

Kapitel 7

Algorithmenkonstruktion

- 7.1 Aufwand von Algorithmen
- 7.2 Teile und Herrsche
- 7.3 Gierige Algorithmen



7 Problem und Algorithmus

1/66

- Algorithmen lösen Abbildungsprobleme $f: A \rightarrow B$:
 - Gegeben ein Wert a aus dem Definitionsbereich A , berechne $b=f(a)$ aus dem Wertebereich B
- Klassifikation:
 - totaler Algorithmus (für totales Problem): Algorithmus funktioniert für alle $a \in A$
 - partieller Algorithmus (für partiell definiertes oder totales Problem): Algorithmus funktioniert nur für $a \in A' \subset A, A' \neq A$
- Für ein Problem P (Abbildung f) kann es keinen oder mehrere Algorithmen geben (f nicht berechenbar / berechenbar)



7 Vom Problem zum Algorithmus

2/66

- Algorithmische Probleme P löst man
 - geschlossen**: $ax+b = 0$, Lösung $x = -b/a$
 - rekursiv oder iterativ**: Berechnung eines Fixpunkts, iterative Approximation einer Lösung hinreichender Genauigkeit
 - durch Zerlegung**
 - Zerlege P in Teilprobleme A, B, C, \dots , löse diese und setze daraus die Lösung von P zusammen (**teile-und-herrsche**)
 - Löse ein elementares Teilproblem und erweitere die Lösung schrittweise zu einer Gesamtlösung (**gierige Algorithmen**)
 - Durch **Suchen** in einem Suchraum (z.B. generiere-und-teste wie beim n -Damen-Problem)
 - Es gibt **weitere Konstruktionsverfahren**, die in späteren Vorlesungen behandelt werden



7 Vergleich von Algorithmen

3/66

Wann sind zwei Algorithmen gleich?

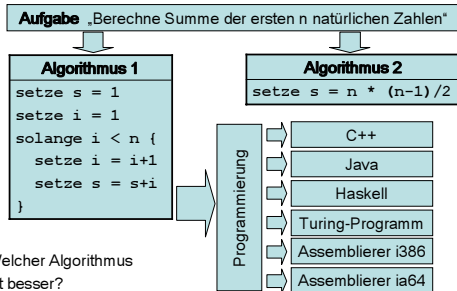
- Wenn sie in der gleichen Programmiersprache, abgesehen von der Bezeichnerwahl, gleich sind?
 - Das wäre zu viel verlangt
- Wenn sie im funktionalen Programm die gleiche Anzahl von Reduktionen, d.h. Anwendungen elementarer Regeln benötigen?
 - ungeeignet, da das gleiche Programm mit verschiedenen Implementierungen der Sprache zu verschiedenen Reduktionszahlen führt
- **Wenn sie auf geeigneter Abstraktionsebene die gleichen Rechenschritte durchführen!**
 - **Beispiel:** Sortieren durch Einfügen (Kap. 5) kann man auch von Hand mit einem Kartenstapel durchführen
 - keine Rechnerbenutzung, kein schriftliches Programm
- Wie der Algorithmus beschrieben ist: mündlich, als lesbarer Text, in irgendeiner Programmiersprache, ..., ist für den Vergleich unwichtig



7 Motivation (II)

4/66

Beispiel:



7 Aufwand von Algorithmen

5/66

- Algorithmen
- verbrauchen Rechenzeit
 - verbrauchen Speicher für Datenstrukturen

Fragestellungen zu Algorithmen:

- Ist Algorithmus A schneller / sparsamer als Algorithmus B?
- Kann ein Algorithmus überhaupt signifikant verbessert werden?
- Ist der Algorithmus unter ökonomischen Gesichtspunkten nutzbar?

Beispiele:

- Wettervorhersagen sind wegen zu geringer Rechenkapazität ungenau.
- Die Kryptographie basiert auf der Hoffnung, daß bestimmte Berechnungen nicht vertretbaren Zeitaufwand benötigen.
- Dublettenprüfung in Adreßdatenbank ist oft teurer als das mehrfach bezahlte Porto.



