

Diese Prozedur setzen wir fort, bis wir sämtliche Bäume zu einem einzigen Baum kombiniert haben, dessen Gewicht die Gesamtanzahl der Zeichen unseres Textes ist.

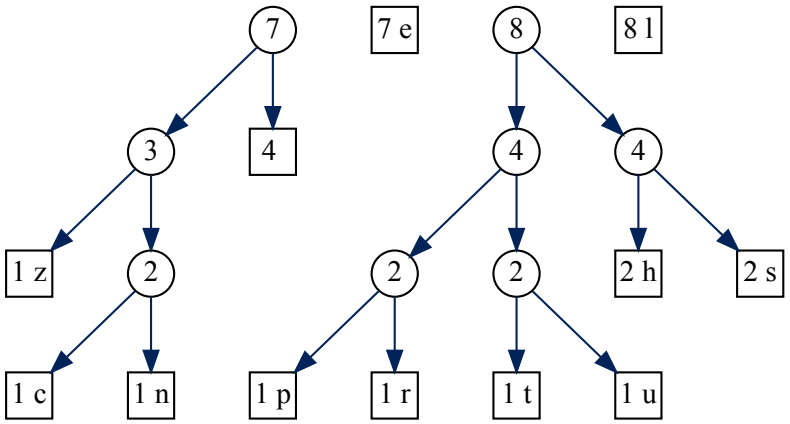
Erstellung eines Huffman-Codes [Slide 9]



Erklärung

Wir kombinieren den Baum mit Gewicht 3 mit dem Blatt mit dem Leerzeichen zu einem Baum mit Gewicht 7,

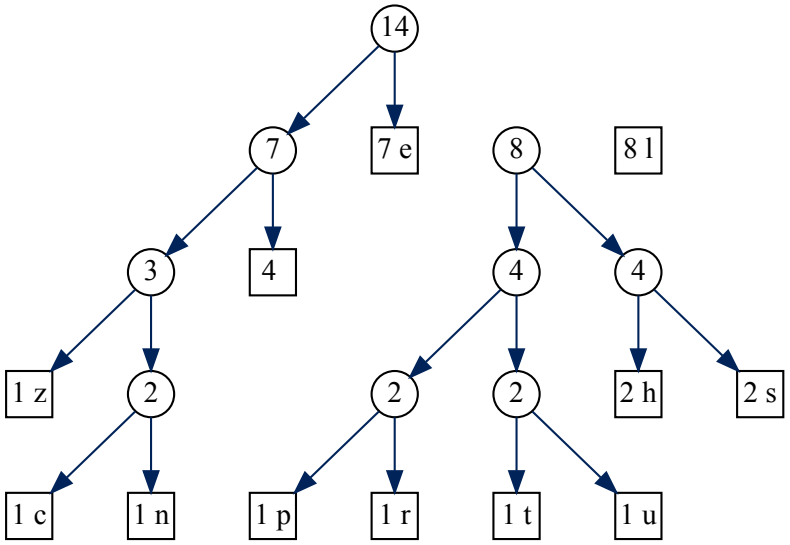
Erstellung eines Huffman-Codes [Slide 10]



Erklärung

und anschließend die beiden Bäume mit Gewicht 4 zu einem Baum mit Gewicht 8.

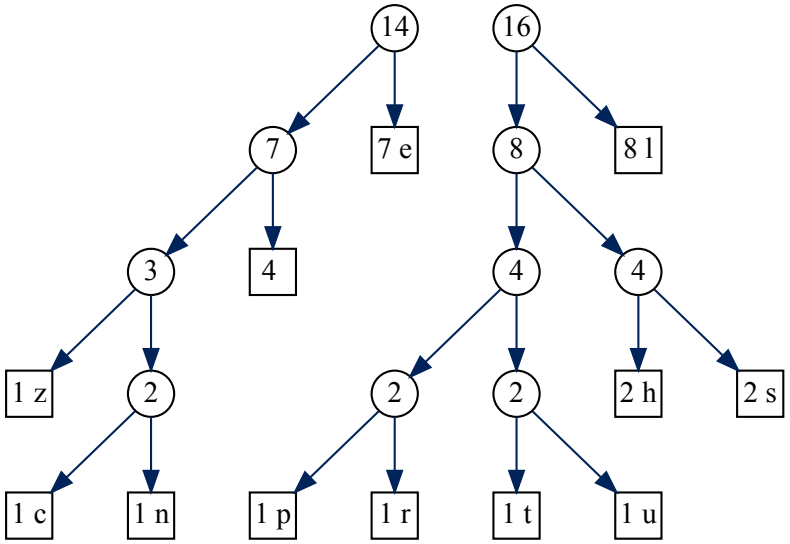
Erstellung eines Huffman-Codes [Slide 11]



