Vorlesung: Algorithmische Diskrete Mathematik

TECHNISCHE UNIVERSITÄT DARMSTADT

Raymond Hemmecke

Vertretungsprofessor für Algorithmische Diskrete Mathematik

Inhalt der Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Komplexitätstheorie**
  - Datenstrukturen und Kodierungsschemata
  - asymptotische Notation, untere und obere Schranken
  - Klassen $\mathcal{P}, \mathcal{NP}, \mathcal{NP}$-vollständig
- **Algorithmen auf Graphen**
  - DFS-Algorithmus (aufspannende Bäume)
  - Greedy-Algorithmus (minimale aufspannende Bäume)
  - Dijkstra, Moore-Bellman, Yen-Variante (kürzeste Wege in Graphen)
  - Ford-Fulkerson (maximale Flüsse in Netzwerken, Matching in bipartiten Graphen)
- **Sortieren in Arrays**
  - Mergesort, Quicksort, Heapsort
  - Divide-and-Conquer
  - untere Komplexitätsschranken für das Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Grundlegende Begriffe der Graphentheorie

Was ist ein Graph?

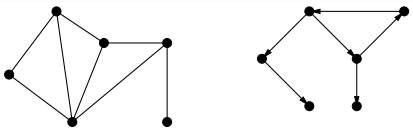
TECHNISCHE UNIVERSITÄT DARMSTADT

Graph

- Ein ungerichter **Graph** ist ein Paar $G = (V, E)$ disjunkter Mengen, wobei die Elemente von E ungeordnete Paare von Elementen aus V sind.
- Die Elemente aus V heißen **Knoten**, die Elemente aus E **Kanten**.

Digraph

- Ein **gerichter Graph** ist ein Paar $G = (V, A)$ disjunkter Mengen, wobei die Elemente von A geordnete Paare von Elementen aus V sind.
- Die Elemente aus A heißen **Bögen** oder **gerichtete Kanten**.



Nachbarschaftsbeziehungen

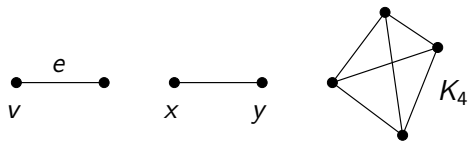
TECHNISCHE UNIVERSITÄT DARMSTADT

Inzidenz, Grad/Valenz

- Ein Knoten v heißt mit einer Kante e **inzident**, wenn $v \in e$ gilt.
- Der **Grad**/die **Valenz** eines Knoten v ist die Anzahl der mit v inzidenten Kanten.

Adjazenz, vollständiger Graph

- Zwei Knoten x und y heißen **adjazent** oder **benachbart** in G, wenn $xy \in E(G)$ ist.
- Sind je zwei Knoten von G benachbart, so heißt G **vollständig**.

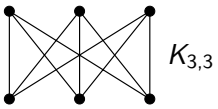


Bipartite Graphen

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

- Ein Graph heißt **bipartit**, wenn sich V disjunkt in zwei Teile V_1 und V_2 zerteilen lässt, so dass jede Kante in E einen Endknoten in V_1 und einen Endknoten in V_2 besitzt.



Fakt → Beweis als Übungsaufgabe

- Ein Graph ist genau dann bipartit, wenn er nur Kreise gerader Länge enthält.

Was sind Kreise?

TECHNISCHE UNIVERSITÄT DARMSTADT

Pfad, Weg, Kreis

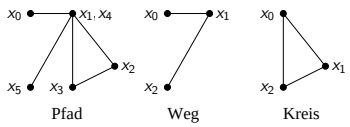
- Ein **Pfad** ist ein nichtleerer Graph $P = (V, E)$ der Form

$$V = \{x_0, x_1, \dots, x_k\}, E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}.$$

- Ein **Weg** ist ein Pfad, in dem alle x_i paarweise verschieden sind.
- Ein **Kreis** ist ein nichtleerer Graph $P = (V, E)$ der Form

$$V = \{x_0, x_1, \dots, x_k\}, E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k, x_kx_0\},$$

wobei die x_i paarweise verschieden sind.



Zusammenhangskomponenten, Bäume und Wälder

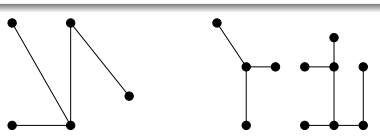
TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenhängende Graphen

- Ein Graph G heißt **zusammenhängend**, falls es zu jedem Paar $v, w \in V$ einen Weg von v nach w in G gibt.
- Die zusammenhängenden Teile von G heißen **Zusammenhangskomponenten**.

Bäume und Wälder

- Ein **Baum** ist ein zusammenhängender Graph, der keinen Kreis besitzt.
- Ein **Wald** ist ein Graph, der keinen Kreis besitzt.



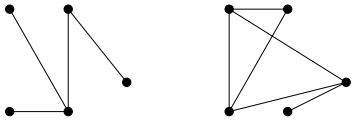
Baum Wald, 2 Komponenten

Komplementärgraph

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplementärgraph

- Der zu G **komplementäre Graph** $\bar{G}$ ist der Graph $\bar{G} = (V, \bar{E})$, mit $ij \in \bar{E} \Leftrightarrow ij \notin E$.



Graph Komplement

Fakt → Beweis als Übungsaufgabe

- Mindestens einer der Graphen G oder $\bar{G}$ ist zusammenhängend.

Untergraphen, aufspannende Untergraphen

TECHNISCHE UNIVERSITÄT DARMSTADT

Untergraph, aufspannende Untergraphen

- $G' = (V', E')$ heißt **Untergraph** von $G = (V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$ gilt.
- $G' \subseteq G$ heißt **aufspannender Untergraph** von G, falls $V' = V$ gilt.

Graph mit Untergraph aufspannender Baum

Finden eines aufspannenden Waldes → DFS-Algorithmus

- Für gegebenes $G = (V, E)$ finde man einen aufspannenden Wald.

Probleme auf Graphen (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

Finden eines Kreises/Hamiltonischen Kreises

- Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen Kreis enthält. Falls ja, gib einen solchen an.
- Entscheide für einen gegebenen Graphen $G = (V, E)$, ob er einen hamiltonischen Kreis enthält. Falls ja, gib einen solchen an. (Ein **hamiltonischer Kreis** ist ein Kreis, der jeden Knoten genau einmal durchläuft.)

Finden eines aufspannenden Waldes → DFS-Algorithmus

- Für gegebenes $G = (V, E)$ finde man einen aufspannenden Wald.

Maximales Matching

- Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Kanten aus E, so dass je zwei Kanten nicht inzident sind.

Probleme auf Graphen (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximale stabile Mengen/Cliquen

- Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Knoten aus V, so dass je zwei Knoten nicht benachbart sind. → stabile Menge
- Finde für gegebenen Graphen $G = (V, E)$ die maximale Anzahl von Knoten aus V, so dass je zwei Knoten benachbart sind. → Clique

Kürzeste Wege

- Für gegebenes $G = (V, E)$ und gegebenen Kantengewichten c_{ij} finde man einen kürzesten Weg zwischen zwei gegebenen Knoten v und w.

Färbungsproblem

- Finde für gegebenen Graphen $G = (V, E)$ die minimale Anzahl k von Farben, so dass sich die Knoten von G so mit k Farben färben lassen, dass benachbarte Knoten unterschiedlich gefärbt sind.

TECHNISCHE UNIVERSITÄT DARMSTADT

Asymptotische Notation

Asymptotische Notation

TECHNISCHE UNIVERSITÄT DARMSTADT

Obere Schranken: O-Notation

- Sei $g : \mathbb{N} \to \mathbb{R}$. Dann bezeichnet

$$O(g) := \{f : \mathbb{N} \to \mathbb{R} : \exists c > 0, n_0 \in \mathbb{N} \text{ mit } |f(n)| \leq c|g(n)| \; \forall n \geq n_0\}$$

die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}$, für die zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt $|f(n)| \leq c|g(n)|$.

Bemerkungen

- Die asymptotische Notation vernachlässigt Konstanten und Terme niedrigerer Ordnung (wie z.B. die Terme $a_k n^k$ mit $k < m$ im Polynom).

Satz

- Für ein Polynom $f(n) = a_m n^m + \ldots + a_1 n + a_0$ vom Grade m gilt $f \in O(n^m)$.

Vergleich

TECHNISCHE UNIVERSITÄT DARMSTADT

Die nachfolgende Tabelle zeigt, wie sechs typischen Funktionen anwachsen, wobei die Konstante gleich 1 gesetzt wurde. Wie man feststellt, zeigen die Zeiten vom Typ $O(n)$ und $O(n \log n)$ ein wesentlich schwächeres Wachstum also die anderen.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Untere Schranken

TECHNISCHE UNIVERSITÄT DARMSTADT

Ω- und Θ-Notation

- Seien $f, g : \mathbb{N} \to \mathbb{R}$. Dann ist $f \in \Omega(g)$, wenn zwei positive Konstanten $c \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ existieren, so dass für alle $n \geq n_0$ gilt:

$$|f(n)| \geq c|g(n)|.$$

- $f \in \Theta(g)$, wenn es positive Konstanten $c_1, c_2 \in \mathbb{R}$ und $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ gilt $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$.
- Falls $f \in \Theta(g)$, dann ist g sowohl eine obere als auch eine untere Schranke für f.

Beispiel: Sequentielle Suche

- Sei $f(n)$ die Anzahl von Vergleichen bei der sequentiellen Suche nach einem bestimmten Wert in einem unsortierten Array mit n Komponenten. Dann ist $f \in O(n)$, da man ja mit n Vergleichen auskommt.
- Andererseits muss man auch jede Komponente überprüfen, denn ihr Wert könnte ja der gesuchte Wert sein. Also $f \in \Omega(n)$ und damit $f \in \Theta(n)$.

Ein weiteres Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Matrixmultiplikation

- Bei der Matrixmultiplikation von $n \times n$ Matrizen ergibt sich die Berechnung eines Eintrags c_{ij} von $C = A \cdot B$ gemäß $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Sie erfordert also n Multiplikationen und $n-1$ Additionen.
- Insgesamt sind für ganz C also n^2 Einträge c_{ij} zu berechnen, und somit $n^2(n+n-1) = 2n^3 - n^2 = O(n^3)$ arithmetische Operationen insgesamt auszuführen.
- Da jeder Algorithmus für die Matrixmultiplikation n^2 Einträge berechnen muss, folgt andererseits, dass jeder Algorithmus zur Matrixmultiplikation von zwei $n \times n$ Matrizen $\Omega(n^2)$ Operationen benötigt.
- Es klafft zwischen $\Omega(n^2)$ und $O(n^3)$ also noch eine "Komplexitätslücke". Die schnellsten bekannten Algorithmen zur Matrixmultiplikation kommen mit $O(n^{2,376})$ Operationen aus.

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexitätstheorie

Was ist ein Problem?

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

- Ein **Problem** ist eine allgemeine Fragestellung, bei der mehrere Parameter offen gelassen sind und für die eine Lösung oder Antwort gesucht wird.
- Ein Problem ist dadurch definiert, dass alle seine Parameter beschrieben werden und dass genau angegeben wird, welche Eigenschaften eine Antwort (Lösung) haben soll.
- Sind alle Parameter eines Problems mit expliziten Daten belegt, dann sprechen wir von einem Problembeispiel (**Instanz**).

Beispiel

- Finde einen kürzesten Hamiltonischen Kreis.
- Offene Parameter: Anzahl Städte, Entfernungen

Algorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Was ist ein Algorithmus?

- Ein **Algorithmus** ist eine Anleitung zur schrittweisen Lösung eines Problems. Wir sagen, ein Algorithmus A löst ein Problem Π, falls A für alle Problembeispiele $I \in \Pi$, eine Lösung in einer endlichen Anzahl an Schritten findet.
- Ein Schritt ist eine elementare Operation: Addieren, Subtrahieren, Vergleichen, Multiplikation, Division.
- Die Laufzeit eines Algorithmus ist die Anzahl der Schritte, die notwendig sind zur Lösung des Problembeispiels.

Forschungsschwerpunkte zu Algorithmen

- Entwurf von Algorithmen
- Berechenbarkeit: Was kann durch einen Algorithmus berechnet werden?
- Komplexität von Algorithmen
- Korrektheit von Algorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexität von Algorithmen

Was ist Effizienz?

TECHNISCHE UNIVERSITÄT DARMSTADT

Was ist Effizienz?

- Komplexitätstheorie
- Speicher- und Laufzeitkomplexität

Trivial

- Laufzeit eines Algorithmus hängt ab von der "Größe" der Eingabedaten
- Laufzeitanalyse erfordert eine Beschreibung, wie Problembeispiele dargestellt werden (Kodierungsschema)
- Notwendigkeit exakter Definitionen
  - geeignetes Rechnermodell $\rightarrow$ **Turing-Maschine**

Kodierungsschema

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen

- Ganze Zahlen werden binär dargestellt, d.h. wir schreiben

$$n = \pm \sum_{i=0}^{k} x_i \cdot 2^i,$$

mit $x_i \in \{0, 1\}$ und $k = \lfloor \log_2(|n|) \rfloor$.

- D.h. die Kodierungslänge $\langle n \rangle$ einer ganzen Zahl n ist gegeben durch die Formel

$$\langle n \rangle = \lceil \log_2(|n| + 1) \rceil + 1 = \lfloor \log_2(|n|) \rfloor + 2,$$

wobei $+1$ wegen des Vorzeichens $+$ oder $-$.

Rationale Zahlen

- Sei $r \in \mathbb{Q}$. Dann existieren $p \in \mathbb{Z}$ und $q \in \mathbb{Z}$, teilerfremd, mit $r = \frac{p}{q}$.

$$\langle r \rangle = \langle p \rangle + \langle q \rangle$$

Kodierungsschema (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Vektoren

- Für $x = (x_1, \ldots, x_n)^\intercal \in \mathbb{Q}^n$ ist

$$\langle x \rangle = \sum_{i=1}^{n} \langle x_i \rangle.$$

Matrizen

- Für $A \in \mathbb{Q}^{m \times n}$ ist

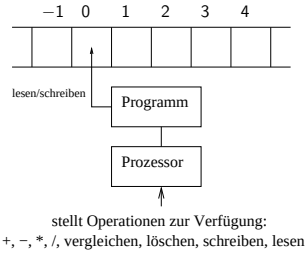
$$\langle A \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle.$$

Rechnermodell

Nach Festlegung der Kodierungsvorschrift muss ein Rechnermodell entworfen werden, auf dem unsere Speicher- und Laufzeitberechnungen durchgeführt werden.

