



Effiziente Algorithmen

Hartmut Klauck
Universität Frankfurt
SS 06
18.4.

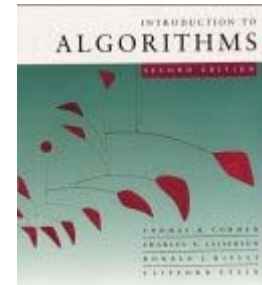


Organisatorisches

- Vorlesungen: Di. 14-16 c.t., Do 12-14 c.t. Magnus
- Übung: Mi. 14-16 SR307
- Schein: Übung
 - 50% der Übungspunkte
 - Aktive Teilnahme
- Zuordnung: T3, ThBI
- Voraussetzung: Vordiplom (Informatik, Mathematik)
- Website:
www.thi.informatik.uni-frankfurt.de/~klauck/EA06.html

Literatur

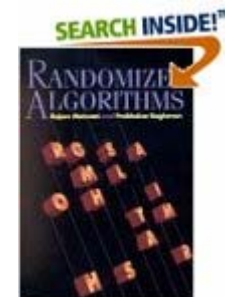
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest:
Introduction to Algorithms
(MIT Press)



Deutsch als **Algorithmen - Eine Einführung**
Bei Oldenbourg



- Rajeev Motwani, Prabhakar Raghavan:
Randomized Algorithms
(Cambridge)



Literatur

- Jon Kleinberg, Eva Tardos: **Algorithm Design**
(Pearson)





Übersicht

- Inhalt der Vorlesung:
 - Entwurf effizienter Algorithmen für interessante Probleme
 - Effizienz:
 - Theoretische Effizienz (z.B. Polynomialzeit)
 - Praktische Effizienz
 - Andere Modelle, z.B. Algorithmen mit konstanter Laufzeit
 - Probleme:
 - Aus verschiedenen Bereichen



Übersicht

- Probleme:
 - Graphprobleme
 - Optimierungsprobleme
 - Geometrische Probleme
 - Online-Probleme
- Techniken:
 - Randomisierung
 - Approximation
 - Greedy Algorithmen
 - Divide and Conquer
 - Dynamisches Programmieren
 - ...



Übersicht

- Anordnung der Vorlesung weitgehend nach Problemfeldern
- Beginnen mit **Graphalgorithmen**
 - Durchsuchen von Graphen
 - Kürzeste Wege
 - Spannbäume
 - Matchings
 - Flussalgorithmen



Übersicht

- Dabei betrachten wir Entwurfstechniken:
 - Greedy Algorithmen
 - Dynamisches Programmieren
 - Randomisierung
- Und sondern Datenstrukturprobleme aus, die wir getrennt betrachten
 - Amortisierte Analyse

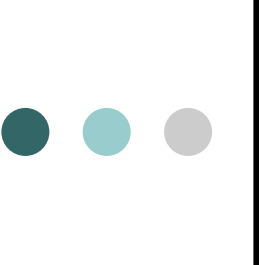


Übersicht

- Später:
 - Andere Problemfelder, z.B.
 - Matrixalgorithmen (schnelle Matrixmultiplikation, FFT)
 - Lineares Programmieren
 - Zahlentheoretische Algorithmen (Primzahltests)
 - Ansätze zur Lösung NP-schwieriger Probleme:
 - Approximation, lokale Optimierung
 - Online Algorithmen
 - Wenn die Eingabe erst im Laufe des Algorithmus bekannt wird

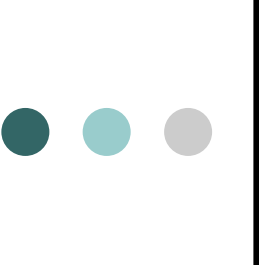


Graphalgorithmen



Graphen

- Ein Graph ist gegeben durch $G=(V,E)$
 - wobei V die Menge der Knoten ist ($|V|=n$)
 - $E \subseteq V \times V$ die Menge der Kanten ($|E|=m$)
 - Es gibt gerichtete und ungerichtete Graphen!
- Wie wird ein Graph repräsentiert?
 - Adjazenzmatrix (dichte Graphen)
 - Adjazenzliste (sonst)
- Adjazenzmatrix: $A[i,j]=1$ gdw $(i,j) \in E$
- Adjazenzliste: Array der Länge n von Listen, jede Liste enthält alle Nachbarn des entsprechenden Knoten



Durchsuchen von Graphen

- Gegeben sei ein gerichteter Graph G
- Weiterhin ein Startknoten s
- Ziel ist es, den Graphen zu durchsuchen, z.B. um einen Zielknoten t zu finden (bzw. zu entscheiden, ob t von s erreichbar)

- Zwei Varianten:
 - Breitensuche
 - Tiefensuche



Graphsuche

- Allgemeines Gerüst:
 - Verwende eine Datenstruktur, die folgende Operationen unterstützt:
 - Einfügen von Knoten
 - Entfernen von Knoten
 - s sei der aktive Knoten
 - Iteriere:
 - vom aktiven Knoten füge alle bisher unbesuchten Nachbarn ein
 - Entferne einen Knoten und und mache ihn aktiv/besuche ihn
 - Zusätzlich Array: schon besucht/noch nicht besucht