Erklärung

Dann kombinieren wir die beiden Bäume mit Gewicht 7 zu einem Baum mit Gewicht 14,

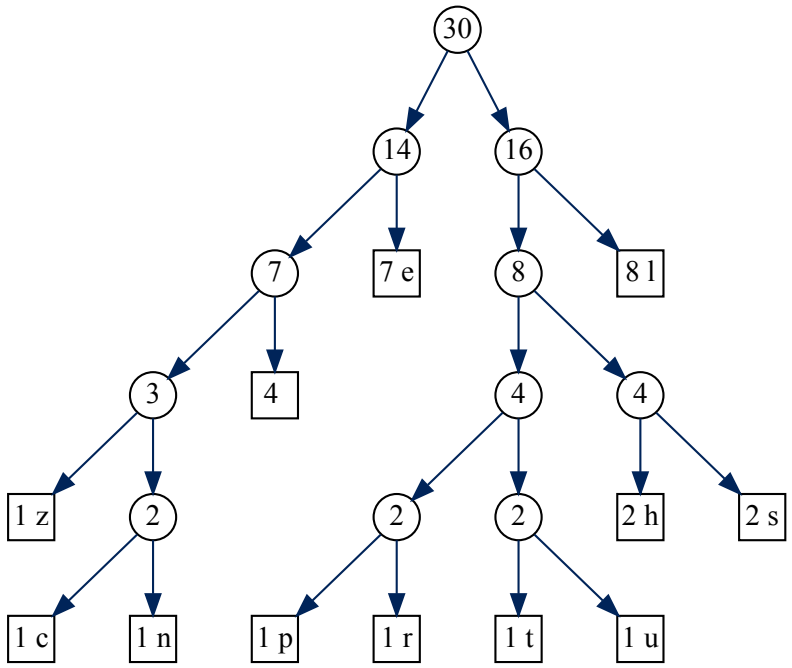
Erstellung eines Huffman-Codes [Slide 12]



Erklärung

und die beiden Bäume mit Gewicht 8 zu einem Baum mit Gewicht 16.

Erstellung eines Huffman-Codes [Slide 13]



Erklärung

Zu guter Letzt kombinieren wir diese beiden verbleibenden Bäume zum fertigen Huffman-Baum mit Gewicht 30, was genau der Gesamtanzahl der Zeichen unseres Texts entspricht.

3 *Huffman Coding*: Algorithmus und Analyse

Video 3 beginnt hier.

Algorithmus [Slide 14]

Algorithm Huffman(X):
  Require: String X of length n with d distinct characters.
  Ensure: T is a Huffman coding tree for X.
  Count the frequency $f(c)$ of each character c of X.
  Initialize a priority queue Q.
  foreach unique c in X **do**
    Create a single-node binary tree T storing c.
    Q.insert($f(c), T$)
  while Q.size() > 1 **do**
    $(f_1, T_1) \leftarrow Q$.removeMin()
    $(f_2, T_2) \leftarrow Q$.removeMin()
    Create a new binary tree T with left and right subtrees T_1 and T_2.
    Q.insert($f_1 + f_2, T$)
  $(f, T) \leftarrow Q$.removeMin()
  return T

Erklärung

Hier sehen Sie diesen Algorithmus in Pseudocode hinreichend detailliert ausformuliert, um ihn zu analysieren. Der wichtigste Aspekt ist die Auswahl der Bäume, die wir als

nächstes zusammenfassen werden. Da es sich hier immer um die Bäume geringsten Gewichts handelt, verwenden wir eine Vorrangwarteschlange. In diese werden wir alle Bäume einfügen, mit ihrem Gewicht als Schlüssel.