In der Komplexitätstheorie: **Turing-Maschine** (ein ganz normaler Computer)

Unendliches Band, auf dem Lese-/Schreiboperationen durchgeführt werden



Ablauf eines Algorithmus auf der Turing-Maschine

Algorithmus A soll Problembeispiel I des Problems Π lösen. Alle Problembeispiele liegen in kodierter Form vor.

Inputlänge

- Die Anzahl der Speicherplätze, die nötig sind, um I vollständig anzugeben, heißt **Inputlänge**, $\langle I \rangle$.

Der Algorithmus liest die Daten und beginnt dann, Operationen auszuführen, d.h. Zahlen zu berechnen, zu speichern, zu löschen.

Speicherbedarf

- Die Anzahl der Speicherplätze, die während der Ausführung des Algorithmus mindestens einmal benutzt werden, nennen wir **Speicherbedarf** von A zur Lösung von I.

Laufzeit

Informell

- Die **Laufzeit** von A zur Lösung von I ist die Anzahl elementarer Operationen. Dazu zählen $+$, $-$, $*$, $/$, Vergleich, Löschen, Schreiben, Lesen von rationalen Zahlen.

Dies ist jedoch zu unpräzise!

- Zur Darstellung der entsprechenden Zahlen werden mehrere Bits benötigt.

→ Für jede Operation müssen wir mehrere Bits zählen.

Korrekte Definition

- Die **Laufzeit** von A zur Lösung von I ist definiert durch die Anzahl elementarer Operationen, die A ausgeführt hat, um I zu lösen, multipliziert mit der größten Kodierungslänge der während der Berechnung aufgetretenen ganzen oder rationalen Zahl.

Laufzeitfunktion, Speicherplatzfunktion (worst-case)

Laufzeitfunktion, polynomielle Laufzeit

- Sei A ein Algorithmus zur Lösung eines Problems Π. Die Funktion $f_A : \mathbb{N} \to \mathbb{N}$,

$$f_A(n) = \max\{\text{Laufzeit von A zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}$$

heißt **Laufzeitfunktion** von A.

- Der Algorithmus A hat eine **polynomiale Laufzeit** (kurz: A ist ein polynomialer Algorithmus), wenn es ein Polynom $p : \mathbb{N} \to \mathbb{N}$ gibt mit $f_A(n) \leq p(n)$ für alle $n \in \mathbb{N}$.

Speicherplatzfunktion, polynomieller Speicherbedarf

- Die **Speicherplatzfunktion** $s_A : \mathbb{N} \to \mathbb{N}$ von A ist definiert durch

$$s_A(n) = \max\{\text{Speicherbedarf von } A \text{ zur Lösung von } I : I \in \Pi, \langle I \rangle \leq n\}.$$

- Der Algorithmus A hat polynomialen Speicherbedarf, falls es ein Polynom $q : \mathbb{N} \to \mathbb{N}$ gibt mit $s_A(n) \leq q(n)$ für alle $n \in \mathbb{N}$.

Weitere Bemerkungen

Bemerkungen

- Nehmen wir an, wir ermitteln die Rechenzeit $f(n)$ für einen bestimmten Algorithmus. Die Variable n kann z.B. die Anzahl der Ein- und Ausgabewerte sein, ihre Summe, oder auch die Größe eines dieser Werte. Da $f(n)$ maschinenabhängig ist, genügt eine a priori Analyse nicht. Jedoch kann man mit Hilfe einer a priori Analyse ein g bestimmen, so dass $f \in O(g)$.
- Wenn wir sagen, dass ein Algorithmus eine Rechenzeit $O(g)$ hat, dann meinen wir damit folgendes: Wenn der Algorithmus auf unterschiedlichen Computern mit den gleichen Datensätzen läuft, und diese Größe n haben, dann werden die resultierenden Laufzeiten immer kleiner sein als eine Konstante mal $|g(n)|$. Bei der Suche nach der Größenordnung von f werden wir darum bemüht sein, das kleinste g zu finden, so dass $f \in O(g)$ gilt.

Die Klassen $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-vollständig

Die Klasse $\mathcal{P}$

Was ist ein Entscheidungsproblem?

- Problem, das nur zwei mögliche Antworten besitzt, "ja" oder "nein".
- Beispiele: "Ist n eine Primzahl?" oder "Enthält G einen Kreis?"

Die Klasse $\mathcal{P}$: informelle Definition

- Klasse der Entscheidungsprobleme, für die ein polynomialer Lösungsalgorithmus existiert

Die Klasse $\mathcal{P}$: formale Definition

- Gegeben ein Kodierungsschema E und ein Rechnermodell M.
- Π sei ein Entscheidungsproblem, wobei jede Instanz aus Π durch Kodierungsschema E kodierbar sei.
- Π gehört zur Klasse $\mathcal{P}$ (bzgl. E und M), wenn es einen auf M implementierbaren Algorithmus zur Lösung aller Problembeispiele aus Π gibt, dessen Laufzeitfunktion auf M polynomial ist.

Die Klasse $\mathcal{NP}$

Motivation

- Enthält G einen Kreis? → "einfach" → $\mathcal{P}$
- Enthält G einen hamiltonischen Kreis? → "schwieriger" → $\mathcal{NP}$

Die Klasse $\mathcal{NP}$: informelle Definition

- Entscheidungsproblem Π gehört zur Klasse $\mathcal{NP}$, wenn es folgende Eigenschaft hat:
  - Ist die Antwort "ja" für $I \in \Pi$, dann kann Korrektheit dieser Aussage mit Hilfe eines geeigneten Zusatzobjekts in polynomialer Laufzeit überprüft werden.

Beispiel: "Enthält G einen hamiltonischen Kreis?"

- Geeignetes Zusatzobjekt wäre ein hamiltonischer Kreis.
- Es muss nun lediglich geprüft werden, ob der angegebene Kreis tatsächlich ein hamiltonischer Kreis von G ist.

Die Klasse $\mathcal{NP}$ (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Formale Definition

- Entscheidungsproblem Π gehört zu $\mathcal{NP}$, wenn es die folgende Eigenschaften hat:
  (a) Für jedes Problembeispiel $I \in \Pi$, für das die Antwort "ja" lautet, gibt es mindestens ein Objekt Q, mit dessen Hilfe die Korrektheit der "ja"-Antwort überprüft werden kann.
  (b) Es gibt einen Algorithmus, der Problembeispiele $I \in \Pi$ und Zusatzobjekte Q als Input akzeptiert und der in einer Laufzeit, die polynomial in $\langle I \rangle$ ist, überprüft, ob Q ein Objekt ist, aufgrund dessen Existenz eine "ja"-Antwort für I gegeben werden muss.

Bemerkungen

- Es wird keine Aussage über die Berechenbarkeit eines geeigneten Q gemacht. Q kann auch geraten werden.
- Da der Überprüfungsalgorithmus Q lesen muss, muss $\langle Q \rangle$ polynomial in $\langle I \rangle$ sein!
- Es wird keine Aussage zu möglicher "nein"-Antwort gemacht. $\rightarrow$ coNP

Charakterisierung besonders schwieriger Probleme in $\mathcal{NP}$

TECHNISCHE UNIVERSITÄT DARMSTADT

Polynomiale Transformation von Problemen

- Gegeben seien zwei Entscheidungsprobleme Π und Π'.
- Eine polynomiale Transformation von Π in Π' ist ein polynomialer Algorithmus, der, gegeben ein (kodiertes) Problembeispiel $I \in \Pi$, ein (kodiertes) Problembeispiel $I' \in \Pi'$ produziert, so dass folgendes gilt:
  - Die Antwort auf I ist genau dann "ja", wenn die Antwort auf I' "ja" ist.

Die Klasse $\mathcal{NP}$-vollständig

- Ein Entscheidungsproblem Π heißt **$\mathcal{NP}$-vollständig**, falls $\Pi \in \mathcal{NP}$ und falls jedes andere Problem aus $\mathcal{NP}$ polynomial in Π transformiert werden kann.

Bemerkung und Fragen

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkung

- Falls ein $\mathcal{NP}$-vollständiges Entscheidungsproblem Π in polynomialer Zeit gelöst werden kann, so gilt das auch für jedes andere Problem aus $\mathcal{NP}$:

$$\Pi \quad \mathcal{NP} - \text{vollständig und } \Pi \in \mathcal{P} \Rightarrow \mathcal{P} = \mathcal{NP}.$$

- Diese Eigenschaft zeigt, dass (bzgl. polynomialer Lösbarkeit) kein Problem in $\mathcal{NP}$ schwieriger ist als ein $\mathcal{NP}$-vollständiges.

Existieren $\mathcal{NP}$-vollständige Probleme? Ja.

- 3-SAT: $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\ldots) \wedge \ldots = \text{TRUE}$

Offene Fragen

$$\begin{aligned} \mathcal{P} &= \mathcal{NP} \cap \text{co}\mathcal{NP} \\ \mathcal{NP} &= \text{co}\mathcal{NP} \\ \mathcal{P} &\neq \mathcal{NP} \end{aligned}$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexität von Optimierungsproblemen

Die Klasse $\mathcal{NP}$-schwer

TECHNISCHE UNIVERSITÄT DARMSTADT

Transformation in Entscheidungsproblem

- Sei O ein Maximierungsproblem (Minimierungsproblem). Man legt zusätzlich zur Instanz I noch eine Schranke B fest und stellt die Frage:
  - Gibt es für I eine Lösung, deren Wert nicht kleiner (größer) als B ist?
- Damit ist ein Optimierungsproblem polynomial in ein Entscheidungsproblem transformierbar.

Die Klasse $\mathcal{NP}$-schwer: formale Definition

- Ein Optimierungsproblem O heißt **$\mathcal{NP}$-schwer**, falls es ein $\mathcal{NP}$-vollständiges Entscheidungsproblem Π gibt, so dass Π in polynomialer Zeit gelöst werden kann, wenn O in polynomialer Zeit gelöst werden kann.

Beispiele

- Rucksackproblem, Maschinenbelegung, stabile Menge maximaler Kardinalität, Clique maximaler Kardinalität, minimale Knotenüberdeckung

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Traveling Salesman Problem

- Input: Instanz I ist charakterisiert durch eine Distanzmatrix $c_{ij} \in \mathbb{Z}\ \forall i, j \in V$, $|V| = n$.
- Problem: "Finde einen bzgl. c minimalen Hamiltonischen Kreis in K_n."
- TSP-Entscheidungsproblem:
  - Input wie beim TSP, $B \in \mathbb{Z}$.
  - Problem: "Existiert eine TSP-Tour der Länge $\leq B$?"

Lemma

- Polynomialer Algorithmus zur Lösung des TSP-Entscheidungsproblems würde polynomialen Algorithmus zur Lösung des TSP implizieren.

Lösung: Binäre Suche

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis

- Länge eines hamiltonischen Kreises in K_n ist ein ganzzahliger Wert im Intervall $[ns, nt]$, wobei $s = \min\{c_{ij} : i, j \in V, i \neq j\}$ und $t = \max\{c_{ij} : i, j \in V, i \neq j\}$.
- Rufe TSP-Entscheidungsproblem mit $B = \lfloor \frac{n(s+t)}{2} \rfloor$ auf, passe obere Schranke nt oder untere Schranke ns je nach Ergebnis an und iteriere.
- Insgesamt reichen demnach $\log(nt - ns)$ Iterationen aus. $\square$

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenstrukturen zur Kodierung von Graphen

Kanten- und Bogenlisten

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

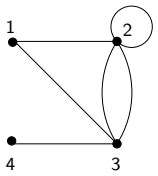
- Ist $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten, so sieht die Kantenliste wie folgt aus:

$$n, m, a_1, e_1, a_2, e_2, \ldots, a_m, e_m,$$

  wobei a_i, e_i die beiden Endknoten der Kante i sind.
- Die Reihenfolge des Aufführens der Endknoten von i bzw. den Kanten ist beliebig.
- Bei Schleifen wird der Endknoten zweimal aufgelistet.
- Bei der Bogenliste eines Digraphen verfahren wir genauso, müssen jedoch darauf achten, dass ein Bogen immer zunächst durch seinen Anfangs- und dann durch seinen Endknoten repräsentiert wird.

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT



Kantenliste

- Dieser Graph hat die folgende Kantenliste:

$$4, 6, 1, 2, 2, 3, 4, 3, 3, 2, 2, 2, 1, 3.$$

- Man beachte, dass hier die Reihenfolge der Knoten beliebig ist, da es sich um ungerichtete Kanten handelt.

Gewichte

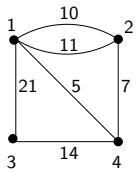
TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

- Haben die Kanten oder Bögen Gewichte, so repräsentiert man eine Kante (einen Bogen) entweder durch Anfangsknoten, Endknoten, Gewicht oder macht eine Kanten- bzw. Bogenliste wie oben und hängt an diese noch eine Liste mit den m Gewichten der Kanten $1, 2, \ldots, m$ an.
- Der Speicheraufwand einer Kanten- bzw. Bogenliste beträgt $2(m + 1)$ Speicherzellen, eine Liste mit Gewichten erfordert $3m + 2$ Zellen.