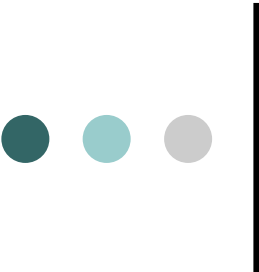
Die Datenstruktur

- Alternative 1: queue
 - Liste, FIFO (first in first out)
- Alternative 2: stack
 - LIFO (last in first out)



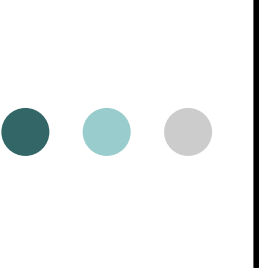
Ergebnis

- Alternative 1: FIFO
 - Breitensuche
 - Nachbarn werden eingefügt, und vor den Nachbarn der Nachbarn besucht
- Alternative 2: LIFO
 - Tiefensuche
 - Nachbarn werden eingefügt, der letzte Nachbar wird neuer Startknoten und seine Nachbarn sind zuerst dran



Durchsuchen von Graphen

- Offensichtlich werden mit beiden Methoden alle erreichbaren Knoten irgendwann besucht
- Verschiedene Anwendungen:
 - Breitensuche findet kürzeste Wege in ungewichteten Graphen
 - Tiefensuche erzeugt eine topologische Sortierung auf dags (directed acyclic graphs), d.h. eine Nummerierung der Knoten, so dass Kanten nur von niedrigen zu höheren Nummern verlaufen



Breitensuche und kürzeste Wege

- Gegeben sei G , sowie ein Startknoten s und ein Zielknoten t
- Finde den kürzesten Weg von s nach t (so einer existiert)!
- $\delta(u,v)$ sei Länge des kürzesten Weges von u nach v

- Verwende Breitensuche von s , stoppe wenn t gefunden.
- Erzeuge Breitensuchbaum (d.h. Menge der Kanten, entlang denen neue Knoten besucht werden)
- Dazu speichere für jeden neu besuchten/aktivierten Knoten seinen Vorgänger als $\pi(v)$



Breitensuchbaum

- Speichere alle Kanten, auf denen neue Knoten besucht werden
- Beobachtung: Dies ergibt einen Baum
 - Zu Beginn ist keine Kante gespeichert
 - Wenn eine Kante gespeichert wird, verläuft sie von einem besuchten zu einem unbesuchten Knoten
 - Jeder Knoten wird nur einmal besucht, hat also nur einen Vorgänger
- Zusätzlich verwalte ein Distanzarray
 - Jeder Knoten erhält eine Distanz $d(v)$
 - $d(s)=0$
 - wenn v von u aus eingefügt wird, setze $d(v)=d(u)+1$



Korrektheit der Distanz

- Klar: $d(s)=0$ ist korrekter Distanzwert
- Angenommen, v wird irgendwann von u aus besucht, und per Induktion ist $d(u)$ korrekt
- Zu zeigen: $d(v)=d(u)+1=\delta(s,v)$
 - $d(v)=d(u)+1 = \delta(s,u)+1$
 $\geq \delta(s,v)$, denn es gibt einen Weg
 $s \rightarrow u \rightarrow v$
 - Noch zu zeigen: $d(v) \leq \delta(s,v)$



Korrektheit

- **Behauptung:** Der Breitensuchbaum enthält kürzeste Pfade von $s \rightarrow v$ für alle v
 - $d(v)$ ist Tiefe von v im Breitensuchbaum, daher korrekte Distanz, also $d(v) = \delta(u, v)$.
- **Beweis:**
 - Für s korrekt, insbesondere sind also alle Knoten in Tiefe 0 im Baum
 - Sei v ein Knoten in Tiefe d des Baumes
 - Per Induktionsannahme seien alle Knoten in Tiefe $< d$ durch kürzeste Pfade mit s verbunden, und alle Knoten in Distanz $< d$ von s seien dort vorhanden
 - Angenommen, es gebe einen kürzeren Weg $s \rightarrow v$ im Graphen, also mit Länge $\leq d-1$
 - Folge diesem Pfad rückwärts, bis ein Knoten w erreicht wird, der im Breitensuchbaum in Tiefe $< d-1$ liegt
 - w ist im Baum über einen kürzesten Weg mit s verbunden
 - $w \neq u$
 - Der Nachfolger von w liegt in Tiefe $\geq d$, hat aber Distanz $\leq d-1$. Dies ist ein Widerspruch.



Laufzeit

- Queue und Stackoperationen laufen in konstanter Zeit (uniformes Kostenmass)
- Im Algorithmus wird jeder Knoten genau einmal eingefügt und einmal entfernt
- Um unbesuchte Nachbarn zu finden müssen alle Kanten einmal betrachtet werden
- Gesamtkosten: $O(n+m)$
 - Adjazenzliste
 - Bei Adjazenzmatrix: $O(n^2)$