Damit ist dieser Algorithmus frei von Überraschungen. Zuerst zählt er die Vorkommen der verschiedenen Zeichen der zu codierenden Zeichenkette X. Anschließend wird für jedes der verschiedenen Zeichen ein Ein-Blatt-Baum erzeugt und mit seiner Häufigkeit als Schlüssel in die Vorrangwarteschlange eingefügt.

In der `while`-Schleife werden in jeder Iteration die zwei Bäume *minimalen* Gewichts aus der Vorrangwarteschlange geholt und zu einem neuen Baum kombiniert. Dieser wird dann mit der Summe der Gewichte der beiden Unterbäume als Schlüssel in die Vorrangwarteschlange eingefügt. Die Schleife endet, wenn die Vorrangwarteschlange nur noch einen einzigen Baum enthält, der dann unseren fertigen Huffman-Code repräsentiert.

Die Hauptaktion des Huffman-Algorithmus besteht in der ***gierigen*** Auswahl der beiden Bäume, die als nächstes kombiniert werden, mittels `removeMin()`. Damit ist er ein typischer ***gieriger*** Algorithmus.

Was ist die Laufzeit des Huffman-Algorithmus? Er besteht aus einer Sequenz von 6 mehr oder weniger detailliert beschriebenen Anweisungen:

1. dem Zählen der Zeichenhäufigkeiten,
2. der Initialisierung der Vorrangwarteschlange,
3. der `foreach`-Schleife,
4. der `while`-Schleife,
5. dem letzten `removeMin()`, und
6. der `return`-Anweisung.

Seine asymptotische Gesamtlaufzeit ist daher die Summe, also das Maximum der asymptotischen Laufzeiten dieser 6 Anweisungen.

Die erste Anweisung, das Zählen der Zeichenhäufigkeiten, können wir implementieren, indem wir in einer Schleife die n Zeichen des Texts durchgehen, und in einer Hash-Tabelle über deren Häufigkeiten Buch führen. Dies erfordert pro Zeichen ein `get()` und ein `put()`, welche beide in konstanter erwarteter Laufzeit implementierbar sind. Damit ist die asymptotische Laufzeit des Zeichen-Zählens $\Theta(n)$.

Die Initialisierung der leeren Vorrangwarteschlange Q nimmt lediglich konstante Laufzeit in Anspruch.

Die `foreach`-Schleife iteriert über die *verschiedenen* Zeichen in X. Um diese zu erhalten, iterieren wir über unserer soeben erstellten Hash-Tabelle und nicht über X. Damit iteriert diese Schleife lediglich $\Theta(d)$ Mal bei d verschiedenen Zeichen, und nicht $\Theta(n)$ Mal.

Jede Iteration der `foreach`-Schleife erstellt in konstanter Zeit einen trivialen Baum und fügt diesen in die Vorrangwarteschlange ein. Da die Vorrangwarteschlange mit jeder Iteration wächst, ist die Gesamtlaufzeit dieser Schleife $O(d \log d)$. Mittels *Bottom-Up Heap Construction* könnten wir sie auf $\Theta(d)$ begrenzen.

Die `while`-Schleife beginnt mit einer Vorrangwarteschlange, die d Elemente enthält, und reduziert diese in jeder Iteration um ein Element. Sie iteriert also $\Theta(d)$ Mal.

In jeder Iteration ruft sie zweimal `removeMin()` und einmal `insert()` auf, jeweils mit einer asymptotischen Laufzeit von $O(\log d)$ bei Verwendung eines Heaps. Alle anderen Operationen innerhalb der `while`-Schleife sind in konstanter Laufzeit implementierbar. Damit ist die Gesamtlaufzeit der `while`-Schleife $O(d \log d)$.

