

VORLESUNGSFOLIEN:

Das Stable-Matching-Problem - 01

Analyse von Algorithmen - 02

Graphen - 03

Greedy-Algorithmen - 04

Divide-and-Conquer - 05

Suchbäume - 06

Hashing - 07

Praktische Datenstrukturen - 08

Polynomialzeitreduktion & NP - 09

NP-Vollständigkeit Spezialfälle - 10

Optimierung – Branch-and-Bound - 11

Optimierung – Dynamische Programmierung - 12

Optimierung - Approximation - 13

Optimierung - Heuristische Verfahren – 14

- 1 **Stable Matching**
 - 1.1 Gale-Shapely-Algorithmus
- 2 **Algorithmenanalyse**
 - 2.1 Asymptotisches Wachstum
 - 2.2 Laufzeiten einiger gebräuchlicher Funktionen
 - 2.3 Sortieren als praktisches Beispiel
 - 2.3.1 Bubble Sort
 - 2.3.2 Selection Sort
 - 2.3.3 Insertion Sort
- 3 **Graphen**
 - 3.1 Breitensuche (BFS)
 - 3.2 Tiefensuche (DFS)
 - 3.3 Zusammenhangskomponenten im gerichteten Graphen
 - 3.4 DAGs
 - 3.5 Topologische Sortierung
 - 3.6 Dijkstra-Algorithmus
 - 3.7 Priority Queue / Heap
- 4 **Greedy-Algorithmen**
 - 4.1 Interval Scheduling
 - 4.2 MST - Minimaler Spannbaum
 - 4.2.1 Algorithmus von Prim
 - 4.2.2 Algorithmus von Kruskal
 - 4.2.3 Union-find-Datenstruktur
- 5 **Divide-and-Conquer**
 - 5.1 Merge Sort
 - 5.2 Quick Sort und Tim Sort
 - 5.3 Inversionen zählen
 - 5.4 Dichtestes Punktpaar
- 6 **Suchbäume**
 - 6.1 Durchmusterungen
 - 6.2 Binäre Suchbäume
 - 6.3 Balancierte Bäume
 - 6.3.1 Höhenbalancierte Bäume: AVL-Bäume
 - 6.3.2 Breitenbalancierte Bäume: B-Bäume und B*-Bäume
- 7 **Hashing**
 - 7.1 Hashfunktionen
 - 7.1.1 Multiplikations-Methode
 - 7.1.2 Modulo-Methode
 - 7.2 Kollisionsbehandlungen
 - 7.2.1 Verkettung der Überläufer
 - 7.2.2 Offene Hashverfahren – Sondierung
 - Lineare Sondierung
 - Quadratische Sondierung
 - Double Hashing und Uniform Hashing
 - Verbesserung nach Brent

8 Datenstrukturen in Java

8.1 Java Collections Framework

9 Komplexitätstheorie

9.1 Polynomialzeitreduktion und NP

9.1.1 Problemtypen und Reduktion von Ja/Nein-Probleme

9.1.2 3 Reduktionsstrategien

Äquivalenz:

Independent Set und Vertex Cover

MNB (Maximaler Nicht Blockierer) und MST* (Minimaler Spannbaum)

Spezialfall auf allgemeinen Fall:

Vertex Cover auf Set Cover

Reduktion durch Gadgets:

3SAT auf Independent Set

9.1.3 P, NP, NP-C, NP-Hard

9.1.4 Zertifikat, Zertifizierer

9.1.5 SAT auf 3 Color

9.2 NP-Vollständigkeit Spezialfälle

9.2.1 Independent Set

9.2.2 Vertex Cover

9.2.3 3 Color mit Intervallen

10 Optimierung

10.1 Branch-and-Bound

10.1.1 Rucksackproblem

10.1.2 Maximierungsprobleme: Allgemeiner Algorithmus

10.1.3 Minimierungsprobleme: Allgemeiner Algorithmus

10.1.4 Minimales Vertex Cover

10.2 Dynamische Programmierung

10.2.1 Gewichtetes Interval Scheduling Problem

10.2.2 Segmented Least Squares

10.2.3 Rucksackproblem

10.2.4 Kürzeste Pfade mit negativen Kantengewichten aber ohne negativen Kreisen

10.3 Approximation

10.3.1 Vertex Cover

10.3.2 Spanning-Tree-Heuristik (ST) für das symmetrische und euklidische TSP

10.3.3 Lastverteilung – „Load Balancing“

10.3.4 Center Selection

10.4 Heuristische Verfahren

10.4.1 Konstruktionsverfahren

10.4.2 Verbesserungsheuristiken – Lokale Suche

Vertex Cover, Lokale Suche 1 und 2 mit Approximationsalgorithmus

MAX-SAT, Bit Flip

TSP, 2-exchange

10.4.3 Metaheuristiken

Simulated Annealing SA – Graph Bi Partitionierung

Tabu-Suche

Evolutionäre Algorithmen

Das Stable-Matching-Problem

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2020

Letzte Änderung: 3. März 2020

Vorlesungsfolien



1 / 23

Stable-Matching-Problem

Gegeben seien n Kinder und n Gastfamilien, die an einem Austauschprogramm für Schülerinnen und Schülern teilnehmen.

Ziel: Finde eine passende Zuordnung von Kindern zu Gastfamilien.

- Jedes Kind hat eine Präferenzliste von Gastfamilien.
 - Jede Gastfamilie hat eine Präferenzliste von Kindern.

Kinder: Xaver, Yvonne, Zola.