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT



Kantenliste mit Gewichten

- Dieser gewichtete Graph ist in den beiden folgenden Kantenlisten mit Gewichten gespeichert:
  - 4,6,1,2,10,1,2,11,2,4,7,4,3,14,3,1,21,1,4,5
  - 4,6,1,2,1,2,2,4,3,4,1,4,1,3,11,10,7,14,5,21

Adjazenzmatrix eines ungerichteten Graphen

TECHNISCHE UNIVERSITÄT DARMSTADT

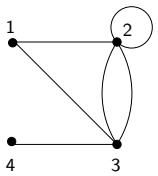
Definition

- Sei $G = (V, E)$ ein ungerichteter Graph mit $V = \{1, \ldots, n\}$, so ist die symmetrische Matrix $A \in \mathbb{R}^{n \times n}$ mit
  - $a_{ij} = a_{ji} =$ Anzahl der Kanten, die i mit j, $i \neq j$, verbinden,
  - $a_{ii} = 2*$ (Anzahl der Schleifen, die i enthalten), $i = 1, \ldots, n$,

  die **Adjazenzmatrix** von G.
- Wegen der Symmetrie genügt es, die obere Dreiecksmatrix von A zu speichern.
- Hat G Kantengewichte und ist G einfach, so setzt man
  - $a_{ij} =$ Gewicht der Kante, wenn $ij \in E$,
  - $a_{ij} = 0, -\infty, +\infty$, anderenfalls.
- Der Speicheraufwand für Adjazenzmatrizen beträgt n^2 Zellen (bzw. $\frac{n(n-1)}{2}$ für obere Dreiecksmatrix).

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT



Adjazenzmatrix

- Die Adjazenzmatrix dieses Graphen ist:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Adjazenzmatrix eines gerichteten Graphen

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

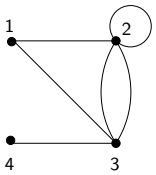
- Die Adjazenzmatrix eines Digraphen $D = (V, A)$ mit $V = \{1, \ldots, n\}$ ohne Schleifen ist definiert durch
  - $a_{ii} = 0$, $i = 1, \ldots, n$,
  - $a_{ij} =$ Anzahl der Bögen in A mit Anfangsknoten i und Endknoten j.
- Bogengewichte einfacher Digraphen werden wie im ungerichteten Fall behandelt.
- Der Speicheraufwand für Adjazenzmatrizen beträgt n^2 Zellen (bzw. $\frac{n(n-1)}{2}$ für obere Dreiecksmatrix).

Adjazenzlisten

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

- Speichert man für einen Graphen $G = (V, E)$ die Anzahl von Knoten und für jeden Knoten seinen Grad und die Namen der Nachbarknoten, so nennt sich eine solche Datenstruktur **Adjazenzliste** von G.



Adjazenzliste

	Knotennummer	Grad	Nachbarknoten
4	1	2	2, 3
	2	4	1, 3, 3, 2
	3	4	1, 2, 2, 4
	4	1	3

- Die Adjazenzliste dieses Graphen ist:

$$4, 1, 2, 2, 3, 2, 4, 1, 3, 3, 2, 3, 4, 1, 2, 2, 4, 4, 1, 3.$$

Bemerkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Adjazenzlisten für Digraphen

- Bei Digraphen geht man analog vor: nur speichert man lediglich die Nachfolger eines Knotens (jeder Bogen kommt also nur einmal vor).
- Die Speicherung der Adjazenzliste eines Graphen erfordert $2n + 1 + 2m$ Speicherplätze, ihre Speicherung bei einem Digraphen erfordert $2n + 1 + m$ Speicherplätze.

Vor- und Nachteil von Adjazenzlisten

- kompakt, "gut für dünn besetzte Graphen",
- aber schlecht: Suchen, ob Kante ij in G enthalten ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Finden von aufspannenden Wäldern

Depth-First Search Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Input/Output

- **Input:** Einfacher Graph $G = (V, E)$ in Form einer Adjazenzliste, d.h. für jeden Knoten v ist eine Nachbarliste $N(v)$ gegeben.
- **Output:** Kantenmenge T (= aufspannender Wald) und Kantenmenge B (mit $B \cup T = E$ und $B \cap T = \emptyset$)

Depth-First Search Algorithmus (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee und Ziel des DFS-Algorithmus

- Berechnet für einen Graphen G:
  - die Anzahl der Komponenten,
  - entscheidet, ob G Kreis enthält und
  - findet für jede Komponente einen aufspannenden Baum.
- Tiefensuche: "so schnell wie möglich einen Knoten verlassen"
- rekursives Hinzufügen von Kanten zum Wald T oder zu kreisschließenden Kanten B
- DFS-Algorithmus teilt die Kanten des Graphen in zwei Teilmengen:
  - Kante xy heißt Vorwärtskante, wenn wir im Verfahren von einem markierten Knoten x entlang xy zum Knoten y gehen und y markieren.
    → Kante des aufspannenden Waldes
  - Anderfalls heißt xy Rückwärtskante.
    → Kante würde Kreis schließen

DFS-Algorithmus im Detail

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithmus

- Alle Knoten seien unmarkiert.
- Setze $T := \emptyset$, $B := \emptyset$.
- Für alle $v \in V$ führe aus
  - Ist v unmarkiert, dann **CALL SEARCH**(v).
  - (**Bemerkung:** Es existieren genau soviele Zusammenhangskomponenten in G wie hier **SEARCH**(v) aufgerufen wird.)
- Gib T und B aus.

Funktion SEARCH(v)

- Markiere v.
- Für alle Knoten $w \in N(v)$ führe aus:
  - Ist w markiert und $vw \notin T$, setze $B := B \cup \{vw\}$.
  - Ist w unmarkiert, setze $T := T \cup \{vw\}$ und **CALL SEARCH**(w).

Eigenschaften des DFS-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeit

- Laufzeit des Verfahrens: $O(|V| + |E|)$, d.h. polynomiell

Korrektheit

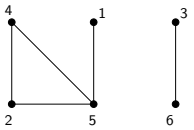
- Die Menge aller Vorwärtskanten ist ein Wald von G, der in jeder Komponente einen aufspannenden Baum bildet.
  → Beweis als Übungsaufgabe
- G enthält einen Kreis $\Leftrightarrow B \neq \emptyset$.

Datenstruktur

- DFS-Algorithmus ist konzipiert für die Datenstruktur einer Adjazenzliste!

Beispiel für DFS-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT



Adjazenzliste (in Tabellenform)

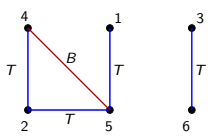
Dieser Graph hat die folgende Adjazenzliste:

	Knotennummer	Grad	Nachbarknoten
6	1	1	5
	2	2	4, 5
	3	1	6
	4	2	2, 5
	5	3	1, 2, 4
	6	1	3

Beispiel für DFS-Algorithmus (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- $1 \to 5 \to 2 \to 4$ $T = \{15, 52, 24\}$
- $4 \to 2$ $(24 \in T)$
- $4 \to 5$ $B = \{45\}$
- Knoten 4 erledigt, zurück zu Knoten 2
- $2 \to 5$ $(52 \in T)$
- Knoten 2 erledigt, zurück zu Knoten 5
- $5 \to 1$ $(15 \in T)$
- Knoten 5 erledigt, zurück zu Knoten 1, Knoten 1 erledigt, Komponente erledigt
- $3 \to 6$ $T = \{15, 52, 24, 36\}$
- Knoten 6 erledigt, zurück zu Knoten 3, Knoten 3 erledigt, Komponente erledigt



TECHNISCHE UNIVERSITÄT DARMSTADT

Finden von gewichts-maximalen Wäldern

Greedy–Max Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Input/Output

- **Input:** Graph $G = (V, E)$ mit Kantengewichten c_e, für alle $e \in E$.
- **Output:** Ein Wald $W \subseteq E$ mit maximalen Gewicht $c(W)$.

Algorithmus

1. (Sortieren)
   $c(e_1) \geq c(e_2) \geq \ldots \geq c(e_K) > 0 \geq c(e_{K+1}) \geq \ldots \geq c(e_{|E|})$.
2. (Initialisierung)
   Setze $W := \emptyset$.
3. ***Für*** $i = 1, \ldots, K$ führe durch:
   Falls $W \cup \{e_i\}$ keinen Kreis enthält,
   setze $W := W \cup \{e_i\}$.
4. (Ausgabe)
   Gib W aus.

TECHNISCHE UNIVERSITÄT DARMSTADT

Finden von kürzesten Wegen in Graphen

Problem

TECHNISCHE UNIVERSITÄT DARMSTADT

Problemformulierung

- Gegeben sei ein Digraph $D = (V, A)$ mit Bogengewichten $c(a)$ für alle $a \in A$. Weiterhin seien $s, t \in V$ gegeben.
- Gesucht ist ein s-t-Weg minimalen Gesamtgewichts (= Summe der Kantengewichte).

Schwierigkeiten

- negative Gewichte
- Hamiltonische-Wege Problem (jeder Knoten genau einmal besucht)

Unterscheidung der Algorithmen

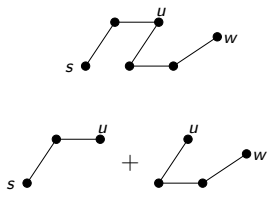
- nur nichtnegative Gewichte
- auch negative Gewichte möglich

Ein Startknoten, nichtnegative Gewichte

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee

- Dekomposition eine kürzesten (s, w)-Weges in einen kürzesten (s, u)-Weg plus kürzesten (u, w)-Weg.



Dijkstra-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Input/Output

- **Input:** Digraph $D = (V, A)$, Gewichte $c(a) \geq 0$ für alle $a \in A$, ein Knoten $s \in V$ (und ein Knoten $t \in V \setminus \{s\}$).
- **Output:** Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge (bzw. einfach nur ein kürzester (s, t)-Weg).

Datenstrukturen

$$\begin{array}{lcl} \mathrm{DIST}(v) & = & \text{Länge des kürzesten } (s,v)\text{-Weges} \\ \mathrm{VOR}(v) & = & \text{Vorgänger von } v \text{ im kürzesten } (s,v)\text{-Weg} \end{array}$$

Initialisierung

$$\begin{array}{lcl} \mathrm{DIST}(s) & = & 0 \\ \mathrm{DIST}(v) & = & +\infty \text{ für alle } v \in V \setminus \{s\} \\ \mathrm{VOR}(v) & = & s \text{ für alle } v \in V \end{array}$$

Dijkstra-Algorithmus im Detail

TECHNISCHE UNIVERSITÄT DARMSTADT

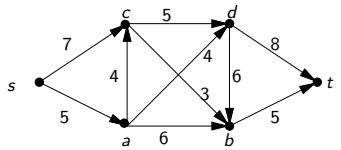
Algorithmus

1. Alle Knoten seien unmarkiert.
2. Sei u unmarkierter Knoten mit $\mathrm{DIST}(u) = \min\{\mathrm{DIST}(v) : v \text{ unmarkiert}\}$. Markiere u. (Falls $u = t$, breche ab.)
3. Für alle unmarkierten Knoten v mit $uv \in A$ führe aus:
   $$\begin{array}{rcl} \text{Falls } \mathrm{DIST}(v) & > & \mathrm{DIST}(u) + c(uv) \text{ setze:} \\ \mathrm{DIST}(v) & := & \mathrm{DIST}(u) + c(uv) \text{ und } \mathrm{VOR}(v) := u. \end{array}$$
4. Sind noch nicht alle Knoten markiert, gehe zu 2.

- Für alle markierten Knoten v ist $\mathrm{DIST}(v)$ die Länge eines kürzesten (s, v)-Weges.
- Ist v markiert und $\mathrm{DIST}(v) < +\infty$, so ist $\mathrm{VOR}(v)$ der Vorgänger von v in einem kürzesten (s, v)-Weg. Durch Rückwärtsgehen bis s kann ein kürzester (s, v)-Weg bestimmt werden.
- Brechen wir das Verfahren nicht in Schritt 2 ab und gilt am Ende $\mathrm{DIST}(v) = +\infty$, so heißt das, dass es in D keinen (s, v)-Weg gibt.

Beispiel für Dijkstra-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT



	DIST(*s*)	*DIST*(*a*)	*DIST*(*b*)	*DIST*(*c*)	*DIST*(*d*)	*DIST*(*t*)	*Unmarkiert*
Start	0	∞	∞	∞	∞	∞	$\{s, a, b, c, d, t\}$
Mark. s	0	5*s*	∞*s*	7*s*	∞*s*	∞*s*	$\{a, b, c, d, t\}$
Mark. a	0	5	11*a*	7	9*a*	∞	$\{b, c, d, t\}$
Mark. c	0	5	10*c*	7	9	∞	$\{b, d, t\}$
Mark. d	0	5	10	7	9	17*d*	$\{b, t\}$
Mark. b	0	5	10	7	9	15*b*	$\{t\}$
(Mark. t)							$\emptyset$

Demnach ist $s \to c \to b \to t$ **ein** kürzester Weg von s nach t und hat die Länge 15.

Korrektheit des Dijkstra-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Der Dijkstra-Algorithmus arbeitet korrekt.

Beweis

- Über Induktion nach der Anzahl k der markierten Knoten zeigen wir:
  - (a) Ist v markiert, so enthält DIST(v) die Länge eines kürzesten (s, v)-Weges.
  - (b) Ist v unmarkiert, so enthält DIST(v) die Länge eines kürzesten (s, v)-Weges, wobei nur markierte Knoten als innere Knoten zugelassen sind.
- $k = 1$: Dann ist nur s markiert und die Behauptung gilt sicherlich.
- Sei die Behauptung also richtig für $k \geq 1$ markierte Knoten.

Induktionsschritt für (a)

TECHNISCHE UNIVERSITÄT DARMSTADT

Induktionsschritt für (a), Teil 1

- Das Verfahren habe in Schritt 2 einen $(k+1)$-sten Knoten u markiert.
- Nach Ind.-voraussetzung Teil (b) ist DIST(u) die Länge eines kürzesten (s, u)-Weges, der als innere Knoten nur die ersten k markierte Knoten benutzen darf.
- Wir zeigen, dass DIST(u) die Länge eines kürzesten (s, u)-Weges ist.