Die asymptotischen Laufzeiten der beiden letzten Anweisungen `removeMin()` und `return` sind geringer.

Damit ist die Gesamtlaufzeit des Huffman-Algorithmus $O(n + d \log d)$, dominiert durch das Zeichen-Zählen und die `while`-Schleife. Diesen Ausdruck können wir nicht weiter verlustfrei vereinfachen, da je nach Zusammensetzung der Zeichenkette $d \log d$ größer

oder kleiner sein kann als n. Immerhin können wir als obere Schranke $O(n \log n)$ angeben, denn im schlimmsten Fall sind alle n Zeichen verschieden, also $d = n$.

Analyse [Slide 15]

Laufzeit: $O(n + d \log d)$

Korrektheit: Unter allen Codierungen, die alle Zeichen eines gegebenen Texts *voneinander unabhängig* codieren, liefert der Huffman-Algorithmus einen kürzesten Code.

(Beweis per Induktion über d)

Anmerkung

Es handelt sich um einen ***gierigen*** *(greedy)* Algorithmus: Bei jedem Schritt wählen wir die Möglichkeit *(den kürzesten Code)*, die uns unmittelbar *(nur mit Blick auf dieses eine Zeichen)* am weitesten bringt.

Erklärung

Der Huffman-Algorithmus ist *korrekt* in dem Sinne, dass er für einen gegebenen Text garantiert einen kürzest möglichen Code findet, der alle Zeichen des Texts *unabhängig voneinander* codiert.

Man kann das durch Induktion über der Anzahl d der verschiedenen Zeichen beweisen. Die Details sparen wir uns hier.

Es gibt i.A. noch kürzere Codierungen, die ausnutzen, dass manche Zeichen*sequenzen* häufiger auftreten als andere. Solche Algorithmen werden wir hier aber nicht besprechen.

4 Paradigma

Algorithmisches Paradigma: *Gierige* Algorithmen *(Greedy Algorithms)* [Slide 16]

Gierige Algorithmen sind auf *Optimierungsprobleme* (die eine *Konfiguration* suchen, die eine *Zielfunktion* optimiert) anwendbar, die die folgende Eigenschaft besitzen:

Optimalität gieriger Entscheidungen: Beginnend an einem geeigneten Ausgangspunkt, führt eine Sequenz ***lokal*** *optimaler Entscheidungen* zu einer ***global*** *optimalen Lösung.*

Definition

Das gierige Paradigma ist allgemein auf Optimierungsprobleme anwendbar, die eine Konfiguration suchen, die eine Zielfunktion optimiert. Im Falle der Münzrückgabe besteht die gesuchte Konfiguration in einer Menge Münzen, deren Gesamtwert dem Rückgeld entspricht, und die Zielfunktion ist die Größe dieser Menge. Beim Huffman-Coding besteht die gesuchte Konfiguration in den Zeichencodes, und die Zielfunktion ist die Länge der Bitsequenz, die den Originaltext codiert.

Die Lösungen, die ein gieriger Algorithmus findet, sind jedoch nur optimal, falls das Problem die Eigenschaft der ***Optimalität gieriger Entscheidungen*** besitzt: Diese Eigenschaft besagt, dass eine Sequenz *lokal* optimaler Entscheidungen tatsächlich zu einer *global* optimalen Lösung führt.

Sowohl unser originales Münzrückgabe-Problem als auch das Huffman-Coding besitzen diese Eigenschaft. Das um die 80-Cent-Münze erweiterte Münzrückgabe-Problem besitzt sie jedoch nicht. Dies hindert uns zwar nicht an der Anwendung unseres gierigen Algorithmus, aber die Lösungen, die er findet, sind nicht mehr unbedingt optimal. Eine Sequenz

von Münzen, die den Restgeldbetrag bei jeder Iteration jeweils *maximal* reduzieren, führt oft zu einer *suboptimalen* Gesamtlösung.

5 Das fraktionale Rucksack-Problem

Video 4 beginnt hier.