Gastfamilien: Abel, Boole, Church.

	höchste Präferenz		niedrigste Präferenz	
	1.	2.	3.	
Xaver	Abel	Boole	Church	
Yvonne	Boole	Abel	Church	
Zola	Abel	Boole	Church	

Präferenzlisten der Kinder

Präferenzlisten der Kinder

	höchste Präferenz		niedrigste Präferenz	
	1.	2.	3.	
A	Abel	Yvonne	Xaver	Zola
B	Boole	Xaver	Yvonne	Zola
C	Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable Matching Problem

Stable Matching

keine instabilen Matches (Paare)

instable Matching

Partner-Wechsel vorteilhaft
für Familie und Kind,



Das bessere Paar

Gale-Shapley-Algorithmus

Markiere alle Kinder als „frei“

Markiere alle Familien als „frei“

while (\exists freies Kind $\wedge \exists$ freie Familie)

$s \leftarrow$ freies Kind

$f \leftarrow$ 1. Familie in Präferenzliste von s

if ($f =$ frei)

Match ($s-f$)

else if ($f \neq$ frei $\wedge s$ höher auf Liste als aktueller Partner von f)

Match ($s-f$)

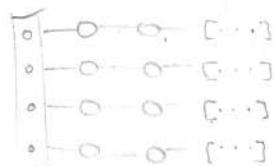
$s' =$ frei

else

f lehnt s ab

while Schleife mit $O(n^2)$ Iterationen

weil



n Kinder mit n Familien auf
Liste, Kinder könnten im
Worst Case gesamte Liste
durchgehen

siehe Seite 18/23

Implementierung mit der richtigen Datenstruktur

FPref [s, i] - Array	Bestimmt von s, die bevorzugte Familie an Stelle i
SPref [f, i] - Array	Bestimmt von f, das bevorzugte Kind an Stelle i
SFree - Queue	
Next [s] - Array	Nächste noch nicht überprüfte Stelle im Kind-Array Nächste Familie: FPref [s, Next[s]]
Current [f]	Match von f bzw „-1“ wenn frei
Ranking [f, s]	Wert von allen Matchings (wird vor Algorithmus erstellt) Eigentlich SPref [f, i] nur statt i kann man direkt nach Kind suchen

Vor Algorithmus Ranking Liste erstellen $O(n^2)$:

```
for i <= 0 bis n-1
    for j <= 0 bis n-1
        Ranking [i, FPref[i, j]] <- j
```

GS-Algorithmus:

Fallen 2, Seite 72/91

Analyse von Algorithmen
Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2020
Letzte Änderung: 4. März 2020
Vorlesungsfolien



1 / 91

Effizient berechenbare Probleme

„For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.“

Francis Sullivan

2 / 91

Asymptotische Analyse von Algorithmen (Landau-Symbole)

Obere Schranke - groß O

für $n \rightarrow \infty$ $a_n \in O(b_n)$ wenn $\exists c > 0$, sodass

$$\left| \frac{a_n}{b_n} \right| \leq c \quad \text{für } n \geq N \quad (n \geq 0)$$

Untere Schranke - Omega Ω

$$\left| \frac{b_n}{a_n} \right| \leq c \quad \text{für } n \geq N \quad (n \geq 0)$$

Scharfe Schranke - Theta Θ

$\exists c_1 > 0 \wedge \exists c_2 > 0$, sodass

$$c_1 \cdot |b_n| \leq |a_n| \leq c_2 \cdot |b_n| \quad \text{für } n \geq N$$

Also zugleich
 $a_n \in O(b_n)$ und $b_n \in O(a_n)$

Daneben folgt

$$a_n \in \Theta(b_n) \iff b_n \in \Theta(a_n)$$

Asymptotische Dominanz

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^2 \ll n^3 \ll n^k \ll 2^n \ll k^n \ll n! \ll n^n$$

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^k \ll k^n \ll n! \ll n^n$$

Sortieralgorithmen als Beispiel

Bubblesort $\Theta(n^2)$

bekannt aus EP 1

Selectionsort $\Theta(n^2)$

Suche Minimum aus unsortierten Teil



$A[0] \dots A[i-1]$ sortiert

$A[i] \dots A[n]$ unsortiert

Vertauscht jede Zahl min. $dx \rightarrow \frac{n(n-1)}{2} \in \Theta(n^2)$

Insertionsort $\Theta(n^2)$

Nimmt Element aus unsortierten Teil und inserted an richtigen Stelle im sortierten Teil

(Beweis kompliziert für $\Theta(n^2)$)

(ALTE
UNTERLAGEN)

Beispiel für Laufzeitanalyse:

Obere Schranke

$O(f(n))$ Menge aller Funktionen die von $f(n)$ von oben beschränkt werden

$T(n) \in O(f(n))$ wenn

$\exists c > 0 \quad \exists n_0 > 0$, sodass

$\forall n \geq n_0$ gilt:

$$T(n) \leq \overset{c}{\cancel{c}} \cdot f(n) \quad \text{konstante Faktoren ausgliessen}$$

$$T(n) = 32n^2 + 17 + 5$$

$T(n) \in O(n^2)$ wenn:

$$\exists c > 0, n_0 > 0 : \forall n \geq n_0 \quad T(n) \leq c \cdot f(n)$$

$$32n^2 + 17n + 5 \leq c \cdot n^2 \quad | : n^2$$

$$32 + \frac{17}{n} + \frac{5}{n^2} \leq c$$

→ Beispieldweise gültig wenn

$$n \geq 18$$

$$c \geq 34$$

$$\begin{array}{l} n_0 = 18 \\ c = 34 \end{array}$$

Untere Schranke

$$T(n) = 32n^2 + 17 + 5$$

$T(n) \in \Omega(n^2)$ wenn:

$$\underline{\quad} \quad \underline{\quad} \quad T(n) \geq c \cdot f(n)$$

$$32n^2 + 17n + 5 \geq c \cdot n^2$$

$$32n + \frac{17}{n} + \frac{5}{n^2} \geq c$$

→ Beispieldweise gültig wenn

$$n \geq 1$$

$$\underline{\quad} \quad \underline{\quad} \quad c \leq 32$$

$$\begin{array}{l} n_0 = 1 \\ c = 32 \end{array}$$

Schwarze Schnecke

$T(n)$ gleichzeitig $\in \Omega(f(n)) \wedge \in O(f(n))$

Hier trifft das wie gest. bereits gezeigt zu.