Induktionsschritt für (a)

TECHNISCHE UNIVERSITÄT DARMSTADT

Induktionsschritt für (a), Teil 2

- Wir zeigen, dass DIST(u) die Länge eines kürzesten (s, u)-Weges ist.
- Angenommen, es existiere ein kürzerer Weg P von s nach u.
  - Dann muß er einen Bogen von einem markierten Knoten zu einem unmarkierten Knoten enthalten.
  - Sei vw der erste derartige Bogen auf dem Weg P.
  - Der Teilweg $\bar{P}$ von P von s nach w benutzt nur markierte Knoten als innere Knoten.
  - Deshalb gilt DIST(w) $\leq c(\bar{P})$ nach Ind.-voraussetzung Teil (b).
  - Weiterhin gilt
    - $c(\bar{P}) \leq c(P)$ (nichtnegative Kantengewichte) und
    - $c(P) <$ DIST(u) (Annahme über P).
  - Zusammen ergibt sich: DIST(w) $<$ DIST(u).
  - Aber dies widerspricht DIST(u) $\leq$ DIST(w) entsprechend der Wahl von u in Schritt 2.

Induktionsschritt für (b)

TECHNISCHE UNIVERSITÄT DARMSTADT

Induktionsschritt für (b)

- Es bleibt zu zeigen: Für die derzeit unmarkierten Knoten v ist DIST(v) die Länge eines kürzesten (s, v)-Weges, der nur markierte Knoten als innere Knoten hat.
- Der vorletzte Knoten auf einem kürzesten (s, v)-Weg ist entweder u oder ein anderer (vor u) markierter Knoten.
- Man beachte jetzt, dass in Schritt 3 die Länge eines (s, v)-Weges über markierte Knoten verschieden von u verglichen wird mit der Länge eines (s, v)-Weges über markierte Knoten, der als vorletzten Knoten den Knoten u enthält.
- Damit folgt die Behauptung. □

Bemerkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Kürzeste-Wege-Baum

- In der Datenstruktur VOR merken wir uns zu jedem Knoten v seinen Vorgänger in einem kürzesten (s, v)-Weg.
- Einen kürzesten (s, v)-Weg erhält man dann gemäß:

$$v \quad \text{VOR}(v) \quad \text{VOR}(\text{VOR}(v)) \quad \ldots \quad \text{VOR}(\ldots \text{VOR}(v) \ldots) = s$$

- Offensichtlich ist durch VOR( ) eine Arboreszenz definiert.

Satz

- Sei $D = (V, A)$ ein Digraph mit nichtnegativen Bogengewichten und $s \in V$.
- Es gibt eine Arboreszenz B mit Wurzel s, so dass für jeden Knoten $v \in V$, für den es einen (s, v)-Weg in D gibt, der (eindeutig bestimmte) gerichtete Weg in B von s nach v ein kürzester (s, v)-Weg ist.

Anzahl Operationen des Dijkstra-Algorithmus: $O(|V|^2)$

Ein Startknoten, beliebige Gewichte

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem

- Problem, einen kürzesten Weg in einem Digraphen mit beliebigen Bogengewichten zu bestimmen, ist äquivalent zum Problem, einen längsten Weg in einem Digraphen mit beliebigen Gewichten zu finden.
- Wäre dies in polynomialer Zeit lösbar, so könnte man auch das $\mathcal{NP}$-vollständige Problem in polynomialer Zeit lösen, zu entscheiden, ob ein Digraph einen gerichteten hamiltonischen Weg hat.

Vergrößerung der Problemklasse

- Dijkstra-Algorithmus funktioniert nur für nichtnegative Gewichte.
- Betrachte jetzt **azyklische** Digraphen mit **beliebigen** Kantengewichten.

Moore-Bellman-Algorithmus für azyklische Digraphen

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz, Beweis → Übungsaufgabe

- Sei $D = (V, A)$ ein azyklischer Digraph.
- Dann können die Knoten in V so numeriert werden, dass $(u, v) \in A \Rightarrow u < v$.

Input/Output

- **Input:** Azyklischer Digraph $D = (V, A)$, Gewichte $c(a)$ für alle $a \in A$ (auch negative Gewichte sind zugelassen), ein Knoten $s \in V$.

  O.B.d.A. nehmen wir an, dass $V = \{1, 2, \ldots, n\}$ gilt und alle Bögen die Form (u, v) mit $u < v$ haben.
- **Output:** Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge

Datenstrukturen

DIST(v) = Länge des kürzesten (s, v)-Weges
VOR(v) = Vorgänger von v im kürzesten (s, v)-Weg

Moore-Bellman-Algorithmus im Detail

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithmus

DIST(s) = 0
DIST(v) = $+\infty$ für alle $v \in V \setminus \{s\}$
VOR(v) = s für alle $v \in V$

FOR $v = s + 1$ TO n DO
  FOR $v = s$ TO $v - 1$ DO
    Falls $(u, v) \in A$ und DIST(u) + $c(uv)$ <DIST(v)
    setze DIST(v) :=DIST(u) + $c(uv)$ und VOR(v) := u.

- Gilt am Ende DIST(v) = $+\infty$, so heißt das, dass es in D keinen (s, v)-Weg gibt.
- Für alle Knoten v mit DIST(v) $< +\infty$ ist DIST(v) die Länge eines kürzesten (s, v)-Weges.
- Ist DIST(v) $< +\infty$, so ist VOR(v) Vorgänger von v in einem kürzesten (s, v)-Weg. Durch Rückwärtsgehen bis s kann ein kürzester (s, v)-Weg bestimmt werden.

Anzahl Operationen des Moore-Bellman-Algorithmus: $O(|V|^2)$

Korrektheit des Moore-Bellman-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Der Moore-Bellman-Algorithmus funktioniert für beliebige azyklische Digraphen mit beliebigen Gewichten.

Beweis

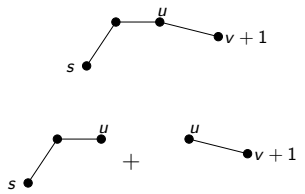
- Nach Voraussetzung haben alle Bögen in D die Form (u, v) mit $u < v$.
- Deshalb enthält jeder (s, v)-Weg mit $v > s$ als innere Knoten nur solche Knoten u mit $s < u < v$.
- Durch Induktion nach $v = s, \ldots, n$ zeigen wir, dass DIST(v) die Länge eines kürzesten (s, v)-Weges ist.
- $v = s$ o.k.
- Sei die Behauptung richtig für alle Knoten $s, \ldots, v$.

Induktionsschritt

TECHNISCHE UNIVERSITÄT DARMSTADT

Induktionsschritt

- Betrachte den Knoten $v + 1$.
- Ein kürzester $s, v + 1$-Weg besteht entweder nur aus dem Bogen $(s, v + 1)$ (falls vorhanden) oder aber er führt zu einem Knoten $s + 1 \leq u \leq v$ und benutzt anschließend die Kante $u, v + 1$.
- Das Minimum all dieser Wege wird jedoch in der innersten Schleife berechnet. □



Yen-Variante

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

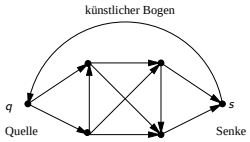
- Variante des Moore-Bellman-Algorithmus, welche für beliebige Gewichte und beliebige Digraphen funktioniert, sofern es keinen gerichteten **negativen** Kreis gibt.
- Laufzeit $O(|V|^3)$
- **Wichtig:** Mit dem Verfahren kann man tatsächlich testen, ob D einen negativen gerichteten Kreis enthält! Ein solcher wird in **polynomialer Zeit** gefunden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Finden maximaler Flüsse in Netzwerken

Maximalfluss-Problem

TECHNISCHE UNIVERSITÄT DARMSTADT



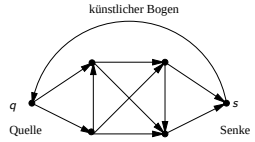
Problemformulierung

- **Gegeben** sei ein Digraph $D = (V, A)$ mit Bogenkapazitäten d_a für alle $a \in A$. Weiterhin mögen ein Knoten $q \in V$ ohne eingehende Bögen (**Quelle**) und ein Knoten $s \in V$ ohne abgehende Bögen (**Senke**) existieren. Letztendlich setzen wir $\bar{A} = A \cup \{sq\}$ und wählen $d_{sq} \geq \sum_{a \in A} d_a$ beliebig.
- Ein Vektor $x = (x_a : a \in \bar{A})$ heißt **Fluß** auf D, falls für jeden Knoten $v \in V$ gilt

$$\sum_{wv \in \bar{A}} x_{wv} = \sum_{vw \in \bar{A}} x_{vw} \quad \text{(Flusserhaltung)}.$$

Maximalfluss-Problem (2)

TECHNISCHE UNIVERSITÄT DARMSTADT



Problemformulierung (2)

- Der **Wert** des Flusses ist gleich x_{sq}.
- **Problemstellung:** Man finde einen Fluss $x \geq 0$ über D maximalen Wertes, der auch die Kapazitäten auf den Bögen respektiert, also $x_a \leq d(a)$ für alle $a \in \bar{A}$ erfüllt.

Max-Flow-Min-Cut Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

Sei $S \subseteq V$ mit $q \in S$ und $s \in V \setminus S$. Dann heißt S Schnitt mit Kapzität

$$\text{cap}(S) = \sum_{vw \in A, v \in S, w \in V \setminus S} d_{vw}.$$

Minimaler Schnitt

- Kapazität eines jeden Schnittes liefert obere Schranke für maximalen Fluss.
- Finde Schnitt minimaler Kapazität.

Max-Flow-Min-Cut Theorem

- Es existiert ein gültiger Fluss $\bar{x} \leq d$ und ein Schnitt $S \subseteq V$ mit $\bar{x}_{sq} = \text{cap}(S)$.
- Beide sind Optimallösungen der jeweiligen Probleme.

→ Dualitätstheorie in linearer Optimierung

Formulierung als lineares Optimierungsproblem

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Für einen Digraphen $D = (V, A)$ ist die Knoten-Kanten-Inzidenzmatrix definiert durch:

- $M_{ij} = 1$, falls Bogen j Knoten i verlässt,
- $M_{ij} = -1$, falls Bogen j in Knoten i endet,
- $M_{ij} = 0$, sonst.

Lemma → Max-Flow Problem als lineares Optimierungsproblem

$x \geq 0$ ist ein Fluss, genau dann wenn $Mx = 0$. → $\min\{x_{sq} : Mx = 0, 0 \leq x \leq d\}$

Beweis → Betrachte Knoten i

$$\text{(eingehende Bögen)} \sum_{vi \in \bar{A}} x_{vi} = \sum_{iw \in \bar{A}} x_{iw} \text{ (ausgehende Bögen)}$$

$$0 = \sum_{iw \in \bar{A}} 1 \cdot x_{iw} + \sum_{vi \in \bar{A}} (-1) \cdot x_{vi} + \sum_{vw \in \bar{A}, v, w \neq i} 0 \cdot x_{vw} = M_{i.}x$$

Lösung des Max-Flow Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee des Ford-Fulkerson Algorithmus

Verbessere Fluss sukzessive durch **augmentierende Pfade**.

Definition

- Sei $D = (V, A)$ ein Digraph mit $q, s \in V$ und $qs \in \bar{A}$, d ein Kapazitätsvektor und x ein gültiger Fluss für d.
- Ein (nicht notwendigerweise gerichteter) Pfad $q = v_0 e_1 v_1 e_2 v_2 \ldots e_k v_k$ heißt **ungesättigt** (bzgl. x und d), falls
  - $e_i \in A$,
  - $x_{e_i} < d_{e_i}$ für alle Vorwärtsbögen,
  - $x_{e_j} > 0$ für alle Rückwärtsbögen,
- Falls $v_0 = q$ und $v_k = s$, so heißt dieser Pfad ein **augmentierender Pfad**.

Ford-Fulkerson Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Konstruktion eines besseren Flusses aus augmentierendem Pfad

- Sei $\epsilon = \min_i\{d_{e_i} - x_{e_i}, x_{e_i}\}$.
- Besserer Fluss als x:

$$x_e = \begin{cases} x_e + \epsilon & \text{falls } e = sq, \\ x_e + \epsilon & \text{falls } e \text{ Vorwärtsbogen im Pfad ist}, \\ x_e - \epsilon & \text{falls } e \text{ Rückwärtsbogen im Pfad ist}, \\ x_e & \text{falls } \textit{sonst}. \end{cases}$$

Ford-Fulkerson Algorithmus

- Initialisiere $x = 0$.
- Solange ein augmentierender Pfad existiert, verbessere x.

Wie findet man augmentierenden Pfad?

TECHNISCHE UNIVERSITÄT DARMSTADT

Hilfsgraph $D' = (V', A')$

- $V' = V$
- $vw \in A'$ mit Kapazität $d_{vw} - x_{vw}$, falls $d_{vw} - x_{vw} > 0$
- $wv \in A'$ mit Kapazität x_{vw}, falls $x_{vw} > 0$

Bemerkung

- Für jeden Bogen in D sind im Hilfsgraphen D' ein bzw. zwei Bögen jenachdem, ob $d_{vw} - x_{vw} > 0$ und $x_{vw} > 0$ gilt.
- Jeder augmentierende Pfad in D (bzgl. x und d) entspricht einem gerichteten q-s-Weg in D'.
- Ob ein gerichteter q-s-Weg in D' existiert, kann mit dem DFS-Algorithmus entschieden werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren von Arrays

- InsertionSort (schon behandelt)
- MergeSort
- QuickSort
- HeapSort

Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

Datentyp

```
struct Item {
    int key;
    // data components
};
```

Beispiel

- Immatrikulationsnummer
  - Name des Studenten, Kontaktdaten des Studenten, Prüfungsergebnisse

Schlüsselwahl → Wahl ganzer Zahlen als Schlüssel ist willkürlich

- Man benötigt Menge mit Ordnungsrelation, es gilt also $a < b$, $a = b$ oder $a > b$ für alle a, b.
  - z.B. Name des Studenten (lexikographische Ordnung wie im Duden)

MergeSort

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee

- Mergesort teilt das zu sortierende Array in zwei gleichgroße Teilarrays,
- sortiert diese (durch rekursive Anwendung von Mergesort auf die beiden Teile) und
- fügt die sortierten Teile zusammen.