7 Problemumfang und Aufwand

6/66

- Bei einem Problem $f: A \rightarrow B$ kann man das Argument $a \in A$ (oder das gewünschte Ergebnis) durch einen **Umfang** n charakterisieren:
 - a ist ganze Zahl: $n =$ Anzahl Dezimal- oder Dualstellen
 - Berechnung von p : Anzahl der gewünschten Dezimal-/Dualstellen
 - Suchen nach $x \in M$: Umfang $n = |M|$ der Menge, Liste, ... M
 - Sortieren von M : Umfang $n = |M|$
 - usw.
- Der **Aufwand** $t_x(n)$ eines Algorithmus A zur Berechnung von f hängt von n ab
 - Verschiedene Algorithmen zur Berechnung von f können, bei gleicher Eingabe a , unterschiedlichen Aufwand haben
 - Der kleinste mögliche Aufwand heißt der **Aufwand für das Problem oder die Komplexität des Problems**



7.1 Aufwand: Begriffe

7/66

- Umfang** n : Anzahl der Eingabewerte (seltener: Anzahl der Bits der Eingabe oder Umfang der Ausgabe)
- Aufwand** $t(n)$: Anzahl der Zeit- bzw. Speichereinheiten, die der Algorithmus für ein Problem mit Umfang n benötigt.
- Komplexität**: geringstmöglicher Aufwand, mit dem man das gegebene Problem mit irgendeinem Algorithmus lösen kann.

Hängt der Aufwand nicht nur vom Umfang ab, sondern auch von den tatsächlichen Eingabewerten, dann interessiert ferner:

- ungünstigster Aufwand** (Aufwand im schlimmsten Fall)
- mittlerer Aufwand**, gemittelt über alle Eingaben gleichen Umfangs
- Erwartungswert des Aufwands** (nach Wahrscheinlichkeiten gewichtetes Mittel)
- Aufwand im besten Fall** (meistens uninteressant)



7.1 Messung und Vergleich des Aufwands

8/66

- Der Aufwand eines Algorithmus hängt ab von
- der Programmiersprache,
 - der Implementierung dieser Sprache (Interpreter, Übersetzer)
 - der Größe und Geschwindigkeit des Hauptspeichers
 - der Geschwindigkeit des Prozessors
 - dem Maschinenmodell (Architektur des Prozessors)
 - CISC-Prozessor (iX86), RISC-Prozessor (MIPS, PowerPC, SPARC), interpretierte Modelle (Haskell-Interpreter, virtuelle Java Maschine), theoretische Prozessormodelle (Turing-Maschine, ...)
- ⇒ Aufwandsvergleich daher bestenfalls bis auf (konstante) Proportionalitätsfaktoren möglich



7.1 Maschinenmodell RAM

9/66

Als einfachstes Modell zum Aufwandsvergleich wird allgemein das (theoretische) Maschinenmodell **Random-Access-Machine (RAM)** akzeptiert

- Eigenschaften:
 - ein Prozessor
 - Hauptspeicher unbeschränkt groß
 - Speicherzellen unbeschränkt lang (keine Beschränkung des Zahlbereichs ganzer Zahlen)
 - eine Zeiteinheit je elementarer Operation (egal welche): arithmetische oder logische Operationen, Vergleich des Inhalts von Speicherzellen, Daten aus oder in den Speicher bringen, usw.
- benötigt ein Algorithmus auf der RAM einen Aufwand $t(n) = f(n)$, so kann man annehmen, daß der tatsächliche Rechenaufwand auf einem realen Rechner $k \cdot f(n)$ ist.
- die Konstante k hängt von Programmiersprache, ..., Prozessorgeschwindigkeit ab

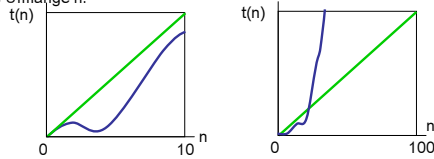


7.1 O-Kalkül (I)

10/66

Edmund Landau, deutscher Zahlentheoretiker, 1877 - 1938

- die Annahme „Unterschied realer Algorithmus – RAM beschreibbar durch konstanten Faktor k “ ist nur für $n \rightarrow \infty$ realistisch.
- daher versteht man unter Aufwand bzw. Komplexität den **asymptotischen Aufwand** bzw. die **asymptotische Komplexität** für sehr große Umfänge n .



