ADS – Vorlesung, SS06
Prof. Dr. Wolfram Conen
Mai 2006

Inhalte:

- Greedy-Algorithmen
- Matroide
- Greedy-Beispiel Huffman-Codierung

Greedy-Optimierung

- Nehmen Sie an, sie brechen nachts in eine Villa ein und wollen
  - soviel an Werten mitnehmen, wie sie tragen können
  - so schnell wie möglich wieder raus
- Es gelten folgende Nebenbedingungen
  - Die Bewohner sind verreist, wir haben im Prinzip „reichlich Zeit“
  - Wir kennen den Wert aller Wertgegenstände
  - Wir können nicht alles mitnehmen, nur ein bestimmtes Gewicht (sonst wäre „Alles mitnehmen“ optimal)
  - Die Gewichte kennen wir auch (Handwaage ist immer dabei!)

Greedy-Optimierung

- Dieses Problem ist (in krimineller Verkleidung) das klassische Knapsack Problem (Rucksack-Problem), formal:
  - Gegeben ist eine Menge M = {1,..,n} von Gegenständen mit den Werten v_i und den Gewichten w_i. Gesucht ist M‘ $\subseteq$ M mit

    $$\sum_{i \in M'} w_i \leq w_{max}$$

    d.h. die Summe der Gewichte der Gegenstände in M‘ überschreitet ein vorgegebenes Maximalgewicht nicht.
  - Der Wert der Gegenstände soll so groß wie möglich sein:

    $$Max_{M' \subseteq M} \leftarrow \sum_{i \in M'} v_i$$

Greedy-Optimierung

- Lösungsidee:
  - Gegenstände nach dem Verhältnis von Wert zu Gewicht („Wert pro Kilo“) absteigend sortieren (der mit dem größten Wert pro Gewichtseinheit steht also vorne), z.B. in einer PQueue
  - Dann nehmen wir solange Gegenstände aus der PQueue, wie wir sie noch tragen können
- Diese Strategie ist „gierig“! (English: Greedy) – man nimmt das „Vielversprechendste“ zuerst usw.
- Ist das immer sinnvoll?

Greedy-Optimierung

- Natürlich nicht (würde ich sonst so fragen...)
- Wir können maximal 50kg tragen, es gibt die folgenden Gegenstände
  - Lehrbuch über Algorithmen und Datenstrukturen,
    Gewicht w_1 = 10kg, Wert v_1 = 60 Euro, d.h. 6 Euro pro kg
  - Taschenrechner mit eingebautem Dijkstra,
    Gewicht w_2 = 20kg, Wert v_2 = 100 Euro, d.h. 5 Euro pro kg
  - Allegorische Statue, die die ewige Schönheit von (bottom-up) Heapsort symbolisiert,
    Gewicht w_3 = 30kg, Wert v_3 = 120 Euro, d.h. 4 Euro pro kg
- Was liefert die Greedy-Strategie als Resultat?
- Wie sieht das optimale Resultat aus?

Greedy-Optimierung

- Aber manchmal geht es auch...sogar immer, wenn die Problemstellung entsprechend ist.
- Andere Gegenstände:
  - Ein Haufen Schnipsel mit Klausurlösungen, Gewicht 10kg, Wert 60 Euro, 6 Euro pro kg
  - Ein Haufen Goldstaub, Gewicht 20kg, Wert 100 Euro, 5 Euro pro kg
  - Ein Haufen Silberstaub, Gewicht 30kg, Wert 120 Euro, 4 Euro pro kg
- Jetzt können wir die Gegenstände beliebig teilen:
- Also nehmen wir die Klausurschnipsel, den Goldstaub, und füllen unseren Rucksack mit Silberstaub auf (zu einem Gesamtwert von 60+100+80 = 240 Euro)
- Besser geht es natürlich nicht!

Greedy-Optimierung

- Probleme des ersten Typs (mit unteilbaren Gegenständen) heißen

  0-1 Knapsack Probleme

- Probleme des zweiten Typs (mit beliebig teilbaren Gegenständen) heißen

  Fractional Knapsack

- Fractional Knapsack Probleme lassen sich immer „greedy“ lösen!
- 0-1 Knapsacks nur ab und an „zufällig“!