MergeSort (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfügen zweier sortierter Arrays: Algorithmus MERGE

- Seien dazu v_1 und v_2 bereits sortierte Arrays der Länge m bzw. n mit Komponenten vom Typ Item.
- Diese sollen in das Array v der Länge $m + n$ verschmolzen werden.
- Dazu durchlaufen wir v_1 und v_2 von links nach rechts mit zwei Indexzeigern i und j:

1. Initialisierung: $i = 0$, $j = 0$, $k = 0$;
2. Wiederhole Schritt 3 bis $i = m$ oder $j = n$.
3. Falls $v_1[i]$.key$< v_2[i]$.key, so kopiere $v_1[i]$ an Position k von v und erhöhe i und k um 1.

   Anderenfalls kopiere $v_2[i]$ an Position k von v und erhöhe i und k um 1.
4. Ist $i = m$ und $j < n$, so übertrage die restlichen Komponenten von v_2 nach v.
5. Ist $j = n$ und $i < m$, so übertrage die restlichen Komponenten von v_1 nach v.

MergeSort (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von MERGE

Bei jedem Wiedereintritt in die Schleife 3 gilt die Invariante

$$\begin{array}{lllllll} v[0].\text{key} & \leq \ldots \leq & v[k-1].\text{key} & & & & \\ & & v[k-1].\text{key} & \leq & v_1[i].\text{key} & \leq \ldots \leq & v_1[m-1].\text{key} \\ & & v[k-1].\text{key} & \leq & v_2[j].\text{key} & \leq \ldots \leq & v_2[n-1].\text{key} \end{array}$$

Hieraus folgt sofort, dass v am Ende aufsteigend sortiert ist.

MergeSort (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel

Betrachten wir die Arrays $v_1 = [12, 24, 53, 63]$ und $v_2 = [18, 25, 44, 72]$. Die dazugehörige Folge der Werte von i, j, k, und v bei jedem Wiedereintritt in die Schleife 3 ist in der nachfolgenden Tabelle angegeben. Am Ende der Schleife ist $i = 4$ und Schritt 4 des Algorithmus wird ausgeführt, d.h. der Rest von v_2, also die 72, wird nach v übertragen.

i	j	k	$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	$v[6]$	$v[7]$
1	0	1	12	–	–	–	–	–	–	–
1	1	2	12	18	–	–	–	–	–	–
2	1	3	12	18	24	–	–	–	–	–
2	2	4	12	18	24	35	–	–	–	–
2	3	5	12	18	24	35	44	–	–	–
3	3	6	12	18	24	35	44	53	–	–
4	3	7	12	18	24	35	44	53	63	–

MergeSort (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

- Sei $C(m, n)$ die maximale Anzahl von Schlüsselvergleichen und
- $A(m, n)$ die maximale Anzahl von Zuweisungen von Komponenten beim Zusammenfügen.

Komplexität des Zusammenfügens

- Vergleiche treten nur in der Schleife 3 auf, und zwar genau einer pro Durchlauf.
- Da die Schleife maximal $n + m - 1$ mal durchlaufen wird, gilt

$$C(m, n) \leq m + n - 1.$$

MergeSort (6)

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithmus MERGESORT

```
void MergeSort(Item v[], int first, int last) {
    int middle;
    if (first < last) {
        middle = (first+last)/2;          teile v in 2 gleiche Teile
        MergeSort(v,first,middle);        sortiere ersten Teil
        MergeSort(v,middle+1,last);       sortiere zweiten Teil
        Merge(v,first,middle,last);       füge beide sortierte Teile zusammen
    }
}
```

Sortieren eines Arrays

Der Aufruf MergeSort$(a, 0, n-1)$ sortiert das Array a der Länge n im gesamten Bereich von 0 bis $n-1$.

MergeSort (7)

TECHNISCHE UNIVERSITÄT DARMSTADT

Korrektheit von MERGESORT

Die Korrektheit von MergeSort ergibt sich sofort durch vollständige Induktion nach der Anzahl n =last−first+1 der zu sortierenden Komponenten.

- Ist $n = 1$, also last=first (Induktionsanfang), so wird im Rumpf von MergeSort nichts gemacht und das Array v ist nach Abarbeitung von MergeSort trivialerweise im Bereich first..last sortiert.
- Ist $n > 1$, so sind first..middle und middle+1..last Bereiche mit weniger als n Elementen, die also nach Induktionsvoraussetzung durch die Aufrufe MergeSort(v,first,middle) und MergeSort(v,middle+1,last) korrekt sortiert werden.
- Die Korrektheit von Merge ergibt dann die Korrektheit von MergeSort.

MergeSort (8)

TECHNISCHE UNIVERSITÄT DARMSTADT

Für $a = [63, 24, 12, 53, 72, 18, 44, 35]$ ergibt der Aufruf MergeSort(a,0,7) den in der folgenden Tabelle dargestellten Ablauf. Dabei beschreiben die Einrücktiefe die Aufrufhierarchie (Rekursionsbaum), und die Kästen die bereits sortieren Teile des Arrays.

MergeSort(a,0,7)	63	24	12	53	72	18	44	35
MergeSort(a,0,3)	63	24	12	53	72	18	44	35
MergeSort(a,0,1)	63	24	12	53	72	18	44	35
MergeSort(a,0,0)	[63]	24	12	53	72	18	44	35
MergeSort(a,1,1)	[63]	[24]	12	53	72	18	44	35
Merge(a,0,0,1)	24	63	12	53	72	18	44	35
MergeSort(a,2,3)	24	63	12	53	72	18	44	35
MergeSort(a,2,2)	24	63	[12]	53	72	18	44	35
MergeSort(a,3,3)	24	63	[12]	[53]	72	18	44	35
Merge(a,2,2,3)	24	63	12	53	72	18	44	35
Merge(a,0,1,3)	12	24	53	63	72	18	44	35
MergeSort(a,4,7)	12	24	53	63	72	18	44	35
MergeSort(a,4,5)	12	24	53	63	72	18	44	35
MergeSort(a,4,4)	12	24	53	63	[72]	18	44	35
MergeSort(a,5,5)	12	24	53	63	[72]	[18]	44	35
Merge(a,4,4,5)	12	24	53	63	18	72	44	35
MergeSort(a,6,7)	12	24	53	63	18	72	44	35
MergeSort(a,6,6)	12	24	53	63	18	72	[44]	35
MergeSort(a,7,7)	12	24	53	63	18	72	[44]	[35]
Merge(a,6,6,7)	12	24	53	63	18	72	35	44
Merge(a,4,5,7)	12	24	53	63	18	35	44	72
Merge(a,0,3,7)	12	18	24	35	44	53	63	72

Worst-Case Analyse von MergeSort

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursionsgleichung

- Wir ermitteln den worst-case Aufwand $C(n)$ für Anzahl der Vergleiche und $A(n)$ für Anzahl der Zuweisungen von Mergesort beim Sortieren eines n-Arrays.
- Aus dem rekursiven Aufbau des Algorithmus ergeben sich folgende Rekursionsgleichungen für $C(n)$:

$$\begin{aligned} C(2) &= 1 \\ C(2n) &= 2 \cdot C(n) + C(n, n) \quad \text{für } n > 1. \\ &= 2 \cdot C(n) + 2n - 1 \qquad (*) \end{aligned}$$

- Das Sortieren eines 2-elementigen Arrays erfordert einen Vergleich.
- Das Sortieren eines Arrays der Länge $2n$ erfordert den Aufwand für das Sortieren von 2 Arrays der Länge n (rekursive Aufrufe von MergeSort für die beiden Teile), also $2 \cdot C(n)$, plus den Aufwand $C(n, n)$ für das Zusammenfügen (Aufruf von Merge).

Worst-Case Analyse von MergeSort (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Lemma

Für $n = 2^q$ hat die Rekursionsgleichung (*) die Lösung

$$C(2^q) = (q-1)2^q + 1.$$

Beweis durch Induktion nach q

- Ist $q = 1$, so ist $C(2^1) = 1$ und $(q-1)2^q + 1 = 1$ (Induktionsanfang).
- Also sei die Behauptung richtig für 2^r mit $1 \leq r \leq q$.
- Wir schließen jetzt auf $q+1$:

$$\begin{aligned} C(2^{q+1}) &= 2 \cdot C(2^q) + 2 \cdot 2^q - 1 \quad \text{Rekursionsgleichung} \\ &= 2 \cdot [(q-1)2^q + 1] + 2 \cdot 2^q - 1 \quad \text{Induktionsvoraussetzung} \\ &= (q-1)2^{q+1} + 2 + 2^{q+1} - 1 \\ &= q \cdot 2^{q+1} + 1 \end{aligned}$$

Worst-Case Analyse von MergeSort (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Worst-Case Analyse der Zuweisungen

- Bezüglich der Anzahl $A(n)$ der Zuweisungen von Arraykomponenten ergibt sich analog:

$$A(2n) = 2 \cdot A(n) + \text{ Zuweisungen in Merge}$$

- In Merge werden zunächst die Teile von v nach v_1 und v_2 kopiert.
- Dies erfordert $2n$ Zuweisungen.
- Für das Mergen sind dann wieder $A(n, n) = 2n$ Zuweisungen erforderlich.
- Also ergibt sich die Rekursionsgleichung

$$\begin{aligned} A(2) &= 4 \\ A(2n) &= 2 \cdot A(n) + 4n \quad \text{für } n > 1. \end{aligned}$$

- Der gleiche Lösungsansatz liefert für $n = 2^q$:

$$A(n) = (q+1)2^q.$$

Worst-Case Analyse von MergeSort (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Mergesort sortiert ein Array mit n Komponenten mit $O(n \log n)$ Vergleichen und $O(n \log n)$ Zuweisungen.

Beweis

Sei $2^{q-1} < n \leq 2^q$. Dann gilt:

$$\begin{aligned} C(n) &\leq C(2^q) = (q-1)2^q + 1 \\ &< \log_2 n \cdot 2^q + 1 \\ &< (\log_2 n) \cdot 2n + 1 \\ &= 2n \log_2 n + 1 \in O(n \log n) \\ A(n) &\leq A(2^q) = (q+1)2^q \\ &< (\log_2 n + 2)2^q \\ &< (\log_2 n + 2) \cdot 2n \\ &= 2n \log_2 n + 4n \in O(n \log n) \end{aligned}$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Beschleunigung durch Aufteilung: Divide-and-Conquer

Aufteilungs-Beschleunigungssatz

TECHNISCHE UNIVERSITÄT DARMSTADT

Gegeben

- Problem der Größe $a \cdot n$ mit der Laufzeit $f(a \cdot n)$.
- Dieses zerlegt man in b Teilprobleme der Laufzeit $f(n)$.
- Ist die Laufzeit für das Aufteilen respektive Zusammenfügen der Teillösungen $c \cdot n$, so ergibt sich die folgende Rekursionsgleichung und der folgende Satz.

Satz

Seien $a > 0$, b, c natürliche Zahlen und sei folgende Rekursionsgleichung gegeben:

$$\begin{aligned} f(1) &= c/a \\ f(a \cdot n) &= b \cdot f(n) + c \cdot n \quad \text{für } n = a^q, q > 1. \end{aligned}$$

Dann gilt

$$f(n) \in \begin{cases} O(n) & \text{, falls } a > b \\ O(n \log n) & \text{, falls } a = b \\ O(n^{\log_a b}) & \text{, falls } a < b \end{cases}$$

Beweis

TECHNISCHE UNIVERSITÄT DARMSTADT

Für $n = a^q$ gilt

$$f(n) = \frac{c}{a} n \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i.$$

Dies zeigt man durch Induktion über q. Für $q = 0$ ist die Summe 0 und daher $f(1) = \frac{c}{a}$.
Die Behauptung sei nun für q gezeigt.
Dann ergibt sich im Induktionsschluss auf $q+1$:

Beweis (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

$$\begin{aligned} f(a^{q+1}) &= f(a \cdot a^q) \\ &= b \cdot f(a^q) + c \cdot a^q \quad \text{Rekursionsgleichung} \\ &= b \cdot \frac{c}{a} \cdot a^q \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i + c \cdot a^q \quad \text{Induktionsvoraussetzung} \\ &= \frac{c}{a} \cdot a^{q+1} \frac{b}{a} \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i + \frac{c}{a} \cdot a^{q+1} \\ &= \frac{c}{a} \cdot a^{q+1} \sum_{i=0}^{q} \left(\frac{b}{a}\right)^{i+1} + \frac{c}{a} \cdot a^{q+1} \\ &= \frac{c}{a} \cdot a^{q+1} \left(\sum_{i=1}^{q} \left(\frac{b}{a}\right)^i + 1 \right) \\ &= \frac{c}{a} \cdot a^{q+1} \sum_{i=0}^{q+1} \left(\frac{b}{a}\right)^i \end{aligned}$$

Beweis (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Also gilt

$$f(n) = \frac{c}{a} n \sum_{i=0}^{\log_a n} \left(\frac{b}{a}\right)^i.$$

Wir betrachten jetzt 3 Fälle:

Fall 1: $a > b$

Dann ist

$$\frac{b}{a} < 1 \Rightarrow \sum_{i=0}^{\log_a b} \left(\frac{b}{a}\right)^i < \sum_{i=0}^{\infty} \left(\frac{b}{a}\right)^i.$$

Die letzte Summe ist eine geometrische Reihe mit Wert $k_1 = \frac{1}{1-\frac{b}{a}} = \frac{a}{a-b}$. Also ist

$$f(n) < \frac{c \cdot k_1}{a} n \Rightarrow f(n) \in O(n).$$

Beweis (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Fall 2: $a = b$

Dann ist

$$f(n) = \frac{c}{a} n \sum_{i=0}^{\log_a n} 1 = \frac{c}{a} n (\log_a n + 1) = \frac{c}{a} n \log_a n + \frac{c}{a} n$$

Für $n \geq a$ ist $\log_a n \geq 1$ und daher

$$f(n) \leq \frac{c}{a} n \log_a n + \frac{c}{a} n \log_a n = 2\frac{c}{a} n \log_a n = \left(\frac{2c}{a} \log_a 2\right) \cdot n \log_2 n \in O(n \log_2 n).$$

Offenbar ist dieser Fall gerade der auf Mergesort zutreffende Fall.