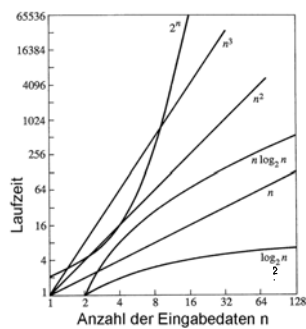
- Ausreißer bei kleinen Umfängen n interessieren nicht.
- konstante Faktoren interessieren nicht
→ Es genügt „interessante“ Operationen zu zählen, wenn die Anzahl anderer Operationen proportional dazu ist



7.1 O-Kalkül (II)

11/66

Repräsentanten verschiedener Funktionenklassen:



Entwurfsschema: gieriger Algorithmus

21/66

- Beispiel:
 - Gegeben: $E = \{7, 22, 39, 20, 9\}$
 - Aufgabe: Packe möglichst viel in Rucksack mit Kapazität 50.

▪ Optimale Lösung: $22 + 20 + 7 = 49$

▪ gieriger Algorithmus:

- 39 paßt \Rightarrow 39 drin
- 22 paßt nicht \Rightarrow 39 drin
- 20 paßt nicht \Rightarrow 39 drin
- 9 paßt \Rightarrow 48 drin
- 7 paßt nicht \Rightarrow 48 drin (suboptimal)

▪ Gierige Algorithmen liefern nur unter bestimmten Bedingungen eine optimale Lösung.



Entwurfsschema: Zufallsgesteuerte Algorithmen

22/66

- Beispiel:
 - Gegeben: $x_0, x_1, \dots, x_n \in \mathbb{N}$ (unsortiert)
 - Gesucht: irgendeine Zahl aus der oberen Hälfte der sortierten Folge

▪ Mögliche Lösung:

- bestimme Maximum (erfordert $n-1$ Vergleiche)

▪ Weitere Lösung:

- bestimme Maximum m von $n/2+1$ Zahlen (erfordert $n/2$ Vergleiche)
 - garantiert, daß m in oberer Hälfte liegt



Entwurfsschema: Zufallsgesteuerte Algorithmen

23/66

- Oft:
 - garantiert korrekte Lösung nicht erforderlich
 - optimale Lösung nicht erforderlich (Näherungslösung)

▪ Wenn korrekte Lösung nicht garantiert sein muß:

- $p(\text{irgend eine Zahl } x_i \text{ ist in unterer Hälfte}) = 1/2$
- $p(\text{irgendwelche } x_i \text{ und } x_j \text{ in unterer Hälfte}) = 1/4$
- $p(\max(x_i, x_j) \text{ ist in unterer Hälfte}) = 1/4$
- $p(\max(x_{i_1}, x_{i_2}, \dots, x_{i_k}) \text{ in unterer Hälfte}) = 2^{-k}$
- $p(\max(x_{i_1}, x_{i_2}, \dots, x_{i_k}) \text{ in oberer Hälfte}) = 1 - 2^{-k}$

▪ Die Wahrscheinlichkeit steigt für große k asymptotisch gegen 1, z.B. $k=20 \Rightarrow p > 0.999999$.

▪ Aufwand ist konstant unabhängig von n .



Entwurfsschema: Zufallsgesteuerte Algorithmen

24/66

- Prinzip:
 - Randomisierter Algorithmus
 - = Deterministischer Algorithmus + Zufallsexperimente
- Einteilung:
 - Monte Carlo (mc = mostly correct)
 - Die Wahrscheinlichkeit $0 < p < 1$ für korrektes Ergebnis kann vorgegeben werden (p-Korrektheit).
 - Terminiert daher stets
 - Hat kürzere Laufzeit als der beste deterministische Algorithmus
 - p kann erhöht werden um den Preis höherer Laufzeit
 - Las Vegas
 - Gibt immer korrektes Ergebnis aus
 - Versucht es zu erraten
 - Laufzeit ist nicht garantiert kürzer als deterministisch (muß nicht terminieren), durchschnittliche Laufzeit ist kürzer



Problemlösung durch Monte Carlo Algorithmus

25/66

- Beispiel:
 - Gegeben: ungerade Zahl n
 - Aufgabe: Stelle fest, ob n Primzahl ist.
- Exaktes Vorgehen:
 - Prüfe für alle Primzahlen z , $2 < z \leq \sqrt{n}$, ob n durch z teilbar ist.
- Problem:
 - Die Zahlen z sind (teilweise) unbekannt.
 - "Gewaltlösung": Prüfe mit allen ungeraden Zahlen $2 < z \leq \sqrt{n}$.
 - Erfordert \sqrt{n} (langsame) Divisionen.