Demo:

$$c_1 n^2 \leq 32n^2 + 17n + 5 \leq c_2 \cdot n^2$$

$$c_1 \leq 32 + \frac{17}{n} + \frac{5}{n^2} \leq c_2$$

$$c_1 = 32$$

$$c_2 = 34$$

$$\text{mit } n_0 = 18 \quad \leftarrow \max(1, 18)$$

✓

Asymptotische Analyse, Beispiel

$$T(n) = 32n^2 + 17n + 5$$

Obere Schranke $T(n) \in O(n^2)$

Behauptung:

$$\exists n_0 > 0 \wedge \exists c > 0 \text{ sodass}$$

$$32n^2 + 17n + 5 \leq c \cdot n^2 \text{ für } \forall n \geq n_0$$

Schärfere Schranke

$$T(n) \in \Theta(n^2)$$

weil $T(n) \in \Omega(n^2)$ und $O(n^2)$

Trotzdem mit Rechenweg \square

Behauptung:

$$\exists c_1 > 0 \wedge \exists c_2 > 0$$

$$\exists n_0 > 0 \text{ sodass}$$

$$c_1 \cdot n^2 \leq 32n^2 + 17n + 5 \leq c_2 \cdot n^2$$

für alle $n \geq n_0$

Beweis:

$$c_1 \leq 32 + \frac{17}{n} + \frac{5}{n^2} \leq c_2$$

\downarrow

$$c_1 = 32 \quad n_0 = 18 \quad (\text{Max}(1, 18))$$

$$\circlearrowleft c_2 = 39$$

$$T(n) \in \Theta(n^2)$$

Beweis:

$$32n^2 + 17n + 5 \leq c \cdot n^2 \quad | : n^2$$

$$32 + \frac{17}{n} + \frac{5}{n^2} \leq c$$

\hookrightarrow gültig für

$$n \geq 18 \quad c \geq 34$$

Best Case:
(beste Approximation)

$$n_0 = 18$$

$$c = 34$$

Es gilt:

$$T(n) \in O(n^2) \quad \square$$

Untere Schranke $T(n) \in \Omega(n^2)$

Behauptung:

$$\exists n_0 > 0 \wedge \exists c > 0 \text{ sodass}$$

$$32n^2 + 17n + 5 \geq c \cdot n^2$$

Beweis:

$$32n^2 + 17n + 5 \geq c \cdot n^2 \quad | : n^2$$

$$32 + \frac{17}{n} + \frac{5}{n^2} \geq c$$

\hookrightarrow gültig für

$$n \geq 1 \quad c \leq 32$$

$$\rightarrow \begin{array}{l} n_0 = 1 \\ c = 32 \end{array}$$

Es gilt

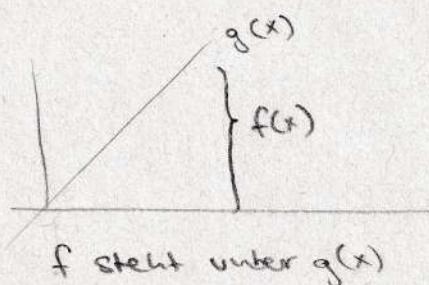
$$T(n) \in \Omega(n^2) \quad \square$$

Wichtig:

Nun könnte auch die Schranken n^3 und n nehmen: $O(n^3), \Omega(n)$ aber n^2 ist die beste Annäherung

Asymptotische Dominanz

g dominiert f $f \ll g$ $\rightarrow f(x) \in O(g(x))$
 $g(x) \notin O(f(x))$



Beispiel

$$f(n) \ll g(n)$$

$$\begin{aligned} g(n) &= n^3 \\ f(n) &= n^2 \end{aligned}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

Se trennen sich immer weiter voneinander

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n^2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0$$

Beispiel

$$\begin{aligned} g(n) &= 2n^2 \\ f(n) &= n^2 \end{aligned} \quad \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\frac{1}{2}n^2}{2n^2} = \frac{1}{2} \neq 0 \Rightarrow \text{keine Dominanz, weder } f \ll g \text{ noch } g \ll f$$

Asymptotische Schranken für einige gebräuchliche Funktionen

Polynome mit Grad d

$$f(n) = a_0 + a_1 n^1 + a_2 n^2 + \dots + a_d n^d \Rightarrow \Theta(n^d) \text{ wenn } a_d > 0$$

(Aufgrund von Additivität)

$O(n^d)$ und $\Omega(n^d)$

Logarithmen

$$a, b > 0$$

$$\Theta(\log_a n) = \Theta(\log_b n) \text{ weil}$$

$$\log_a n = \frac{\log_b n}{\log_b a} \Rightarrow \log_a n = \log_b n$$

Deshalb gibt man bei asymptotischen Angaben die Basis des Logarithms nicht an.

Auch wichtig:

$$\log n \ll n^k$$

weil \log asymptotisch immer langsamer wächst als jedes Polynom

Exponentiell

Dominiert immer Polynome \rightarrow jede Exponentialfunktion wächst asymptotisch schneller als jede Polynomfunktion.