Greedy-Optimierung

- Unser „kürzeste-Wege-Problem“ lässt sich auch „Greedy“ lösen – und genau das tut der Dijkstra auch!
- Das gleiche gilt für das „Minimum Spanning Tree“-Problem, und auch Kruskal bzw. Prim lösen das Problem „greedy“
- Ein guter Hinweis darauf ist oft die Verwendung einer PQueue…

Arbeitsweise von Greedy-Algorithmen

- Wir haben folgendes zur Verfügung
  - Eine Menge von *Kandidaten* C, aus denen wir die Lösung konstruieren wollen (z.B. Kanten beim MST)
  - Eine Teilmenge S $\subseteq$ C, die bereits ausgewählt wurde
  - Boole'sche Funktion *solution*, die sagt, ob eine Menge von Kandidaten eine legale Lösung des Problems darstellt (unabhängig davon, ob die Lösung optimal ist)
  - Eine Testfunktion *feasible*, die sagt, ob eine Teillösung unter Umständen zu einer kompletten legalen Lösung erweitert werden kann
  - Eine Auswahlfunktion *select*, die den nächsten „vielversprechendsten" Kandidaten liefert
  - Eine Zielfunktion *value*, die uns den Wert einer Lösung angibt

Arbeitsweise von Greedy-Algorithmen

```
Function greedy(c)
    S ← ∅
    while not solution(S) and C ≠ ∅ do
                x ← select(C)
                C ← C - {x}
                if feasible(S ∪ {x}) then
                        S ← S ∪ {x}
    if solution(S) then
                return S
    else return „There_is_no_solution“
```

Mit der Funktion *value*(S) kann man am Ende den Wert der gefundenen Lösung bestimmen (falls es eine gab...)

Welche Probleme sind „greedy“ lösbar?

- Wenn das Problem sich als Matroid modellieren läßt, dann kann man es „Greedy“ lösen!
- Aber was ist ein Matroid? [s. Übung]
- Es gilt sogar: ein Problem lässt sich **genau dann** „greedy“ lösen (allgemein für jede Gewichtungsfunktion der Elemente in die positiven reellen Zahlen), **wenn** es eine Matroid-Struktur aufweist (s. auch „Das Geheimnis des kürzesten Weges“, Literaturhinweis zu Gin1b)

Noch ein wichtiges Problem, das man „greedy“ angehen kann: Datenkompression

- Ist ihre Festplatte ständig zu klein?
- ...oder ihre Internet-Anbindung zu langsam?
- Dann ist Datenkompression ein Thema für Sie!
- Ziele:
  - möglichst platzsparende Datenspeicherung bzw. Übertragung
  - entpacken möglich, ohne Fehler in den Daten zu hinterlassen

Datenkompression

- Zwei Teilprozesse:
  - **Kompression**: Ein Prozess, der Daten in einer komprimierte, also „kleinere“, Form überführt
  - **Expansion**: Ein Prozess, der aus der komprimierten Form die Ausgangsdaten rekonstruiert
- Beispielfälle für „Original“-Daten:
  - Ein Text aus 256000 Zeichen, jedes der 256 möglichen (ASCII-)Zeichen tritt genau 1000-mal auf
  - Der Text besteht aus 256000-mal dem Zeichen ‚a‘
- Beide Texte nehmen 256000*8 = 2.048.000 Bit Plattenplatz ein.

Datenkompression

- Betrachten wir einmal die Wahrscheinlichkeit, dass an einer bestimmten Stelle der Files ein bestimmtes Zeichen auftaucht:
  - Im ersten File ist die Wahrscheinlichkeit für das Auftauchen jedes Zeichens an der ersten betrachteten Position gleich, nämlich 1/256, z.B. für ‚a‘
    - An später betrachteten Positionen verschieben sich die Wahrscheinlichkeiten abhängig von den vorher bereits betrachteten Zeichen (wenn es z.B. gar kein a mehr gibt), aber „ungefähr“ bleibt es bei der Wahrscheinlichkeit 1/256 auch an den anderen Positionen
  - Im zweiten Fall ist die Wahrscheinlichkeit für das Auftauchen von ‚a‘ an jeder Position 1.

Datenkompression

- Im ersten Fall besteht eine **hohe Unsicherheit** darüber, welches Zeichen an der betrachteten Position auftritt.
- Um zwischen den verschiedenen möglichen „Ereignissen“ (das Auftreten eines bestimmten der 256 Zeichen) zu unterscheiden, müssen wir den aufgetretenen Fall genau angeben
- Um 256 Ereignisse zu unterscheiden, brauchen wir ($\log_2$ 256) Bit (also 8 ;-)

Datenkompression

- Im zweiten Fall wissen wir **mit Sicherheit**, welches Zeichen an der betrachteten Position auftritt (nämlich „a“)
- Um zwischen den verschiedenen möglichen „Ereignissen“ zu unterscheiden, brauchen wir **gar keine Information**
- Wir müssen nur wissen, wieviele „a“ auftreten
- Insgesamt können wir das File durch „Jetzt kommen 256.000 ‚a‘“ vollständig beschreiben (also mit ungefähr 150 Bit)
- Intuitiv ist klar, dass man im ersten Fall nicht sehr viel komprimieren kann, im zweiten aber schon!