Monte Carlo Algorithmus: Rabin-Miller Primzahltest

26/66

- Konstruiere Monte Carlo Algorithmus **primzahl**, der sich bei der Aussage „n ist Primzahl“ mit Wahrscheinlichkeit $p < 2^{-s}$ irren darf:
 - Konstruiere Funktion $zeuge(a, n)$, die mit Hilfe zufällig gewählter Zahl $a \in \mathbb{N}$, $a < n$
 - TRUE $\Leftrightarrow \exists x \exists y: (x \in \mathbb{N}) \wedge (y \in \mathbb{N}) \wedge (n = x \cdot y)$
 - FALSE \Leftarrow der Nachweis kann mittels a (!) nicht erbracht werden
 - z.B. mit dem „kleinen Fermat“:
 - n prim $\Rightarrow \forall x \in \mathbb{N}, x < n : x^{n-1} \bmod n = 1$
 - Deshalb: $x^{n-1} \bmod n \neq 1 \Rightarrow n$ nicht prim
 - Setze $zeuge(a, n)$ wiederholt mit verschiedenen a ein:
 - istWahrscheinlichPrim = TRUE
 - $\forall a_i \in \mathbb{N}$ wobei $(i, j \in \mathbb{N}, i, j < s$ und aus $i \neq j \Rightarrow a_i \neq a_j)$:
 - $zeuge(a_i, n) \Rightarrow$ istWahrscheinlichPrim = FALSE
 - istWahrscheinlichPrim = TRUE \Rightarrow die Zahl n ist mit der Wahrscheinlichkeit $p < 2^{-s}$ eine Primzahl (Beachte: für $s=10$ ist $p < 10^{-3}$; für $s=20$ ist $p < 10^{-6}$)



Entwurfsschema: Paralleler Algorithmus 27/66

- **Beispiel:**
 - Gegeben: $x_0, x_1, \dots, x_n \in \mathbb{N}$
 - Gesucht: $\sum_{i=0..n} x_i$
- **Algorithmus:**
 - **a** = Prozessor1($\sum_{i=0..(n+2)} x_i$);
 - **b** = Prozessor2($\sum_{i=(n+2)..n} x_i$);
 - Ergebnis = a+b;
- Parallele Algorithmen machen Rechenoperationen, die zeitlich nicht voneinander abhängen, „gleichzeitig“ oder „nebenläufig“.

Informatik I WS 03/04 - Prof. Dr. Gerhard Goos 7. Algorithmenkonstruktion

7.2 Teile-und-Herrsche: divide et impera 28/66

„Teile-und-Herrsche“-Algorithmen (engl.: **divide and conquer**) haben das Grundprinzip:

1. teile die Eingabe in m „kleinere“ Eingaben auf
2. löse Problem rekursiv für diese „kleineren“ Eingaben
3. falls das Problem „sehr klein“ ist, löse es direkt
4. füge die Lösungen für die „kleineren“ Eingaben zur Lösung des Ursprungsproblems zusammen

- häufigster Wert für m: m = 2

Grundprinzip:

```

teile_und_herrsche a
  | umfang a < schwelle = löse_direkt a
  | otherwise let (a1, ..., an) = zerlege a in
                setze_zusammen (teile_und_herrsche a1)
                ...
                (teile_und_herrsche an)
  
```

Informatik I WS 03/04 - Prof. Dr. Gerhard Goos 7. Algorithmenkonstruktion

7.2 Teile-und-Herrsche: Vergleich zur Induktion 29/66

- Beide Methoden haben gemeinsam:
 - Problem ist gelöst (z.B. vordefiniert) für triviale Fälle
 - Jeder nichttriviale Fall kann dadurch gelöst werden, daß seine Lösung eine „einfache“ Folgerung aus der Lösung eines einfacheren Falls ist.
 - Jeder nichttriviale Fall wird schließlich bis zu den trivialen Fällen „heruntergebrochen“.