 $n^d \ll c^n$ $\leftarrow \text{exp.}$
Poly.

$$c > 1 \quad d > 0$$

Unter Logarithmen gilt:

$$\text{wenn } 1 < c_1 < c_2$$

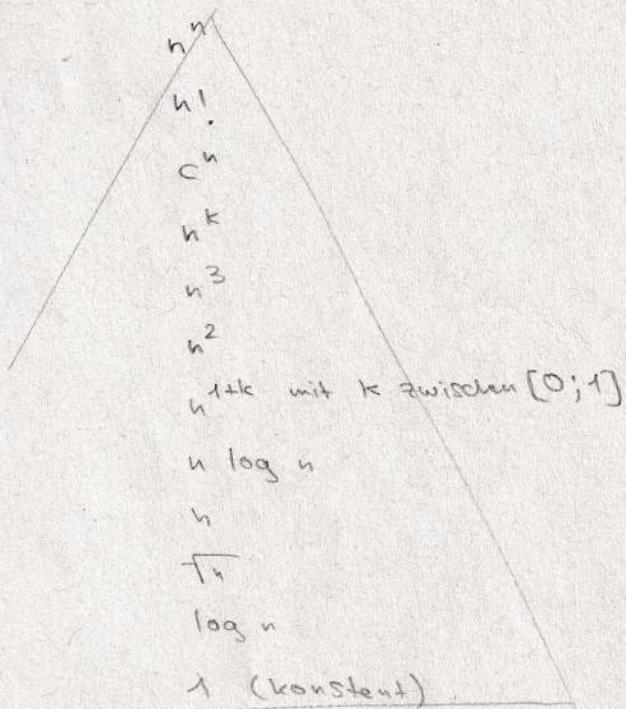
$$c_1^n \ll c_2^n$$

weil:

$$c_1^n \in O(c_2^n)$$

$$\lim_{n \rightarrow \infty} \frac{c_1^n}{c_2^n} = \lim_{n \rightarrow \infty} \left(\frac{c_1}{c_2}\right)^n = 0$$

Hierarchie der Dominanz



Asymptotische Analyse (Mathematik)

Methode, bestimmt den wesentlichen Trend des Grenzverhaltens:

Funktion $f(x)$ und $g(x)$ sind äquivalent ($f \sim g$) wenn

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1 \quad \rightarrow \text{relativer Fehler wird mit } n \rightarrow \infty \text{ dann } 0.$$

$f(n) \in \text{Äquivalenzklasse von } g(n)$ in der alle Funktionen sind die äquivalent sind, „Schranken“

Landau-Notation

$$f(x) \in O(g(x)) \Leftrightarrow \limsup_{x \rightarrow x_0} \left| \frac{f(x)}{g(x)} \right| < \infty \quad \forall x$$

$$f(x) \in o(g(x)) \Leftrightarrow \lim_{\substack{x \rightarrow x_0 \\ \text{klein } 0}} \left| \frac{f(x)}{g(x)} \right| = 0 \quad \forall x$$

Man benutzt die Landau-Notation für Zeitkomplexität \rightarrow „Asymptotische Laufzeit“

Nicht Aufwand auf einem bestimmten Redner

Die Schranken der Landau-Notation:

$$\underbrace{\Omega(g(n))}_{\substack{\text{untere} \\ \text{Schranke}}} \leq T(n) \leq \underbrace{O(f(n))}_{\substack{\text{obere} \\ \text{Schranke}}}$$

Tatsächl. Zeitaufw.

sonstige Relation zwischen Eingabemenge und Zeit bei großen Eingaben

„Skalierbarkeit“

Θ Scherfe Schranke:
wenn obere und untere Schranke identisch

Obere Schranke $O(f(n))$ \rightarrow Menge aller Funktionen die asymptotisch durch $c \cdot f(n)$ oben beschränkt werden
von oben beschränkt

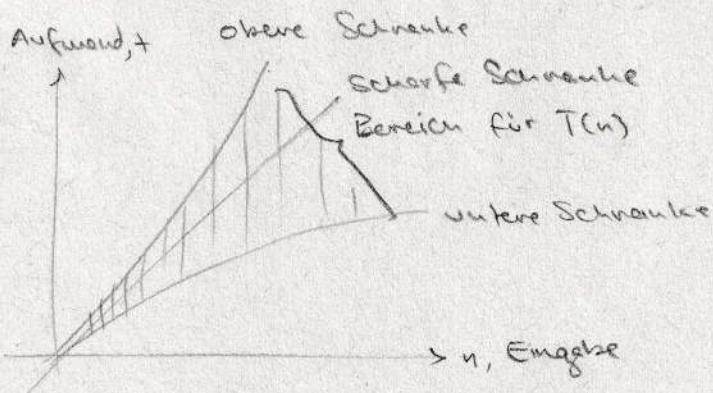
$T(n) \in O(f(n))$ „ $T(n)$ ist in $O(f(n))$ “
wenn $\exists n_0, \exists c$ sodass
 $\forall n \geq n_0 : T(n) \leq c \cdot f(n)$ konstanten ignoriert

Untere Schranke $\Omega(f(n))$

von unten beschr. \rightarrow Menge Δ Funktionen die von $f(n) \cdot c$ unten beschränkt werden asymptotisch
 $T(n) \in \Omega(f(n))$
 $\exists n_0 > 0, \exists c > 0$ sodass
 $\forall n \geq n_0 : T(n) \geq c \cdot f(n)$

Scharfe Schranke $\Theta(f(n))$

$T(n) \in \Theta(f(n))$ \rightarrow Menge Δ Funkt. die asymptotisch gleich großes Wachstum wie $c \cdot f(n)$ besitzen
wenn
 $T(n) \in O(f(n)) \wedge T(n) \in \Omega(f(n))$



man wählt die Schranken möglichst eng zu $T(n)$

Graphen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 26. März 2019
Vorlesungsfolien



1 / 95

Grundlegende Definitionen und Anwendungen

2 / 95

Graphen - Modellierung

Adjazenzmatrix

bool'sche $n \times n$ -Matrix ($\lambda = \text{Knoten sind adjazent}$)