Das fraktionale Rucksack-Problem [Slide 17]

Problem: Wir haben eine Menge S mit N teilbaren Objekten. Jedes Objekt i besitzt einen *Wert* $b_i > 0$ und ein *Gewicht* $w_i > 0$. Wir suchen Teilobjekte x_i aus S mit maximalem Gesamtwert, deren Gesamtgewicht W nicht überschreitet:

$$\max_{0 \leq x_i \leq w_i} \sum_{i \in S} b_i \frac{x_i}{w_i} \text{mit } \sum_{i \in S} x_i \leq W$$

- Beschreibe einen gierigen Algorithmus!
- Charakterisiere die Lösung!

Erklärung

Betrachten wir zur Illustration gieriger Algorithmen ein weiteres, klassisches Problem, bekannt als das fraktionale Rucksack-Problem. Nehmen wir an, Sie packen für eine mehrtägige Bergtour Ihren Rucksack. Sie haben $W = 3000\,\text{g}$ Kapazität für Verpflegung, und möchten diese optimal nutzen. Sie haben zu Hause kleinere Mengen verschiedener Nahrungsmittel vorrätig, nämlich etwas Brot, Bohnenaufstrich und Käse. Wie füllen Sie Ihren Rucksack mit maximalem Nährwert?

Zunächst machen Sie eine Bestandsaufnahme Ihrer Nahrungsmittel-Vorräte und ihres Nährwerts. Sie haben einen Laib Brot, der einen Nährwert von $b_1\,\text{kJ}$ besitzt und $w_1\,\text{g}$ wiegt. Ihr Bohnenaufstrich hat einen Nährwert von $b_2\,\text{kJ}$ und wiegt $w_2\,\text{g}$, und Ihr Käse hat $b_3\,\text{kJ}$ und wiegt $w_3\,\text{g}$,

Unser auf den ersten Blick etwas kompliziert anmutendes Optimierungsproblem besagt nun, dass wir eine Kombination von Brot-, Bohnenaufstrich- und Käse-Anteilen mit maximalem Gesamt-Nährwert suchen, deren Gesamtgewicht die 3-Kilo-Kapazitätsgrenze nicht überschreitet.

Wie finden wir eine solche Kombination?

Ganz einfach: Sie beginnen mit dem Nahrungsmittel mit dem höchsten Nährwert pro Gewichtseinheit, also wohl dem Käse. Wiegt Ihr Käse mehr als 3 kg, schneiden Sie exakt 3 kg ab, packen das Stück in den Rucksack, und sind fertig. Andernfalls packen Sie den gesamten Käse in den Rucksack und fahren für die verbleibende Kapazität mit dem nächst-nahrhaften Nahrungsmittel auf dieselbe Weise fort.

Dieser Algorithmus ist *gierig*, denn Sie treffen eine Sequenz lokal optimaler Entscheidungen: Bei jeder Iteration erhöhen Sie den Nährwert im Rucksack so weit wie möglich.

Optimalität gieriger Entscheidungen [Slide 18]

Proposition: Das fraktionale Rucksackproblem besitzt die Eigenschaft der Optimalität gieriger Entscheidungen.

Beweis (durch Widerspruch): Nehmen wir an, eine optimale Gesamtlösung enthalte zwei Teilobjekte i und j mit

$$x_i < w_i, x_j > 0, \frac{b_i}{w_i} > \frac{b_j}{w_j}.$$

Wir können nun die Menge $y = \min\{w_i - x_i, x_j\}$ von j durch eine gleiche Menge von i ersetzen, und so den Gesamtwert erhöhen, ohne das Gesamtgewicht zu ändern.

Erklärung

Führt diese Sequenz lokal optimaler Entscheidungen zwangsläufig zu einem mit maximalem Nährwert ausgelasteten Rucksack?