Informatik I WS 03/04 - Prof. Dr. Gerhard Goos 7. Algorithmenkonstruktion

7.2 Sortieren durch Mischen (III)

39/66

Beispiel: Sortieren durch Mischen

Programm in Haskell:

```
msort [] = []
msort [z] = [z]
msort zs = mische (msort us) (msort vs)
  where n = length zs
        us = take (div n 2) zs
        vs = drop (div n 2) zs

mische [] ys = ys
mische xs [] = xs
mische (x:xs) (y:ys) | x<=y = x:(mische xs (y:ys))
                    | otherwise = y:(mische (x:xs) ys)
```



7.2 Sortieren durch Zerlegen (I)

40/66

Beispiel: Sortieren durch Zerlegen (*quick sort*; Hoare, 1962)

Grundprinzip:

- Für nicht-leere Liste $x:xs$, zerlege xs in zwei Listen:
 - Liste ls enthält alle Elemente aus xs , die kleiner als x sind
 - Liste rs enthält alle Elemente aus xs , die größer gleich x sind
- Sortiere rekursiv die Listen ls und rs
- Setze Ergebnis zusammen

Programm in Haskell:

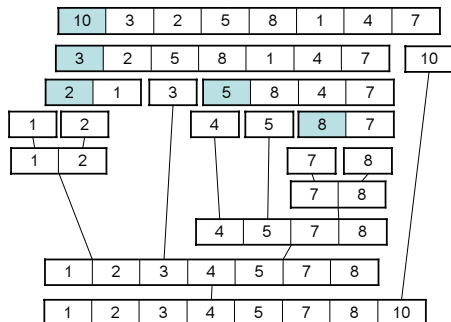
```
qsort [] = []
qsort (x:xs) = qsort [u|u<-xs, u<x]
  ++ [x]
  ++ qsort [u|u<-xs, u>=x]
```



7.2 Sortieren durch Zerlegen (II)

41/66

Beispiel: Sortieren durch Zerlegen



7.2 Algorithmus von Strassen (III)

45/66

- Vorteile von *Strassens* Algorithmus:
 - die Grundidee: wende teile-und-herrsche an und reduziere Anzahl der Grundoperationen im Elementarfall
 - Hauptanwendung: führe Multiplikation sehr langer ganzer Zahlen auf *Strassens* Algorithmus zurück (Kryptographie)
- Nachteile von *Strassens* Algorithmus:
 - geringe numerische Stabilität bei Gleitpunktarithmetik (viele Rechenoperationen und Rundungen → Fehler)
 - schlechte Parallelisierbarkeit
 - hoher Programmieraufwand
- **aber man kann sehen:**
 - Der O-Kalkül betrachtet v.a. das asymptotische Verhalten, unabhängig von der praktischen Relevanz.



7.3 Gierige Algorithmen – Überblick (I)

46/66

Gierige Algorithmen (Verfahren der **schrittweisen Ausschöpfung**, engl. **greedy algorithms**) lösen Aufgaben mit einem Lösungskriterium nach dem Grundprinzip:

- Baue Lösung schrittweise auf (meist Menge oder Liste)
- Anfang: leere Lösungsmenge $lm = \emptyset$.
- Nimm fortlaufend Elemente aus der Eingangsmenge em hinzu, bis eine maximale Lösung erreicht ist.
- Gibt es mehrere Kandidaten zur Aufnahme in die Lösungsmenge, wende das Lösungskriterium, oft ein Optimalitätskriterium, an.
- Entscheidungen, daß ein Element zur Lösung gehört, werden nicht revidiert.



7.3 Gierige Algorithmen – Überblick (II)

47/66

Schema eines solchen Algorithmus:

```
gierig em = gierig' em EmptySet

gierig' em lm
| empty em = lm
| otherwise = gierig' (delete em x) (insert lm x)
                  where x ∈ em && istLösung em lm x
```

Beispiele für Probleme dieser Art

- Gegeben sei eine Menge M von Vektoren. Gesucht ist eine maximale, linear unabhängige Teilmenge $lm \subseteq M$. Der von den Vektoren aus lm aufgespannte Vektorraum umfaßt alle Vektoren aus M .
 - alle anderen mit gierigen Algorithmen lösbaren Probleme lassen sich auf dieses Beispiel zurückführen
- n-Dame Problem: Dieses Problem ist nicht mit gierigen Algorithmen lösbar!