Nicht geeignet wenn Graph nicht dicht ist (dichter Graph: zB vollständiger Graph mit $\frac{n(n-1)}{2}$ Kanten nach Handshake Theorem also $\Theta(n^2)$)

Platzbedarf $\Theta(n^2)$

Check von Adjazenz $\Theta(1)$

Aufzähler aller Kanten $\Theta(n^2)$

Adjazenzlisten

Liste an adjazenten Knoten für jeden Knoten

Platzbedarf $\Theta(n + w)$

Check von Adjazenz $\Theta(\deg v)$

Aufzähler aller Kanten $\Theta(n + w)$

Knoten $|V| = n$

Kanten $|E| = w$

Breitensuche - Ermittlung von Ebenen

BFS - Breadth first search

→ Level L_i hat Distanz i zu Startknoten

BFS(G, s)

Level [s] = 0

Level [$V \setminus \{s\}$] = -1

} Initialisierung $\Theta(n-1)$

Queue $Q \leftarrow \{s\}$

While Q ist nicht leer

$u \leftarrow Q.pop$

foreach Knoten adjazent zu u (v)

if Level [v] = -1

Level [v] = Level [u] + 1

$Q.add(v)$

Jeder Knoten der Adjazenzliste wird durchgängig

$$n + \sum_{v \in V} \deg(v) = n + 2m \in \Theta(n+m)$$

Tiefensuche - Ermittlung des längsten Pfades vom Startpunkt in DAG's oder Zusammen-

DFS - Depth first search

Hang-Komponenten

DFS(G, s):

Discovered [V] = false (für alle Knoten)

} $\Theta(n)$

DFS1(G, s)

DFS1(G, u)

Discovered [u] = true

System.out.print(u)

foreach Knoten v adjazent zu u

if ! Discovered [v]

DFS1(G, v)

worst case:
 $\Theta(n+2m)$

Beide in $\Theta(n+m)$

Zusammenhangskomponenten im gerichteten Graphen

Schwache Zusammenhangskomponenten mit DFS bestimmen

(wäre stark zusammenhängend wenn Graph ungerichtet wäre)

Wichtig: einzelne Knoten auch Komponenten

DFSNUM(a)

Discovered[v] = false

i = 0

foreach Knoten v

if ! Discovered[v]

i++

DFS1(G, v)

return i

Sterke Zusammenhangskomp. mit BFS bestimmen

1. beliebiger Knoten s

2. BFS(G, s)

3. BFS(G^{rev} , s) $\leftarrow G^{rev} = \text{Alle Kanten verdreht}$

- ist stark zusammenh.

wenn alle Knoten in Schritt 2 und 3 erreicht werden können

gerichtete aczyklische Graphen

DAG - directed acyclic graph

```
while  $\exists v \in V$ 
    if  $\exists$  Quelle q
        lösche q und alle incidenten Kanten
        System.out.print(q)
    else
        return G ist kein DAG
return G ist ein DAG
```

Quelle u : $\deg^-(u) = 0$
Senke u : $\deg^+(u) = 0$
 $O(n)$

Topologische Sortierung

Lineare Ordnung der Knoten, wenn $u \rightarrow v$ dann muss
u vor v stehen

(Interpretation:
Aufgabe u muss vor v erledigt werden)

Wichtig: DAG \Leftrightarrow topologische Sortierung

```
foreach  $v \in V$ 
    count [v] = 0
```

Insgesamt :

$O(n + n)$

```
foreach  $v \in V$ 
    foreach Knoten w adjazent zu v
        count [w] = count [w] + 1
```

```
foreach  $v \in V$ 
    if count [v] = 0
        Q.add(v)
```

```
while Q ist nicht leer
    System.out.println(Q.pop  $\rightarrow$  v)
    foreach Knoten w adjazent zu v
        count [w] = count [w] - 1
        if count [w] = 0
            Q.add(w)
```

DIJKSTRA

Startknoten

Definition:

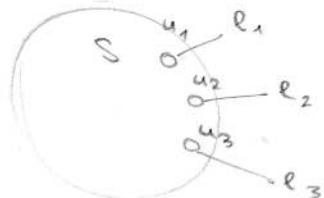
Distanz von Startknoten zu Knoten v : $d(v) = \sum$ Gewichte l von s bis v
Bereits untersuchte Knoten S

Algorithmus:

1. Initialisiere $S = \{s\}$, $d(s) = 0$

2. Wähle Knoten v (wobei $v \notin S$ sätzlich)

$$\min_{e=(u,v): u \in S} d(u) + l_e$$



3. Füge v zu S zu,

$$\text{setze } d(v) = \min_{e=(u,v): u \in S} d(u) + l_e$$

$$\min \{ u_1 + l_1, u_2 + l_2, u_3 + l_3 \}$$

4. Beiträgt Pfad

Dijkstra (G, s)

Discovered [v] = false

$$d[s] = 0$$

$$d[V \setminus \{s\}] = \infty$$

$$Q \leftarrow V$$

while Q ist nicht leer

wähle $u \in Q$ mit minimalstem Wert $d[u]$ (Priority Queue / Heap)

lösche u aus Q

Discovered [u] = true

foreach Knoten adjazent zu u (v)

if !Discovered [v]

if $d[v] > d[u] + l_e$

lösche v aus Q

$$d[v] = d[u] + l_e$$

$Q.add(v)$

Anhuse von Dijkstra

Implementiert mit linked list: $O(n^2)$

$O(n)$ Array initialisierung

$O(n^2)$ while Schleife sucht $n \times$ kleinsten Wert

$O(m)$ foreach (jede Kante max $d(x)$)

$O(n + n^2 + m)$

Implementiert mit min-Heap: $O((n+m) \log n)$

$O(n)$ Array initialisierung

$O(n \log n)$ while Schleife sucht $O(1)$ und löscht $O(\log n)$ kleinsten Wert pro Iteration (n -mal)