Beweis (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Fall 3: $a < b$

$$\begin{aligned} f(n) = \frac{c}{a} n \sum_{i=0}^{\log_a n} \left(\frac{b}{a}\right)^i &= \frac{c}{a} a^q \sum_{i=0}^{q} \left(\frac{b}{a}\right)^i \quad \text{da } n = a^q \\ &= \frac{c}{a} \sum_{i=0}^{q} b^i a^{q-i} = \frac{c}{a} \sum_{i=0}^{q} b^{q-i} a^i = \frac{c}{a} b^q \sum_{i=0}^{q} \left(\frac{a}{b}\right)^i \\ &< \frac{c}{a} b^q \sum_{i=0}^{\infty} \left(\frac{a}{b}\right)^i. \end{aligned}$$

Wie im Fall 1 ist $\sum_{i=0}^{\infty} \left(\frac{a}{b}\right)^i$ eine geometrische Reihe mit Wert $k_2 = \frac{b}{b-a}$. Also ist

$$f(n) < \frac{ck_2}{a} b^q = \frac{ck_2}{a} b^{\log_a n} = \frac{ck_2}{a} n^{\log_a b} \in O(n^{\log_a b}).$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Multiplikation von Dualzahlen

Multiplikation von Dualzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Laufzeit

- Traditionelle Methode erfordert $\Theta(n^2)$ Bit-Operationen.
- Aufteilung und Beschleunigung erreicht $O(n^{\log_2 3}) \in O(n^{1,59})$ Operationen.

Idee (ohne viel Nachdenken)

- Seien x, y zwei n-stellige Dualzahlen, wobei n eine Zweierpotenz sei.
- Wir teilen x, y in zwei $\frac{n}{2}$-stellige Zahlen $x = a2^{n/2} + b$ und $y = c2^{n/2} + d$.
- Dann ist $xy = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$.
- Man hat die Multiplikation also auf 4 Multiplikationen von $\frac{n}{2}$-stelligen Zahlen und einige Additionen und Shifts (Multiplikationen mit $2^{n/2}$ bzw. 2^n), die nur linearen Aufwand erfordern, zurückgeführt.
- Dies führt zur Rekursionsgleichung $T(n) = 4T(n/2) + c_0 n$ mit Lösung $T(n) \in \Theta(n^2)$, also ohne Gewinn gegenüber der traditionellen Methode.

Multiplikation von Dualzahlen (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee (mit etwas mehr Nachdenken)

- Die Anweisungen

$$\begin{aligned} u &:= (a+b)(c+d) \\ v &:= ac \\ w &:= bd \\ z &:= v2^n + (u - v - w)2^{n/2} + w \end{aligned}$$

führen jedoch zur Berechnung von $z = xy$ mit 3 Multiplikationen von Zahlen der Länge $\frac{n}{2}$ bzw. $\frac{n}{2} + 1$, da bei $a + b$ bzw. $c + d$ ein Übertrag auf die $(\frac{n}{2} + 1)$-te Position entstehen könnte.

- Ignorieren wir diesen Übertrag, so erhält man

$$T(n) = 3T(n/2) + c_1 n$$

mit der gewünschten Lösung $T(n) \in \Theta(n^{\log_2 3})$.

Multiplikation von Dualzahlen (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Berücksichtigung des Übertrags

- Um den Übertrag zu berücksichtigen, schreiben wir $a + b$ und $c + d$ in der Form $a + b = \alpha 2^{n/2} + \bar{a}$ und $c + d = \gamma 2^{n/2} + \bar{c}$ mit den führenden Bits α, γ und den $\frac{n}{2}$-stelligen Resten $\bar{a}, \bar{c}$. Dann ist

$$(a+b)(c+d) = \alpha\gamma 2^n + (\alpha\bar{a} + \gamma\bar{c})2^{n/2} + \bar{a}\bar{c}.$$

Hierin tritt nur **ein** Produkt von $\frac{n}{2}$-stelligen Zahlen auf (nämlich $\bar{a}\bar{c}$. Der Rest sind Shifts bzw. lineare Operationen auf $\frac{n}{2}$-stelligen Zahlen (z.B. $\alpha\bar{a}$).

Multiplikation von Dualzahlen (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Berücksichtigung des Übertrags (2)

- Daher erhält man insgesamt die Rekursionsgleichung $T(n) = 3T(n/2) + c_2 n$, wobei $c_2 n$ folgenden Aufwand enthält:

Additionen $a + b$, $c + d$:	$2 \cdot \frac{n}{2}$
Produkt $\alpha\gamma$:	1
Shift $\alpha\gamma$ auf $\alpha\gamma 2^n$:	n
Produkte $\alpha\bar{a}$, $\gamma\bar{c}$:	$2 \cdot \frac{n}{2}$
Addition $\alpha\bar{a} + \gamma\bar{c}$:	$\frac{n}{2} + 1$
Shift $\alpha\bar{a} + \gamma\bar{c}$ auf $(\alpha\bar{a} + \gamma\bar{c})2^{n/2}$:	$\frac{n}{2}$
Shift v auf $v2^n$:	n
Addition $u - v - w$:	$2(\frac{n}{2} + 1)$
Shift $u - v - w$ auf $(u - v - w)2^{n/2}$:	$\frac{n}{2}$
Addition zu z:	$2n$

- Dieser Aufwand addiert sich zu $8,5n + 2 < 9n$ für $n \geq 2$.
- Also kann c_2 als 9 angenommen werden.

Multiplikation von Dualzahlen (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Berücksichtigung des Übertrags (3)

- Als Lösung erhält man nach dem Aufteilungsbeschleunigungssatz

$$T(n) \in \Theta(n^{\log_2 3}) \in \Theta(n^{1,59}).$$

Bemerkung

- Der "Trick" bestand also darin, auf Kosten zusätzlicher Additionen und Shifts, eine "teure" Multiplikation von $\frac{n}{2}$-stelligen Zahlen einzusparen.
- Die rekursive Anwendung dieses Tricks ergibt dann die Beschleunigung von $\Theta(n^2)$ auf $\Theta(n^{1,59})$.
- Für die normale Computerarithmetik ($n = 32$) zahlt sich dieser Trick nicht aus, jedoch bedeutet er für Computerarithmetiken mit beliebigstelligen Dualzahlen, die meist softwaremäßig realisiert werden, eine wichtige Beschleunigung.

Multiplikation von Dezimalzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Verfahren lässt sich natürlich auch im Dezimalsystem anwenden

$$x = 4217 \quad y = 5236$$

Dann ist

$$\begin{array}{cccccccccc} a & = & 42, & b & = & 17 & \text{und} & a+b & = & 59; \\ c & = & 52, & d & = & 36 & \text{und} & c+d & = & 88. \end{array}$$

Also erhalten wir

$$\begin{aligned} u &= (a+b)(c+d) = 55 \cdot 88 = 5192 \\ v &= ac = 42 \cdot 52 = 2184 \\ w &= bd = 17 \cdot 36 = 612 \\ xy &= v \cdot 10^4 + (u - v - w) \cdot 10^2 + w \\ &= 2184 \cdot 10^4 + 2396 \cdot 10^2 + 612 \\ &= 21.840.000 + 239.600 + 612 \\ &= 22.080.212 \end{aligned}$$

Multiplikation von Matrizen

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

- Auf ähnliche Weise wie die Multiplikation von Zahlen lässt sich auch die Multiplikation (großer) $n \times n$ Matrizen beschleunigen.
- Hier erhält man die Rekursionsgleichung

$$T(2n) = 7T(n) + 14n$$

mit der Lösung $\Theta(n^{\log_2 7}) \in \Theta(n^{2,81})$, also eine Beschleunigung gegenüber der normalen Methode mit dem Aufwand $\Theta(n^3)$.

- Hier lassen sich noch weitere Beschleunigungen erzielen.

Der momentane "Rekord" steht bei $O(n^{2,39})$.

TECHNISCHE UNIVERSITÄT DARMSTADT

Quicksort

Bemerkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

- Quicksort basiert (im Gegensatz zu Mergesort) auf **variabler** Aufteilung des Eingabearrays.
- Es wurde 1962 von Hoare entwickelt.
- Es benötigt zwar im Worst Case $\Omega(n^2)$ Vergleiche, im Mittel jedoch nur $O(n \log n)$ Vergleiche.

  Beides werden wir in der Vorlesung nachweisen.

Quicksort: Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithmus

- Wir geben zunächst eine Grobbeschreibung von Quicksort an.
  1. Gegeben ist $v[]$ mit vSize Komponenten.
  2. Wähle eine beliebige Komponente $v[\text{pivot}]$.
  3. Zerlege v in zwei Teilbereiche $v[0] \ldots v[k-1]$ und $v[k+1] \ldots v[\text{vSize}-1]$ mit
     a) $v[i].\text{key} < v[\text{pivot}].\text{key}$ für $i = 0, \ldots, k-1$
     b) $v[i].\text{key} = v[\text{pivot}].\text{key}$
     c) $v[j].\text{key} > v[\text{pivot}].\text{key}$ für $i = k+1, \ldots, \text{vSize}-1$
  4. Sofern ein Teilbereich aus mehr als einer Komponente besteht, so wende Quicksort rekursiv auf ihn an.
- Die Korrektheit des Algorithmus folgt leicht durch vollständige Induktion.

  Die Aufteilung erzeugt Arrays kleinerer Länge, die nach Induktionsvoraussetzung durch die Aufrufe von Quicksort in Schritt 4 korrekt sortiert werden.

  Die Eigenschaften 3*a*)-3*c*) ergeben dann die Korrektheit für das ganze Array.

Quicksort: Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel

- In unserem Standardbeispiel ergibt sich, falls man stets die mittlere Komponente wählt (gekennzeichnet durch *) die in der folgenden Abbildung dargestellte Folge von Zuständen (jeweils nach der Aufteilung).
- Die umrahmten Bereiche geben die aufzuteilenden Bereiche an.

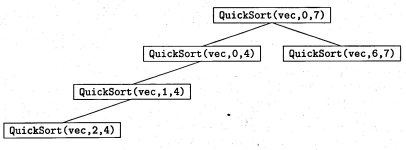
Input	63	24	12	53*	72	18	44	35
1. Aufteilung	18	24	12*	35	44	53	72*	63
2. Aufteilung	12	24	18*	35	44	53	63	72
3. Aufteilung	12	18	24	35*	44	53	63	72
Output	12	18	24	35	44	53	63	72

Quicksort: Rekursionsbaum

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursionstiefe

- Da die Aufteilung (im Unterschied zu Mergesort) variabel ist, hat der Rekursionsbaum bei Quicksort im Allgemeinen Teilbäume unterschiedlicher Höhe.
- Im Standardbeispiel ergibt sich der in der folgenden Abbildung dargestellte Baum.



- Man sieht, dass der Baum am tiefsten wird, wenn als Vergleichselement jeweils das kleinste oder größte Element des aufzuteilenden Bereichs gewählt wird.

  In diesem Fall entartet der Rekursionsbaum zu einer Liste (keine Verzweigungen).

→ Die Rekursionstiefe kann also bis zu $n-1$ betragen.

Quicksort: Rekursionsbaum (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Anzahl von Rekursionsaufrufen

- Wir zeigen durch Induktion, dass auch die Anzahl $R(n)$ der rekursiven Aufrufe bei einem Arraybereich der Länge n höchstens $n-1$ beträgt.
- Für $n = 1$ erfolgt kein Aufruf, also gilt $R(1) = 0$ (Induktionsanfang).
- Für $n > 1$ bewirkt der erste Aufruf eine Aufteilung in Bereiche mit n_1 und n_2 Komponenten, wobei $n_1 + n_2 = n - 1$ gilt, da die Vergleichskomponente wegfällt.

  Also sind $n_1, n_2 < n$ und man erhält

$$\begin{aligned} R(n) &= 1 + R(n_1) + R(n_2) \\ &\leq 1 + (n_1 - 1) + (n_2 - 1) \quad \text{Induktionsvoraussetzung} \\ &= n_1 + n_2 - 1 \\ &\leq n - 1, \end{aligned}$$

  also $R(n) \leq n - 1$.

Quicksort: Worst-case Aufwand

TECHNISCHE UNIVERSITÄT DARMSTADT

Worst-case Aufwand

- Vergleiche von Schlüsseln treten bei Quicksort nur bei den Aufteilungen auf.

  Dabei muss jeder andere Schlüssel mit dem Vergleichsschlüssel verglichen werden, also erfolgen bei einem Bereich von n Komponenten $n-1$ Vergleiche.
- Wird nun jeweils der größte bzw. kleinste Schlüsselwert als Vergleichsschlüssel gewählt, so verkleinert sich der Bereich jeweils nur um ein Element und man erhält

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \quad \text{Vergleiche.}$$

→ Also gilt für die Worst-Case Anzahl $C(n)$ von Vergleichen

$$C(n) \in \Omega(n^2).$$

- Entsprechend ergibt sich für die Worst-Case Anzahl $A(n)$ von Zuweisungen

$$A(n) \in \Omega(n^2).$$

Quicksort: Mittlerer Aufwand

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

- Qicksort ist also im Worst-Case schlecht.
- Erfahrungsgemäß ist Quicksort aber sehr schnell im Vergleich zu anderen $\Omega(n^2)$ Sortierverfahren wie Bubblesort u. a.

  Dies liegt daran, dass der Worst-Case nur bei wenigen Eingabefolgen auftritt.
- Man wird daher Quicksort gerechter, wenn man nicht den Worst-Case betrachtet, sondern den Aufwand über all möglichen Eingabefolgen mittelt, also den **mittleren Aufwand** $\bar{C}(n)$ bei Gleichverteilung aller $n!$ Reihenfolgen der Schlüssel $1, 2, \ldots, n$ betrachtet.