7.3 Beispiel: Zeitplanerstellung

48/66

- Gegeben: Menge M von Kunden k , deren Bedienung je $c(k)$ Zeiteinheiten benötigt
- Gesucht: Bedienreihenfolge, die die durchschnittliche Wartezeit minimiert
- Behauptung: die Reihenfolge $c_1 \leq c_2 \leq \dots \leq c_n$ ist optimal
- Also löst unser Algorithmus das Problem mit
ist Lösung em im $x = c_1 == \text{minimum} [c | c \leftarrow em]$



7.3 Zeitplanerstellung: Beweis

49/66

- Gesamtwarezeit aller Kunden bei Bedienung in sortierter Folge:
 $C = c_1 + c_1 + c_2 + \dots + c_1 + L + c_n = \sum_{k=1}^n (n-k+1)c_k$
- Vertauschung der Kunden j, k mit $j < k$ ergibt die Wartezeit C' :
 $C - C' = (n-j+1)(c_j - c_k) + (n-k+1)(c_k - c_j) = (k-j)(c_j - c_k)$
- Nach Voraussetzung ist $c_j \leq c_k$, also $C - C' \leq 0$
- Vorziehen von Kunden mit längerer Bedienzeit verlängert also die Gesamtwarezeit
- Aufwand des Algorithmus mindestens $O(n \log n)$: dieser Aufwand wird für das Sortieren der Kunden mindestens benötigt. Der gierige Algorithmus läuft dann in linearer Zeit $O(n)$



7.3 Zeitplanerstellung: Verallgemeinerung

50/66

- Die Kunden werden an m Bedienstationen parallel bedient
- An Station x warten n_x Kunden. Sei $g < h$ und für Bedienzeiten $j^{(g)}, k^{(h)}$ gelte $j^{(g)} < k^{(h)}$. Vertauschung dieser Aufträge liefert
 $C - C' = (n_g - n_h + k - j)(c_j^{(g)} - c_k^{(h)})$
- Wenn $n_g = n_h$, sollte also $c_j^{(g)} \leq c_k^{(h)}$ gelten
- Lösung also wieder durch Sortieren der Kunden und Verteilung auf die Stationen in Reihenfolge aufsteigender Bedienzeiten
- Dies sind elementare Beispiele einer großen Klasse von Problemen, die bei der optimalen Steuerung von Maschinenparks wichtig sind (aber im Supermarkt und beim Arzt üblicherweise nicht angewandt werden)



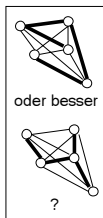
7.3 Minimale spannende Bäume (I)

51/66

Beispiel:

Problemstellung:

- Gegeben ist eine Landkarte als Graph G.
- Es soll ein Wasserleitungsnetz aufgebaut werden, das alle Ortschaften versorgt.
- Die Gesamtkosten seien proportional zur Summe der Längen aller vorkommenden Leitungen.
- Welche Ortschaften sollten verbunden werden (Teilgraph von G), so daß die Kosten minimal werden?



Ähnliche Probleme gibt es beim Entwurf von Rechnernetzen



7.3 Minimale spannende Bäume (II)

52/66

Gegeben: Graph mit Kanten k , die mit Kosten c_k bewertet sind

Gesucht:

- Teilgraph, der alle Ecken enthält bei minimalen Kantengewichten
- gesuchter Graph ist offensichtlich azyklisch (\Rightarrow **Baum**)
- gesuchter Graph enthält alle Ecken (\Rightarrow **spannender Baum**)

\Rightarrow **minimal spannender Baum**

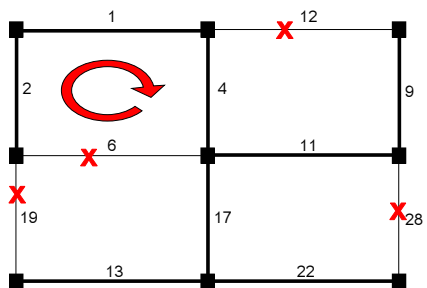
- *Kruskals* Algorithmus:
- Füge bei jedem Schritt diejenige Kante hinzu, welche
 - das kleinste Gewicht unter den verbliebenen Kandidaten hat
 - keinen Zyklus erzeugt
- Wiederhole solange, bis alle Ecken verbunden sind.



7.3 Minimale spannende Bäume (III)

53/66

Beispiel: *Kruskals* Algorithmus



7.3 Kruskals Algorithmus: Verfeinerung der Idee

54/66

1. Grundverfahren: benutze den Algorithmus gierig mit:
ist_Lösung em lm k = k ist minimal unter den Kanten 2 em,
die mit genau einer Ecke zu lm gehören
 - Problem: mit jeder neuen Ecke in der Lösung ändert sich die Menge der zu untersuchenden Kanten
2. Kruskals Verbesserungen:
 - Beginne mit sortierter Kantenliste in aufsteigender Reihenfolge
 - Kante gehört zur Lösung, wenn sie
 - zwei noch nicht betrachtete Ecken verbindet (neuer Baum)
 - einen vorhandenen Baum um eine Ecke erweitert
 - zwei verschiedene Bäume verbindet
 - Jetzt kann man die Kanten nacheinander betrachten und sofort entscheiden, ob die Kante zur Lösung gehört oder nicht