$O(m \log n)$ foreach, für jede Neuberechnung pro Kante muss Heap reorganisiert werden (Heapify)

$$O(n + n \log n + m \log n) = O((n+m) \log n)$$

> Noch effizienter mit Fibonacci Heap: $O(m + n \log n)$

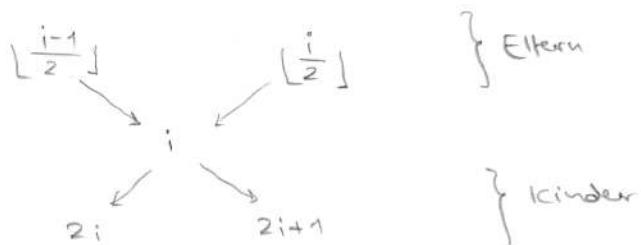
Priority Queue - Datenstruktur:

MIN-HEAP

Ordnung:

$L \geq N \leq R$ ^{nicht} wie bei Biner-Baum

Darstellung als Array:



Wichtig:

wenn man mit Index 1 beginnt

$$L = 2i$$

$$R = 2i+1$$

wenn man mit Index 0 beginnt

$$L = 2i+1$$

$$R = 2i+2$$

Operationen

Heap aus Array bilden $O(n)$

for $i = \lfloor \frac{n}{2} \rfloor$ bis 1 ← alle von Leaf bis root
 Heapify-down(i)

Min / Max lesen $O(1)$

$A[1]$ (= root)

Min / Max entfernen $O(\log n)$ „Extract Min/Max“

root mit Element an letzter Stelle vertauschen

Heapify-down(1)

← Allgemeines
Entfernen
Heapify-down(i)

Element einfügen $O(\log n)$

Einfügen an Stelle $n+1$

Heapify-up($n+1$)

Einfügen:

Heapify-up

Entfernen:

Heapify-down

Initialisieren:

Heapify-down

Heapify-up(i)

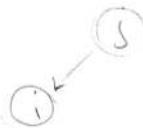
if $i > 1$

$$j = \lfloor \frac{i}{2} \rfloor$$

if $A[i] < A[j]$

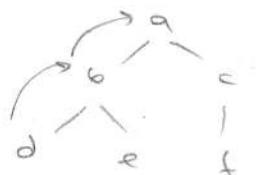
swap ($A[i], A[j]$)

Heapify-up (j)



j sollte $\geq i$ sein

1	2	3	4	5	6
a	b	c	d	e	f



Heapify-down (i)

$$n = A.length - 1$$

if $2i > n$

return

else if $2i < n$

$$\text{left} = 2i$$

$$\text{right} = 2i + 1$$

$$j = \min(A[\text{left}], A[\text{right}])$$

else if $2i = n$

$$j = 2i$$

if $A[j] < A[i]$

swap ($A[j], A[i]$)

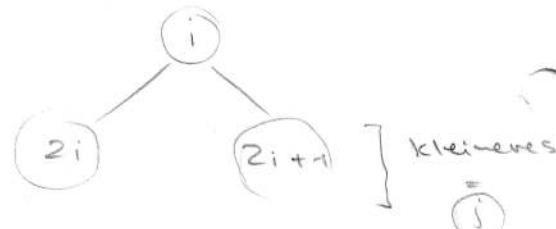
Heapify-down (j)

] leaf erreicht

] \exists links, \exists rechts

] \exists links, $\not\exists$ rechts

Vertausche i mit kleinerem Kind,
Setze bei Kind fort



Heapify-down (i)

Element an Stelle i „rutscht hinab“ wo es hingehört

Heapify-up(i)

Element „rutscht hinauf“ —

Beispiel : Array zu Heap

Rep 1 - 2018

X	5	10	3	8	6	9	4	1	7	2
0	1	2	3	4	5	6	7	8	9	10

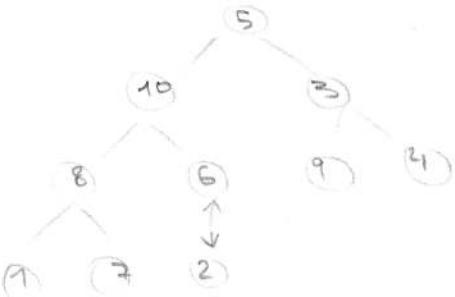


Array entspricht Heap mit verletzten

Heap-Eigenschaften :

$$\left\lfloor \frac{n}{2} \right\rfloor = 5$$

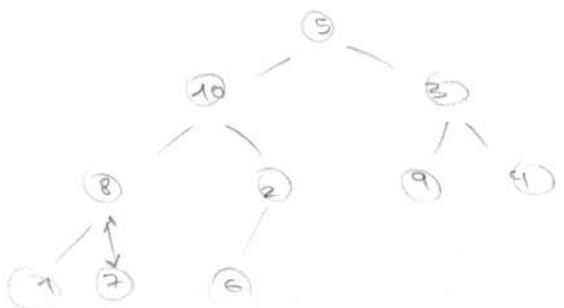
Rechtes Kind : \leq
Linkes Kind : $>$



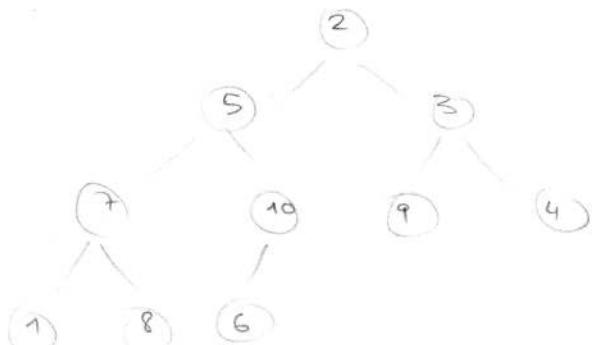
Lösung: init (A, n) :

for $i = \left\lfloor \frac{n}{2} \right\rfloor$ bis 1 $\rightarrow 5$ bis 1

 Heapify-down (A, i)



etc bis



Greedy-Algorithmen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 9. Mai 2019
Vorlesungsfolien