Datenkompression

- Zum Komprimieren muss man den „Eingabetext“ sinnvoll kodieren.
- Elementare Idee:
  Zeichen, die häufig vorkommen, erhalten vergleichsweise kurze Codes
- Das nennt man „*variable Kodierung*“
- Sie soll den folgenden Ausdruck minimieren

$$\sum l(c_i) * f(c_i), \quad 1 \leq i \leq n$$

  - n = Anzahl der Zeichen
  - $l(c_i)$ = Länge der Codierung des Zeichens c_i
  - $f(c_i)$ = Häufigkeit von c_i im Text

Datenkompression

	a	b	c	d	e	f
Häufigkeit	45	13	12	16	9	5
ASCII	01100001	01100010	01100011	01100100	01100101	01100110
Code 1	0	011	100	101	0011	1100
Code 2	0	101	100	111	1101	1100

Was ist schlecht am Code 1? Probieren Sie mal, 001100101 zu expandieren!

Man könnte natürlich die Länge des nächsten Codes abspeichern, aber so recht macht das keinen Sinn...bei Code 2 geht das auch so!

Datenkompression

- Definition: Eine Codierung heißt **präfix-frei**, wenn kein Code Anfangsstück (=Präfix) eines anderen Codes ist
- Einen solchen Code kann man mit Hilfe von binären Entscheidungsbäumen finden:
  - Die Blätter sind die zu kodierenden Zeichen
  - Die Codes ergeben sich aus den Wegen im Baum, die von der Wurzel zu den Zeichen führen
  - Eine Zweig nach links steht für ein 0, ein Zweig nach rechts für eine 1 [s. Mitschrieb]
  - Eine solche Kodierung ist immer präfix-frei (warum?)

Datenkompression

- Unsere „neue“ Aufgabe ist also:
  - Gegeben ist eine Datei mit zu komprimierenden Daten
  - Man konstruiere einen binären Entscheidungsbaum, der zu einer optimalen präfix-freien Codierung des Dateiinhalts führt.
- Eine sehr bekannte Lösung dieser Aufgabe ist die so genannte *Huffman-Codierung*

Huffman-Codierung

- Baum zur Kodierung auf Basis der Zeichenhäufigkeit optimal aufbauen
- Infos über die gewählte Codierung in der erzeugten Datei mit der Komprimierung speichern
- Führt je nach Daten zu Komprimierungen ca. zwischen 20%-70% (im worst-case spart man nix...im Gegenteil, die Codetabelle kostet ja auch etwas)

Huffman-Codierung: Ablauf

1. Ein Durchlauf zur Bestimmung der vorkommenden Zeichen und zur Ermittlung ihrer Vorkommenshäufigkeit
2. Aufbau des optimalen Codebaums
3. Ableiten der Codes und Codelängen aus dem Baum
4. Abspeichern der Codeinformationen in der Ausgabedatei
5. Zweiter Durchlauf, um die Zeichen zu codieren, nebst Ablage in der Ausgabedatei

Huffman-Codierung: Ablauf

- Häufigkeitsermittlung ist klar
- Aufbau des Codebaums:
  - „Gierige“ Suche nach einem optimalen Baum (Gierig, weil die Zeichen in der Reihenfolge „absteigende Häufigkeit“ genau einmal angepackt werden)

Huffman-Codierung: Baumaufbau

STUDENTEN SCHLAFEN NIE (na ja, ungefähr...)

Zeichen	blank ‚_‘	D	E	F	N	O	P	S	T	U
Häufigkeit	2	1	4	1	5	1	1	1	3	1

- Erstes Ziel: die vorkommenden Zeichen in zwei Gruppen zerlegen, die möglichst gleich häufig sind
  - Warum? Mit einem Bit können sie perfekt eine „Schwarz-Weiss“-Entscheidung wiedergeben.
  - Wenn die beiden Ereignisse „Schwarz“ und „Weiss“ gleichwahrscheinlich sind, dann brauchen sie tatsächlich ein ganzes Bit zu ihrer Unterscheidung
  - Wenn sie ungleich verteilt sind, dann kämen sie „auf lange Sicht“ auch mit weniger aus (warum?)
  - Wir möchten den möglichen Informationsgehalt in einem Bit optimal ausschöpfen (und nichts verschenken!)