7.3 Kruskals Algorithmus: Programm

55/66

- Betrachte zusammenhängenden ungerichteten Graph (E,K) mit mindestens 2 Ecken.
- Eckenmenge E = [1..n]. Kante von i nach j mit Kosten c repräsentiert durch Tripel (i,j,c).
- Dann wende den Algorithmus der vorigen Folie an



7.3 Kruskals Algorithmus: Programm

56/66

```
type Kante = (Int,Int,Int)
kruskal :: [Kante] -> [Kante]
kruskal kanten = kruskal' (sort kanten) []
kruskal' [] lm = head lm
kruskal' ((i,j,c):kanten) lm | vorhanden i j lm = kruskal' kanten lm
                           | otherwise = kruskal' kanten (zufuegen (i,j,c) lm)

where
  vorhanden i j [] = False
  vorhanden i j (b:bs) | (teste i b) && (teste j b) = True
                      | otherwise = vorhanden i j bs
  zufuegen k [] = [[k]]
  zufuegen (i,j,c) (b:bs) | (teste i b) || (teste j b) = verein (i,j,c) b bs
                        | otherwise = b : (zufuegen (i,j,c) bs)
  verein (i,j,c) b [] = [(i,j,c) : b]
  verein (i,j,c) b (y:ys) | (teste i y) || (teste j y) = ((i,j,c) : b) ++ y : ys
                        | otherwise = y : (verein (i,j,c) b ys)
  teste i [] = False
  teste i ((i',j',c'):ks) | (i == i') || (i == j') = True
                        | otherwise = teste i ks
```



7.3 Kruskals Algorithmus: Ergebnis

57/66

Main> kruskal [(1,2,1),(2,3,12),(4,5,6),(5,6,11),(7,8,13),(8,9,22),
(1,4,2),(4,7,19),(2,5,4),(5,8,17),(3,6,9),(6,9,28)]
[(8,9,22),(5,8,17),(5,6,11),(2,5,4),(1,4,2),(1,2,1),(3,6,9),(7,8,13)] :: [Kante]
(1458 reductions, 2328 cells)



7.3 Topologisches Sortieren (I)

58/66

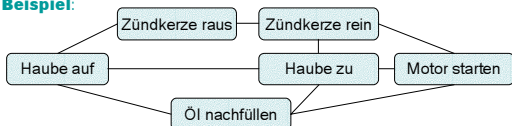
- Gegeben eine endliche halbgeordnete Menge (M, \preceq) . Konstruiere eine totale Ordnung \leq so, daß aus $a \preceq b$ folgt $a \leq b$ (vgl. Kap. 2)
- Gieriger Algorithmus: Baue eine geordnete Liste L neu auf, entnimm der Menge M nacheinander Elemente, die minimal bezüglich \preceq sind und füge sie vorne in L ein.
- Ergebnis: die lineare Reihenfolge der Elemente in L verfeinert die Ordnung \preceq mit einem maximalen Element als erstem Element.
- Beweis: Sei $L = [a_1, \dots, a_n]$. Annahme $a_i \preceq a_j$ und $i < j$. Dann wäre a_j vor a_i in L aufgenommen worden, obwohl a_j nicht minimal in M war. Widerspruch.



7.3 Topologisches Sortieren: Beispiel I

59/66

Beispiel:



Der Graph gibt an, welche Tätigkeiten sich gegenseitig bedingen.

Frage:

- In welcher Reihenfolge müssen diese Tätigkeiten abgearbeitet werden, so daß alle Abhängigkeiten erfüllt sind?
- (Welche Tätigkeiten könnten parallel ausgeführt werden?)