Nehmen wir einmal an, die Lösung, die unser gieriger Algorithmus findet, sei nicht optimal. Folglich muss es möglich sein, den Nährwert im Rucksack zu erhöhen, indem ich einige Gramm eines Nahrungsmittels j durch dieselbe Menge eines anderen Nahrungsmittels i ersetze. Dies ist nur möglich, falls drei Bedingungen erfüllt sind:

1. habe ich noch etwas vom Nahrungsmittel i übrig, also $x_i < w_i$.
2. steckt vom Nahrungsmittel j eine Menge $x_j > 0$ im Rucksack.
3. ist der spezifische Nährwert von Nahrungsmittel i größer als der von Nahrungsmittel j, also $b_i/w_i > b_j/w_j$.

Es ist jedoch unmöglich, dass diese drei Bedingungen erfüllt sind, denn in diesem Fall würde unser Algorithmus Nahrungsmittel j erst anschneiden, nachdem Nahrungsmittel i komplett in den Rucksack gepackt wurde.

Damit ist bewiesen, dass die Sequenz gieriger Entscheidungen unseres Algorithmus zu einer global optimalen Gesamtlösung des fraktionalen Rucksackproblems führt.

Algorithmus [Slide 19]

Laufzeit?

$O(N \log N)$, $\Omega(N)$

```
Algorithm fractionalKnapsack(S, W):
  Require: Set S of items with positive benefits b_i and weights w_i;
           positive maximum total weight W.
  Ensure: Amounts x_i maximize the total benefit
          while not exceeding the total weight limit W.
  for each item i ∈ S do
    x_i ← 0
    v_i ← b_i/w_i                    // "value index" of item i
  Q ← HeapPriorityQueue({−v_i}, {i}) // bottom−up construction
  w ← 0                              // total weight
  while w < W do
    i ← Q.removeMin()                // greedy choice
    a ← min{w_i, W − w}              // avoid weight overflow
    x_i ← a
    w ← w + a
```

Definition

Hier sehen wir eine präzise Formulierung unseres gierigen Algorithmus zur Lösung des fraktionalen Rucksack-Problems.

Zunächst berechnen wir in der `for`-Schleife die spezifischen Werte v_i der N teilbaren Objekte i.

Anschließend initialisieren wir eine Heap-basierte Vorrangwarteschlange der N Objekte mit den negativen v_i als Schlüsseln, und initialisieren das bisher beladene Gesamtgewicht mit $w = 0$.

In der `while`-Schleife holen wir uns das Objekt mit dem größten spezifischen Wert v_i aus der Vorrangwarteschlange, und packen so viel davon wie möglich in den Rucksack.

Erklärung

Die asymptotische Laufzeit dieses Algorithmus ist einfach zu ermitteln: Die Anweisungen vor der `while`-Schleife haben eine Gesamtlaufzeit von $\Theta(N)$. Die `while`-Schleife iteriert $O(N)$ Mal im schlechtesten Fall, und ein einziges Mal im besten Fall. Jede Iteration hat wegen des `removeMin()` eine Laufzeit von $O(\log N)$.

Damit ist die Gesamtlaufzeit von `fractionalKnapsack()` $O(N \log N)$ wegen der `while`-Schleife, und $\Omega(N)$ wegen der `for`-Schleife und der *Bottom-Up Heap Construction*.

Das (nicht fraktionale) Rucksackproblem [Slide 20]

Hier können wir Objekte entweder ganz oder gar nicht einpacken.

Dies ist ein klassisches, rechnerisch sehr komplexes Optimierungsproblem und äquivalent zur Münzrückgabe.

Erklärung

Unser kleines, durch einen gierigen Algorithmus leicht lösbares fraktionales Rucksack-Problem hat übrigens einen berühmt-berüchtigten großen Bruder, nämlich das nicht-fraktionale Rucksackproblem oder gemeinhin einfach *das Rucksackproblem*. Bei *dem* Rucksackproblem sind die einzelnen Objekte nicht teilbar, sondern können nur entweder ganz oder gar nicht eingepackt werden.

Dieses Problem ist übrigens äquivalent zum allgemeinen Münzrückgabeproblem, und ist erheblich aufwändiger zu lösen.

6 Literatur

Literatur [Slide 21]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.