  Gleichverteilung bedeutet hier, dass jede Reihenfolge (Permutation) der Werte $1, \ldots, n$ mit dem gleichen Gewicht (nämlich $1/n!$) in das Mittel eingeht.

Quicksort: Mittlerer Aufwand (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnung

- Sei Π die Menge aller Permutationen von $1, \ldots, n$. Für $\pi \in \Pi$ sei $C(\pi)$ die Anzahl von Vergleichen, die Quicksort benötigt, um π zu sortieren. Dann ist

$$\bar{C}(n) = \frac{1}{n!} \sum_{\pi \in \Pi} C(\pi).$$

- Wir werden jetzt $\bar{C}(n)$ nach oben abschätzen.
- Dafür teilen wir die Menge Π aller Permutationen in die Mengen Π_1, ..., Π_n, wobei

$$\Pi_k = \{\pi \in \Pi : \text{ das Vergleichselement hat den Wert } k\}.$$

Für $n = 3$ ergibt sich $\Pi_1 = \{213, 312\}$, $\Pi_2 = \{123, 321\}$ und $\Pi_3 = \{132, 231\}$.

- In Π_k ist das Vergleichselement fest vorgeschrieben, die anderen Komponenten können jedoch in jeder Reihenfolge auftreten. Also ist

$$|\Pi_k| = (n-1)! \quad \text{für} \quad k = 1, \ldots, n.$$

Quicksort: Mittlerer Aufwand (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnung (2)

- Für alle $\pi \in \Pi_k$ ergibt die erste Aufteilung in Quicksort die Teilarrays bestehend aus einer Permutation π_1 von $1, 2, \ldots, k-1$ und einer Permuation π_2 von $k+1, \ldots, n$ (da ja das Vergleichselement gerade k ist).
- $Z(\pi)$ sei die Anzahl der Vergleiche mit der π in die Teile π_1 und π_2 zerlegt wird.

  Dann ist für alle $\pi \in \Pi_k$

$$C(\pi) = Z(\pi) + C(\pi_1) + C(\pi_2).$$

  Dabei ist $Z(\pi) \leq n$.
- Summiert man über alle $\pi \in \Pi_k$, so ergibt sich wegen $|\Pi_k| = (n-1)!$

$$\sum_{\pi \in \Pi_k} C(\pi) = \sum_{\pi \in \Pi_k} Z(\pi) + \sum_{\pi \in \Pi_k} C(\pi_1) + \sum_{\pi \in \Pi_k} C(\pi_2) =: S_1 + S_2 + S_3.$$

Quicksort: Mittlerer Aufwand (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnung (3)

- Hierin ist

$$S_1 \leq \sum_{\pi \in \Pi_k} n = (n-1)! \cdot n = n!.$$

- Wenn π alle Permutationen aus Π_k durchläuft, entstehen bei π_1 alle Permuationen von $1, \ldots, k-1$, und zwar jede $(n-1)!/(k-1)!$ mal, da Π_k ja insgesamt $(n-1)!$ Permutationen enthält.

  Also ist

$$\begin{aligned} S_2 &= \frac{(n-1)!}{(k-1)!} \sum_{\pi_1 \text{ Permutation von } 1,\ldots,k-1} C(\pi_1) \\ &= (n-1)! \bar{C}(k-1). \end{aligned}$$

- Entsprechend folgt

$$S_3 = (n-1)! \bar{C}(n-k).$$

Quicksort: Mittlerer Aufwand (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnung (4)

Durch Zusammensetzen aller Gleichungen bzw. Ungleichungen ergibt sich

$$\begin{aligned} \bar{C}(n) &= \frac{1}{n!} \sum_{\pi \in \Pi} C(\pi) = \frac{1}{n!} \sum_{k=1}^{n} \sum_{\pi \in \Pi_k} C(\pi) \\ &\leq \frac{1}{n!} \sum_{k=1}^{n} [n! + (n-1)! \bar{C}(k-1) + (n-1)! \bar{C}(n-k)] \\ &= \frac{n!}{n!} \sum_{k=1}^{n} 1 + \frac{(n-1)!}{n!} \sum_{k=1}^{n} \bar{C}(k-1) + \frac{(n-1)!}{n!} \sum_{k=1}^{n} \bar{C}(n-k) \\ &= n + \frac{1}{n} \sum_{k=1}^{n} \bar{C}(k-1) + \frac{1}{n} \sum_{k=1}^{n} \bar{C}(k-1) \\ &= n + \frac{2}{n} \sum_{k=1}^{n} \bar{C}(k-1). \end{aligned}$$

Quicksort: Mittlerer Aufwand (6)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnung (5)

Wir haben damit eine Rekursionsgleichung für $\bar{C}(n)$ gefunden. Beachtet man noch die Anfangswerte

$$\bar{C}(0) = \bar{C}(1) = 0, \bar{C}(2) = 1,$$

so ist

$$\bar{C}(n) \leq n + \frac{2}{n} \sum_{k=2}^{n-1} \bar{C}(k) \quad \text{für} \quad n \geq 2.$$

Lemma

Für die Lösung $r(n)$ der Rekursionsgleichung

$$r(n) = n + \frac{2}{n} \sum_{k=2}^{n-1} r(k) \quad \text{für} \quad n \geq 2.$$

mit den Anfangswerten $r(0) = r(1) = 0$, $r(2) = 1$ gilt für alle $n \geq 2$

$$r(n) \leq c \cdot n \cdot \ln n \quad \text{mit} \quad c = 2.$$

Quicksort: Mittlerer Aufwand (7)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis des Lemmas

Der Beweis erfolgt durch vollständige Induktion nach n.

- Induktionsanfang: Für $n = 2$ ist $r(2) = 1$. Andererseits ist $c \cdot 2 \ln 2 \approx 1,39$. Also gilt der Induktionsanfang.
- Induktionsvoraussetzung: Die Behauptung gelte für $2, 3, \ldots, n-1$.
- Schluss auf n:

$$\begin{aligned} r(n) &= n + \frac{2}{n} \sum_{k=2}^{n-1} r(k) \\ &\leq n + \frac{2}{n} \sum_{k=2}^{n-1} c \cdot k \ln k \quad \text{nach Induktionsvoraussetzung} \end{aligned}$$

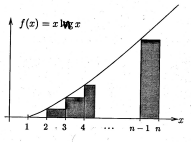
- Um diesen Ausdruck weiter nach oben abzuschätzen, betrachten wir die Funktion $f(x) = x \ln x$.

Quicksort: Mittlerer Aufwand (8)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis des Lemmas (2)

$\sum_{k=2}^{n-1} k \ln k$ ist der Flächeninhalt der schraffierten Fläche unter der Kurve $f(x)$.



- Also gilt

$$\begin{aligned} \sum_{k=2}^{n-1} k \ln k &\leq \int_2^n x \ln x \, dx = \frac{x^2}{2} \ln x \Big|_2^n - \int_2^n \frac{x}{2} \quad \text{(partielle Integration)} \\ &= \frac{n^2}{2} \ln n - 2 \ln 2 - \left(\frac{n^2}{4} - 1 \right) \\ &\leq \frac{n^2}{2} \ln n - \frac{n^2}{4} \end{aligned}$$

Quicksort: Mittlerer Aufwand (9)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis des Lemmas (3)

Hieraus folgt

$$\begin{aligned} r(n) \leq n + \frac{2}{n} \sum_{k=2}^{n-1} c \cdot k \ln k &= n + \frac{2c}{n} \sum_{k=2}^{n-1} k \ln k \\ leq \quad n + \frac{2c}{n} \left(\frac{n^2}{2} \ln n - \frac{n^2}{4} \right) \\ &= n + c \cdot n \cdot \ln n - \frac{c}{2} n \\ &= c \cdot n \cdot \ln n \quad \text{wegen} \quad c = 2. \end{aligned}$$

□

Quicksort: Mittlerer Aufwand (10)

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Für die mittlere Anzahl $\bar{C}(n)$ der Vergleiche zum Sortieren eines n-elementigen Arrays mit Quicksort gilt $\bar{C}(n) \in O(n \log n)$.

Beweis des Satzes

Aus $\bar{C}(n) \leq r(n)$ und dem Lemma folgt

$$\bar{C}(n) \leq 2 \cdot n \ln n = 2 \cdot n \cdot \frac{\log n}{\log e} = \frac{2}{\log e} \cdot n \cdot \ln n$$

für all $n \geq 2$. Also ist $\bar{C}(n) \in O(n \log n)$ mit der O-Konstanten $2/\log e \approx 2,89$. □

Bemerkungen

- Entsprechend kann man für die mittlere Anzahl $\bar{A}(n)$ von Zuweisungen beweisen, dass $\bar{A}(n) \in O(n \log n)$.
- Quicksort arbeitet also im Mittel beweisbar sehr schnell.

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapsort

Heap

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

Heapsort basiert im Gegensatz zu Mergesort und Quicksort nicht auf dem Prinzip der Aufteilung, sondern nutzt eine spezielle Datenstruktur **(Heap)**, mit der wiederholt auf das größte Element eines Arrays zugegriffen wird.

Heap

Ein **Heap** (priority queue) ist eine abstrakte Datenstruktur mit folgenden Kennzeichen:
Wertebereich: Eine Menge von Werten des homogenen Komponententyps.
Operationen:

a) Einfügen einer Komponente
b) Zugriff auf die Komponente mit maximalem Wert
c) Entfernen einer Komponente
d) Änderung des Werts einer Komponente

Grobstruktur von Heapsort

TECHNISCHE UNIVERSITÄT DARMSTADT

Grobstruktur

Gegeben ein Array a mit n Komponenten.

1 Initialisiere den Heap mit den Komponenten von a.
2 FOR $i = n-1$ DOWNTO 0 DO
  2.1 Greife auf das maximale Element des Heaps zu.
  2.2 Weise diesen Wert der Arraykomponente $a[i]$ zu.
  2.3 Entferne das größte Element aus dem Heap und aktualisiere ihn.

Termination und Korrektheit

Klar!

Ziel

TECHNISCHE UNIVERSITÄT DARMSTADT

Ziel

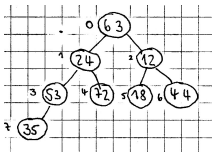
- Effiziente Implementation des Heaps, so dass die Operationen a)-d) schnell ausgeführt werden können.
- Konkret:
  - Initialisierung des Heaps in $O(n)$
  - Zugriff auf das größte Element in $O(1)$
  - Entfernen des größten Elements in $O(\log n)$

Verwirklichung des Ziels

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee

- Man stelle sich das Array a als binären Baum vor.
- Z.B. für $a = [63, 24, 12, 53, 72, 18, 44, 35]$ ergibt sich



- Mit dieser Interpretation gilt sofort:
  - $a[0]$ = Wurzel des Baums
  - $a[2i+1]$ = linker Sohn von $a[i]$
  - $a[2i+2]$ = rechter Sohn von $a[i]$

Heapeigenschaft

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition

Wir sagen, dass das Array a mit n Komponenten die **Heapeigenschaft** hat, falls gilt:

$$\left.\begin{array}{rcl} a[i] & \geq & a[2i+1] \\ a[i] & \geq & a[2i+2] \end{array}\right\} \forall i \quad \text{mit} \quad \begin{array}{rcl} 2i+1 & \leq & n-1, \\ 2i+2 & \leq & n-1. \end{array}$$

Bemerkungen

- Hat das Array a die Heapeigenschaft, so ist $a[0]$ das maximale Element des Arrays und entlang jeden Weges von einem Blatt zu der Wurzel sind die Schlüsselwerte aufsteigend sortiert.
- Demnach kann auf das größte Element von a in $O(1)$ zugegriffen werden.

Herstellen der Heapeigenschaft

TECHNISCHE UNIVERSITÄT DARMSTADT

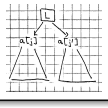
Heapify(a,L,U)

Input: a hat die Heapeigenschaft im Bereich $L+1, \ldots, U$.
Output: a hat die Heapeigenschaft im Bereich $L, \ldots, U$.

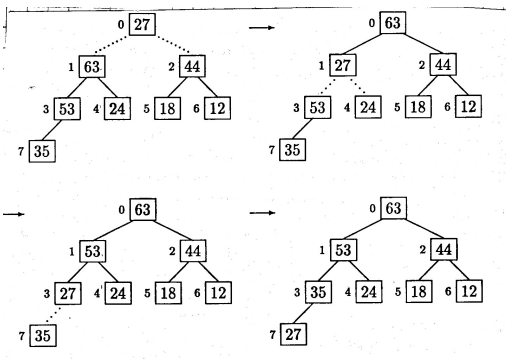
1. Bestimme den Sohn j von L mit maximalem Wert.
2. Falls $a[j] > a[L]$, tausche $a[L]$ mit $a[j]$.
3. Rufe Heapify(a,j,U) auf.

Korrektheit

- Nach Schritt 2 gilt: $a[L] \geq a[j]$ und $a[L] \geq a[j']$.
- Ferner gilt die Heapeigenschaft im Teilbaum mit Wurzel j'.
- Sie könnte jedoch im Teilbaum mit Wurzel j verletzt sein, was aber durch den rekursiven Aufruf in Schritt 3 repariert wird.



Beispiel



Erstellen eines Heaps

Bemerkungen

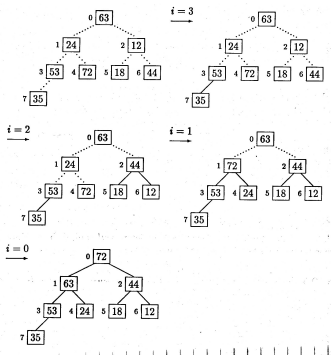
- Im folgenden bezeichne die Überprüfung und ggf. die Herstellung der Heapeigenschaft für einen Knoten mit seinen Söhnen eine **Prüfaktion**.
- Der Aufruf von Heapify(a,L,U) benötigt $\lceil \log(U-L) \rceil$ Prüfaktionen.
- Die Routine Heapify lässt sich zum Herstellen der Heapeigenschaft eines Arrays a nutzen.