7.3 Exkurs: Tiefensuche in azyklischen Graphen

63/66

- Gerichtete azyklische Graphen entsprechen halbgeordneten Mengen (M, \leq) .
- Tiefensuche in azyklischen Graphen liefert keine Rückkanten
- Die Blätter des ersten Baums in FB, den die Tiefensuche liefert, sind minimal in der Ordnung \leq
- Die Blätter e aller weiteren Bäume sind entweder minimal oder die Elemente a mit $a \leq e$ gehören zu Bäumen in der bereits konstruierten Baumfolge (ignoriere Kante (e,a) , Beweis: vollständige Induktion)
- Wenn wir die Ecken e, e', \dots eines Baumes in Postfixreihenfolge anschreiben, d.h. immer dann, wenn wir die Ecke zuletzt besuchen, so gilt $e \leq e'$, wenn e vor e' angeschrieben wird, oder e und e' sind in der Halbordnung unvergleichbar
- Folgt der Baum b in der Baumfolge FB nach dem Baum b' , so gilt für Ecken $e \in b, e' \in b'$: entweder $e' \leq e$ oder e, e' sind unvergleichbar.
- Also: wenn wir die Ecken aller Bäume in FB in Postfixreihenfolge anschreiben und dabei die Reihenfolge der Bäume in FB bewahren, so gilt für Ecken e, e' : wenn $e' \leq e$, dann kommt e' auch vor e in der Reihenfolge
- Also ist diese Reihenfolge eine topologische Sortierung von (M, \leq) .



7.3 Topologisches Sortieren: Haskell-Programm

64/66

```
type Eckennr = Int
type Kanten = [Eckennr]
type Ecke = (Eckennr, Kanten)
type Graph = [Ecke]

topSort g = snd (until p f (g, []))
  where p (g, res) = g == []
        f (g, res) = (g', res' ++ res)
          where (g', _, res') = tiefensuche g [ecke] []; ecke = fst (head g)

tiefensuche [] _ res = ([], [], res)
tiefensuche g [] res = (g, [], res)
tiefensuche g kanten res = until p f (g, kanten, res)
  where p (g, kanten, res) = kanten == []
        f (g, e:rest, res) | vorhanden = tiefensuche g' rest ((e:res') ++ res)
                          | otherwise = tiefensuche g rest res
          where (vorhanden, es) = finde e g
                (g', _res') = tiefensuche (ohne e g) es []

ohne e [] = []
ohne e ((nr, es):rest)
  | e == nr = rest
  | otherwise = (nr, es) : (ohne e rest)

finde e [] = (False, [])
finde e ((nr, es):rest)
  | e == nr = (True, es)
  | otherwise = finde e rest
```



7.3 Topologisches Sortieren: Ergebnisse

65/66

```
TopSort> topSort [(1,[2,3,4]),(2,[5]),(3,[6]),(4,[3,6]),(5,[6]),(6,[])]
[1,4,3,2,5,6] :: [Eckennr]
(308 reductions, 650 cells)
TopSort> topSort [(1,[2,8]), (2,[3,8]), (3,[6]), (4,[3,5]), (5,[6]), (6,[]), (7,[8]),
(8,[]), (9,[])]
[9,7,4,5,1,2,8,3,6] :: [Eckennr]
(534 reductions, 1149 cells)
TopSort> topSort [(8,[4]),(12,[4,6]),
(4,[2]),(6,[2,3]),(9,[3]),(10,[2,5]),(14,[2,7]),(15,[3,5]),
(2,[1]),(3,[1]),(5,[1]),(7,[1]),(11,[1]),(13,[1]),(1,[])]
[13,11,15,14,7,10,5,9,12,6,3,8,4,2] :: [Eckennr]
(1177 reductions, 2411 cells)
```



- Was haben wir gelernt?
 - Denkweise der Informatik
 - Algorithmische Problemlösung
 - Berechnungsmodelle
 - Semi-Thue, Markov
 - Termersetzung, Lambda (Haskell)
 - Komplexität: O-Kalkül
 - Datenmodelle
 - Formale Sprachen
 - Relationale Algebra (SQL)
 - Datenstrukturen
 - Liste, Schlange, Keller, ...
 - Bäume und Graphen
 - Reihungen, Heaps
 - Algorithmen
 - sortieren, suchen
- Was folgt?
 - Neue Berechnungsmodelle
 - Endliche Automaten
 - Turingmaschinen
 - Berechenbarkeit, Komplexität
 - Vertiefung der Datenmodelle
 - Formale Sprachen, Bezug zu Automaten
 - Neue Programmiersprachen
 - imperativ, strukturiert, OO
 - deklarativ, Logiksprachen...
 - Neue Datenstrukturen
 - Haschtabellen
 - Balancierte Bäume
- **Ein Ingenieur löst praktische Probleme.
Die Theorie zeigt, wie!**