Huffman-Codierung: Baumaufbau

Der Text ist 20 Zeichen lang, wir haben 10 verschiedene Zeichen.

Gruppe 1	E D F O P	8
Gruppe 2	N T ,_, S U	12

- Das führt zur ersten Ebene des „Konstruktions“-Baumes (s. Mitschrieb)
- Zerlegt man die Gruppe 1 weiter, erhält man bereits ein einzelnes Zeichen als Blatt (s. Mitschrieb)
- Insgesamt ergibt sich der Ergebnisbaum des Mitschriebs

Huffman-Codierung: Codierung

Das führt zu folgender Codierung (Codetabelle):

Zeichen	blank ,_,	D	E	F	N	O	P	S	T	U
Häufigkeit	2	1	4	1	5	1	1	1	3	1
Code	1110	0100	00	0101	10	0110	0111	11110	110	11111
Länge	4	4	2	4	2	4	4	5	3	5

- Unsere Eingabe war: STUDENTEN PENNEN OFT (sorry!)
- Wie sieht die Kodierung aus?
- 111101101111101000010110001011100111001010001011100110010 1101110
- Diese Bitfolge kann man natürlich bei gegebener Codetabelle in eindeutiger Weise wieder expandieren – probieren sie es!

Huffman-Codierung: Implementierung

- Implementieren kann man das leichter „von unten“:
  1. Starten mit den Blättern (eines je vorkommendem Zeichen) und ihren Häufigkeiten
  2. Suchen der beiden Blätter mit den niedrigsten Häufigkeiten, z.B. S und U
  3. Konstruktion eines Knoten, dessen linker Nachfolger der eine (hier: S) und dessen rechter Nachfolger der andere Knoten (hier: U) ist, die zugehörige Häufigkeit ergibt sich als Summe der Häufigkeiten der Kindknoten
  4. Entfernen der beiden „alten“ Knoten aus der Menge „vaterloser“ Knoten, fügen den neuen Knotens hinzu
  5. Wiederhole 2-4, bis nur noch ein Knoten übrig ist

Huffman-Codierung: Implementierung

- Die Implementierung mit einer PQueue ist „straightforward“ (Strickmuster: 2 raus, einen rein)
- Aus dem entstandenen Baum läßt sich die Codetabelle, wie bereits beschrieben, unmittelbar generieren (wie?)
- Komplexität des PQueue-Handlings ist für k verschiedene Zeichen wie gehabt O(k log k)
  - Initial k Knoten einfügen
  - dann jeweils 2 entnehmen und einen Hinzufügen, insgesamt also n-1 neue Inserts (mit steigenden Häufigkeiten)
- Für „längere“ Texte mit n Zeichen(vorkommen), n >> k, dominiert der (zu n lineare) Aufwand für das Einsammeln der Häufigkeiten

Huffman-Codierung: Expansion

- Die Codetabelle wird in der Ausgabedatei abgespeichert
- Aus der Codetabelle kann man direkt den Baum rekonstruieren
  - 0 = links, 1 = rechts, die Tiefe eines Zeichens entspricht der Länge des Codes für ein Zeichen
- Die Expansion läßt dann den binären „Codestring“ durch den Baum rieseln:
  1. Beginn mit dem ersten Zeichen des Codestrings
  2. Wegwahl entsprechend der binären Ziffern entlang des Baumes, beginnend mit der Wurzel, bis ein Blatt erreicht wird
  3. Zum Blatt gehöriges Zeichen ausgeben und auf die Wurzel zurückgehen, wenn noch nicht alles expandiert ist, zum Schritt 2 zurückkehren

Literatur

- Zum generellen Greedy-Ablauf und zur Huffman-Codierung:
  - B. Owsnicki-Klewe: *Algorithmen und Datenstrukturen*, 4. Auflage, Wißner-Verlag, Augsburg, 2002 (gut lesbares Buch, launig-nett geschrieben, kann beim Verstehen sicher helfen, kaum/keine Beweise, wenig Aufgaben, keine Lösungen, aber dafür nicht sehr teuer, ca. 15 Euro, und berührt viele unserer Themen)
  - Ansonsten finden sie praktisch in allen genannten Büchern Informationen zu Greedy-Algorithmen und Codierungen
- Zu Matroiden: In allen guten Büchern zu Optimierung (z.B. dem von Korte und Nguyen) oder auch in Cormen, Rivest, Leierson, Stein oder bei Schöning (s. frühere Literaturangaben).