CreateHeap(a,n)

FOR $i = \lfloor \frac{n-1}{2} \rfloor$ DOWNTO 0 DO

Rufe auf Heapify(a,i,n-1).

Beispiel



Zählen der Prüfaktionen

Lemma

CreateHeap(a,n) erfordert $O(n)$ Prüfaktionen.

Beweis

Sei $n \in \mathbb{N}$, so dass $2^k \leq n \leq 2^{k+1} - 1$. Dann gilt:

Schicht k	:	keine Prüfaktionen
Schicht $k-1$	:	eine Prüfaktion
	⋮	
Schicht $k-i$	:	i Prüfaktionen
	⋮	
Schicht 0	:	k Prüfaktionen

$$\Rightarrow \#\ \text{Prüfaktionen} \leq 2^{k-1} \cdot 1 + \ldots + 2^{k-i} \cdot i + \ldots + 2^0 \cdot k = \sum_{i=1}^{k} 2^{k-i} \cdot i = \sum_{i=1}^{k} \frac{2^k}{2^i} \cdot i = 2^k \sum_{i=1}^{k} \frac{i}{2^i}$$

$$\leq 2n, \quad \text{da} \quad 2^k \leq n \quad \text{und} \quad \sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

HeapSort

HeapSort

Input: a,n
Output: aufsteigend sortiertes Array a

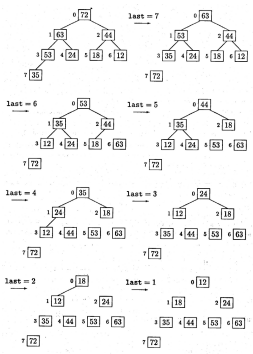
1 CreateHeap(a,n)
2 FOR $u = n-1$ DOWNTO 0
  2.1 temp= $a[0]$, $a[0] = a[u]$, $a[u]$ =temp
  2.2 Heapify(a,0,u-1)

Satz

Heapsort arbeitet korrekt und benötigt $O(n \log n)$ Prüfaktionen.

$\Rightarrow O(n \log n)$ Vergleiche und Zuweisungen

Beispiel



Untere Komplexitätsschranken für das Sortieren

Bemerkungen und untere Schranke

Bemerkungen

- Die besten bisher kennengelernten Sortierverfahren für Arrays haben einen Aufwand von $O(n \log n)$ im Worst-Case (Mergesort, Heapsort), bzw. im Average-Case (Quicksort).
- Es stellt sich nun die Frage, ob es noch bessere Sortierverfahren geben kann.
- Dies kann tatsächlich der Fall sein, wenn man zusätzliche Informationen über die Schlüsselmenge hat, wie das Verfahren **Bucketsort** zeigt.
- Basieren die Algorithmen jedoch nur auf paarweisen Vergleichen von Schlüsseln, so gilt:

Satz

Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen von Schlüsseln basiert, braucht zum Sortieren eines n-elementigen Arrays sowohl im Worst-Case als auch im Mittel (bei Gleichverteilung) $\Omega(n \log n)$ Vergleiche.

Weitere Bemerkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkungen

- Dieser Satz zeigt, dass Mergesort, Heapsort und Quicksort bzgl. der Größenordnung **optimal** sind, und dass Laufzeitunterschiede höchstens der O-Konstanten zuzuschreiben sind.
- Man beachte den Unterschied zu den bisher gemachten $O(...)$-Abschätzungen für ein Problem.
  - Diese haben wir dadurch erhalten, dass ein **konkreter** Algorithmus, der das Problem löst, analysiert wurde.
  - Die im Satz formulierte $\Omega(...)$-Abschätzung bezieht sich jedoch auf **alle möglichen** Sortierverfahren (bekannte und unbekannte).
  - Sie macht also eine Aussage über eine **Klasse von Algorithmen** statt über einen konkreten Algorithmus und ist damit von ganz anderer Natur.

Das Entscheidungsbaum-Modell

TECHNISCHE UNIVERSITÄT DARMSTADT

Entscheidungsbaum

Zum Beweis des Satzes werden wir die Tatsache nutzen, dass jeder deterministische Sortieralgorithmus, der nur auf paarweisen Vergleichen von Schlüsseln basiert, durch einen **Entscheidungsbaum** wie folgt beschrieben werden kann.

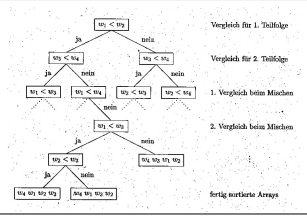
- Innere Knoten des Entscheidungsbaums sind Vergleiche im Algorithmus.
- Blätter des Baumes sind die sortierten Arrays, also $n!$ bei n zu sortierenden Elementen.
- Ein Weg von der Wurzel bis zu einem Blatt entspricht im Algorithmus der Folge der angestellten Vergleiche.
- Dabei wird vereinbart, dass beim Weitergehen nach **links** bzw. **rechts** der letzte Vergleich richtig (**true**) bzw. falsch (**false**) ist.

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel

- Als Beispiel betrachten wir einen Ausschnitt des Entscheidungsbaumes für das Sortieren von w_1, w_2, w_3, w_4 mit Mergesort.
- Es erfolgt also zunächst die Aufteilung in die Teilfolgen w_1, w_2 und w_3, w_4.
- Diese werden dann sortiert und gemischt.
- Es entsteht der in der folgenden Abbildung gezeigte Baum.



Jeder Algorithmus erzeugt Entscheidungsbaum

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen basiert, erzeugt einen solchen Entscheidungsbaum T.

Beweis

- Da der Algorithmus deterministisch ist, hat er für jede Eingabefolge denselben **ersten** Vergleich zwischen Arraykomponenten.
- Dieser bildet die **Wurzel** des Entscheidungsbaumes.
- In Abhängigkeit vom Ausgang des Vergleiches ("$<$" oder "$>$") ist der nächste Vergleich wiederum eindeutig bestimmt.
- Die Fortsetzung dieser Argumentation liefert für jede Eingabefolge eine endliche Folge von Vergleichen, die einem Weg von der Wurzel bis zu einem Blatt (sortierte Ausgabe) entspricht. □

Ein paar Überlegungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Gedanken

- Wir überlegen nun, wie wir den Worst-Case- bzw. Average-Case-Aufwand $C(n)$ bzw. $\bar{C}(n)$ des Algorithmus im Baum T ablesen können.
- Sei dazu $h(v)$ die Höhe des Knoten v im Baum T, $h(T)$ die **Höhe** von T, und $H(T) := \sum_{h \text{ Blatt}} h(v)$ die sogenannte **Blätterhöhensumme** von T.

Lemma

Sei T Entscheidungsbaum für Algorithmus A und $C(n)$ bzw. $\bar{C}(n)$ die Worst-Case- und Average-Case-Anzahl (bei Gleichverteilung) von Vergleichen bei n zu sortierenden Komponenten. Dann gilt:

a) $C(n) = \max\limits_{v \text{ Blatt}} h(v) = h(T)$,

b) $\bar{C}(n) = \frac{1}{n!} \sum\limits_{v \text{ Blatt von } T} h(v) = \frac{1}{n!} H(T)$.

Ein paar Überlegungen (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Lemma

Sei T Entscheidungsbaum für Algorithmus A und $C(n)$ bzw. $\bar{C}(n)$ die Worst-Case- und Average-Case-Anzahl (bei Gleichverteilung) von Vergleichen bei n zu sortierenden Komponenten. Dann gilt:

a) $C(n) = \max\limits_{v \text{ Blatt}} h(v) = h(T)$,

b) $\bar{C}(n) = \frac{1}{n!} \sum\limits_{v \text{ Blatt von } T} h(v) = \frac{1}{n!} H(T)$.

Beweis

- a) Anzahl der Vergleiche, um zur sortierten Ausgabe v zu kommen, ist gerade $h(v)$.
- b) Gleichverteilung der $n!$ verschiedenen Eingabereihenfolgen und damit auch der Ausgabereihenfolgen

Analyse des Entscheidungsbaumes

TECHNISCHE UNIVERSITÄT DARMSTADT

Bemerkung

Die Abschätzung von $C(n)$ bzw. $\bar{C}(n)$ nach unten reduziert sich also auf die Abschätzung der Höhe bzw. der Blätterhöhensumme eines binären Baumes mit $n!$ Blättern nach unten.

Lemma

Sei T ein binärer Baum mit b Blättern. Dann gilt:

a) $h(T) \geq \log b$,

b) $H(T) \geq b \cdot \log b$.

Beweis

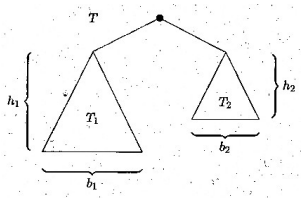
- Der Beweis wird in beiden Fällen durch vollständige Induktion nach der Höhe $h(T)$ von T geführt.
- Ist $h(T) = 0$, so besteht T nur aus der Wurzel, die zugleich ein Blatt ist.
- Also ist $b = 1$ und $\log b = 0 = h(T)$.
- Entsprechend ist $H(T) = \sum_{h \text{ Blatt}} h(v) = 0$ und $b \cdot \log b = 0$.

Analyse des Entscheidungsbaumes (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis (2)

- Es gelten nun a), b) für $h(T) = 0, 1, \ldots, h-1$ (Induktionsvoraussetzung).
- Zum Schluss auf h betrachte man die beiden Teilbäume T_1 und T_2 von T, wobei einer leer sein kann, vgl. folgende Abbildung.



- Jeder der Teilbäume hat eine geringere Höhe als h, also trifft auf T_1 und T_2 die Induktionsvoraussetzung zu.

Analyse des Entscheidungsbaumes (3)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis (3)

- Sei b_i die Anzahl der Blätter und h_i die Höhe von T_i, $(i = 1, 2)$, und sei o.B.d.A. $b_1 \geq b_2$.
- Dann gilt:

$$\begin{aligned} h(T) = 1 + \max\{h_1, h_2\} &\geq 1 + h_1 \\ &\geq 1 + \log b_1 \quad \text{Induktionsvoraussetzung} \\ &= \log 2 + \log b_1 \\ &= \log(2b_1) \\ &\geq \log(b_1 + b_2), \quad (\text{da } b_1 \geq b_2) \\ &= \log b. \end{aligned}$$

- Dies beweist a).

Analyse des Entscheidungsbaumes (4)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis (4)

- Im Fall b) gilt

$$H(T) = b + H(T_1) + H(T_2),$$

da in T jedes Blatt gegenüber T_1 und T_2 eine um 1 größere Höhe hat.

- Auf T_1 und T_2 ist die Induktionsvoraussetzung anwendbar und es folgt

$$\begin{aligned} H(T) &\geq b + b_1 \log b_1 + b_2 \log b_2 \quad \text{Induktionsvoraussetzung} \\ &= b + b_1 \log b_1 + (b - b_1) \log(b - b_1). \end{aligned}$$

- Da wir nicht genau wissen, wie groß b_1 ist, fassen wir die rechte Seite als von Funktion von $x = b_1$ auf und suchen ihr Minimum.
- Also ist

$$H(T) \geq b + \min_{x \in [1,b]} [x \log x + (b - x) \log(b - x)].$$

- Eine Kurvendiskussion zeigt, dass $f(x) := x \log x + (b - x) \log(b - x)$ das Minimum auf $[1, b]$ bei $x = b/2$ annimmt.

Analyse des Entscheidungsbaumes (5)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis (5)

- Damit ist

$$\begin{aligned} H(T) &\geq b + \frac{b}{2} \log \frac{b}{2} + \frac{b}{2} \log \frac{b}{2} \\ &= b + b \log \frac{b}{2} \\ &= b + b(\log b - \log 2) \\ &= b + b(\log b - 1) \\ &= b \log b. \end{aligned}$$

□

Bemerkung

Für die endgültige Abschätzung benötigen wir noch eine Abschätzung von $n!$ nach unten.

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 \geq n \cdot (n-1) \cdot \ldots \cdot \lceil n/2 \rceil \geq \lceil n/2 \rceil^{\lfloor n/2 \rfloor + 1} \geq (n/2)^{n/2}.$$

Beweis des Satzes über untere Komplexitätsschranken beim Sortieren

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz

Jeder deterministische Sortieralgorithmus, der auf paarweisen Vergleichen von Schlüsseln basiert, braucht zum Sortieren eines n-elementigen Arrays sowohl im Worst-Case als auch im Mittel (bei Gleichverteilung) $\Omega(n \log n)$ Vergleiche.

Beweis

- Sei T der Entscheidungsbaum zum gegebenen Sortieralgorithmus A.
- Bei einem Inputarray der Länge n hat T $n!$ Blätter.

$$\begin{aligned} C(n) = h(T) &\geq \log n! \\ &\geq \log[(n/2)^{n/2}] \\ &= \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2} = \frac{n}{3} \log n + \frac{n}{6} \log n - \frac{n}{2} \\ &\geq \frac{n}{3} \log n \quad \text{für} \quad \frac{n}{6} \log n \geq \frac{n}{2}, \text{ also für } n \geq 8 \\ &\in \Omega(n \log n). \end{aligned}$$

Beweis des Satzes über untere Komplexitätsschranken beim Sortieren (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beweis (2)

- Entsprechend ist

$$\bar{C}(n) = \frac{1}{n!} H(T) \geq \frac{1}{n!}(n! \log n!) = \log n! \in \Omega(n \log n) \quad \text{(wie oben)}.$$

Bemerkung

- Dieses Ergebnis, bzw. genauer die Ungleichungen

$$C(n) \geq \log n! \quad \text{und} \quad \bar{C}(n) \geq \log n!$$

werden auch als **informations-theoretische Schranken** für das Sortieren bezeichnet.

- Sie lassen sich anschaulich folgendermaßen interpretieren.
  - Jeder Sortieralgorithmus muss zwischen $n!$ Möglichkeiten (den sortierten Reihenfolgen) unterscheiden und muss daher $\log n!$ Bits an Information sammeln.
  - Ein Vergleich ergibt höchstens ein Bit an Information.