1 / 44

Einleitung

2 / 44

Greedy (muss nicht immer optimal sein)

Münzen wechseln

Aufwagen mit größtem Wert

Interval Scheduling

Jobs mit Index $j = 1, \dots, n$

Startzeit s_j

Endzeit f_j

→ Möglichst viele Jobs hintereinander ausführen

$O(n \log n)$ Sortiere Jobs aufsteigend nach f_j sodass $f_1 \leq f_2 \leq f_3 \dots \leq f_n$

$A \leftarrow \emptyset$

$O(n)$

for $j \leftarrow 1$ bis n

if Job j ist kompatibel zu A

$A \leftarrow A \cup \{j\}$

return A

Maximiere # der Jobs:

Wir fangen immer mit Job an
der am frühesten aufhört damit
die # der Jobs für die es noch
Platz gibt maximal bleibt.

MST - Minimal spanning tree

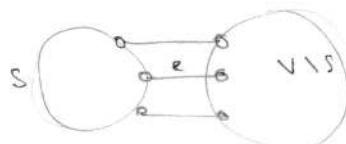
Nicht Hamiltonproblem: geschlossener Pfad, der alle Knoten beinhaltet

Gesucht:

- Spannender Baum (offen, azyklisch) zwischen allen Knoten, mit minimalem Kanten Gewicht.

Basierend auf 3 Lemmata:

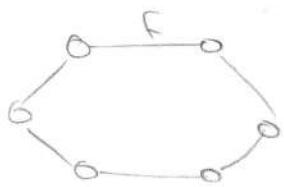
1. Kantschitt Lemma



Angenommen es gibt K Kanten von Teilmenge S zu Teilmenge $V \setminus S$

MSP muss die eine Kante unter diesen beinhalten mit dem min Gewicht

2. Kreislösung



in einem Kreis kann die Kante mit max Gewicht ausgelassen werden

Conclusion:

immer Kante mit min Gewicht nehmen,
keine Kreise bilden

3. Paritätslösung

Def: Kreis: Menge $E(C)$ in der ein Knoten im Pfad doppelt vorkommt

Def: Kanten Schnittmenge: Kanten die nur ein Ende in Teilmenge S haben

Eine Kanten Schnittmenge hat nur eine gerade Anzahl an Kanten mit jedem beliebigen Kreis

Laufrouten:

Dichte Graphen: $m \in \Theta(n^2) \rightarrow$ Prim
Dünne Graphen: $m \in \Theta(n) \rightarrow$ Kruskal

Prim

Heap: $m \log n$

Fibonacci-Heap: $m + n \log n$

Kruskal

Mit Union find, operationen in fest $O(1)$

Der zweite Teil fest $O(n)$

Sortieren:

$O(m \log n)$

MST - Algorithmus von Prim

$O(n \log n)$

1. Initialisiere S mit beliebigen Knoten
2. Wende das Kantschrittkennzeichen ab:
Wähle billigste Kante von Kantschrittkennzeige (S) und
füge Knoten auf anderen Seite S hinzu

Prim (G, c)

foreach ($v \in V$)

$A[v] \leftarrow \infty$

Priority Queue $Q \leftarrow \emptyset$

foreach ($v \in V$)

$Q.add(v)$

$S \leftarrow \emptyset$

while Q ist nicht leer

$u \leftarrow \text{minimales Element aus } Q$

$S \leftarrow S \cup \{u\}$

foreach Knoten v adjazent zu u

if $v \notin S$ und $c_e = (u, v) < A[v]$

$A[v] = c_e$

Initialisierung

MST - Algorithmus von Kruskal

1. sortiere Kanten absteigend nach Gewicht
2. Wähle so lange sich kein Kreis bildet, nach Kreisferme

Kruskal (G, c)

Sortiere Kanten gewichte $\mathcal{O}(m \log n)$

$T \leftarrow \emptyset$

foreach $u \in V$

erzeuge einelementige Menge mit u

for $i \leftarrow 1$ bis m

$(u, v) = e_i$

if u und v sind in verschiedenen Mengen (Zusammenhangskomponenten)

$T \leftarrow T \cup \{e_i\}$

vereinige u und v

return T

1. Möglichkeit: BFS

2. Möglichkeit: Union-Find

Detestructure

Union-Find - Datenstruktur

DDM „Dynamische Disjunkte Mengen“

$S = \{S_1, S_2, \dots, S_k\}$

↑

Jede Menge hat einen repräsentativen Knoten

makeSet (v) Erzeugt Menge $\{v\} = S_v \rightarrow v$ ist repräsentant

union (v, w) Merges $S_v \cup S_w \rightarrow v \text{ xor } w$ ist repräsentant

findset (v) liefert repräsentanten der Menge S mit $v \in S$

Divide-and-Conquer

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 2. April 2019
Vorlesungsfolien



1 / 74

Algorithmen: Paradigmen

Greedy: Erstelle inkrementell eine Lösung, bei der nicht vorausschauend ein lokales Kriterium zur Wahl der jeweils nächsten hinzuzufügenden Lösungskomponente verwendet wird.

Divide-and-Conquer: Teile ein Problem in Teilprobleme auf. Löse jedes Teilproblem unabhängig und kombiniere die Lösung für die Teilprobleme zu einer Lösung für das ursprüngliche Problem.

2 / 74

Divide and Conquer

Mergesort

1. Teilt rekursiv in 2 Teile

2. Sortiert beim mergen indem kleinere Zahl von beiden Seiten im Merge-Array abgelegt wird

Laufzeit

$C(n) \dots \# \text{ comparisons bei } n \text{ Elementen}$

$$C(n) \leq \begin{cases} 0 & \text{wenn } n=1 \\ C(\lfloor n/2 \rfloor) + C(\lfloor n/2 \rfloor) + n & \\ L & R & \text{verschmelzen} \end{cases}$$

Laufzeit $\Theta(n \log n)$
Speicher $\Theta(n)$

Quicksort

1. wähle Pivotelement
2. Gehe von links nach rechts und stelle alle Elemente \leq pivot links und \geq rechts
3. Wiederhole für linke und rechte Hälfte

Laufzeit

Höhe $\Theta(\log n)$

Auf jeder Ebene $\Theta(n)$ Vergleiche

$\Theta(n \log n)$ Worst case aber $O(n^2)$ wenn pivot falsch gewählt

Laufzeit $\Theta(n \log n)$

Speicher $\Theta(n)$

(besser als Mergesort)
durchschnittlich

$\Theta(\log n)$

Conclusion

Laufzeit: Mergesort

worst $\Theta(n \log n)$

avg $\Theta(n \log n)$

best $\Omega(n \log n)$

Quicksort

n^2

$n \log n$

$n \log n$

Speicher:

worst $\Theta(n)$

n

avg $\Theta(n)$

$\log n$

best $\Omega(1)$

$\log n$

Stabilität:

✓

✗

Wichtig:

Sortieralgorithmen können nicht effizienter als $n \log n$ sein

Inversionen zählen

Definition:

Index $i < j$

Wert $a_i > a_j$

1. Teile Liste in 2 Hälften

2. Summiere: Inversionen links, Inversionen rechts, Inversionen zwischen beiden Hälften

↳
nur möglich wenn beide Hälften
sortiert sind.

Sort-and-count (L)

Teile in 2 Hälften $\{A, B\}$

$(r_A, A) \leftarrow \text{sort-and-count}(A)$

$(r_B, B) \leftarrow \text{sort-and-count}(B)$

$(r, L) \leftarrow \text{merge-and-count}(A, B) \leftarrow \text{bereits sortiert}$

return $r_A + r_B + r$ sowie sortierte Liste L

Merge-and-count (A, B)

Mergesort, dass beim mergesort Inversionen zählt

Count++ wenn $a_i \in A > b_j \in B$

Eigentlich = Mergesort

Laufzeit

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + G(n) \rightarrow O(n \log n)$$

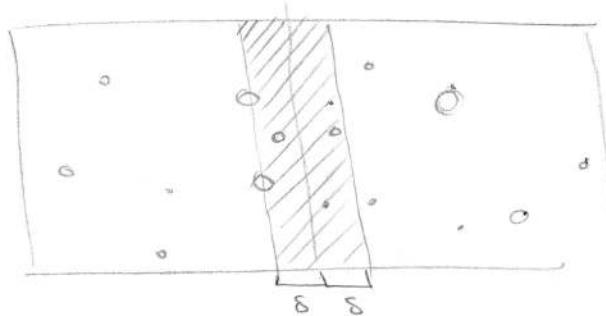
Dichtestes Punktepaar

in euklidischem Raum

Wenn sich alle Punkte auf einer Linie befinden (1D-Version): $O(n \log n)$

Algorithmus:

1. Teile mit Linie, sodass
in jeder Hälfte ca. gleich
Viele sind
2. Berechne Rekursiv L, R
3. Berechne Mitte
4. returniere Max (L, R, Mitte)



→ beobachte nur Teilraum mit Abstand $\min(L, R)$ von Mittellinie = δ
Sortiere Punkte nach y-Koordinate
bestimme Greedy eine Lösung → verkleinere δ wenn möglich

Wichtig: Im Teilraum sind in keiner Hälfte Punkte näher zueinander als δ

$O(n \log^2 n)$

Suchbäume

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019

Letzte Änderung: 9. Mai 2019

Vorlesungsfolien



1 / 74

Wörterbuchproblem

Wörterbuch:

- Als Wörterbuch wird eine Menge von Elementen eines gegebenen Grundtyps bezeichnet, auf der man die Operationen Suchen, Einfügen und Entfernen ausführen kann.
- Für jedes Element wird ein Schlüssel k (*key*) und seine Nutzdaten gespeichert.
- Wir nehmen hier beispielhaft $k \in \mathbb{N}$ an.
- Alle Elemente sind über den Schlüssel identifizierbar.
- Die Operationen sind nur vom Schlüssel abhängig, sodass wir zur weiteren Vereinfachung annehmen, dass ein Wörterbuch aus einer Menge ganzzahliger Schlüssel besteht.

Wörterbuchproblem: Finde eine geeignete Datenstruktur zusammen mit möglichst effizienten Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln.

2 / 74

Schützen

Durchmusterungen (Traversals)

In order L N R

~~Preorder~~ Z N

Postorder L R Z

Piñere Suchbäume

۱۷۲۴

Bestimmung von Vorgänger | Nachfolger

Successor (n) :

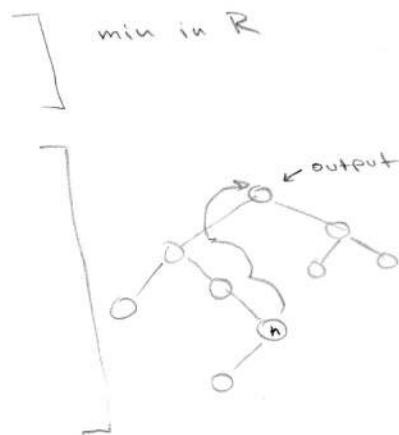
```

if n.right ≠ null
    return minimum(n.right)

if n.right = null
    p ← n.parent
    while p ≠ null & n = p.right
        n ← p
        p ← p.parent

return p

```



`predecessor(v)`:

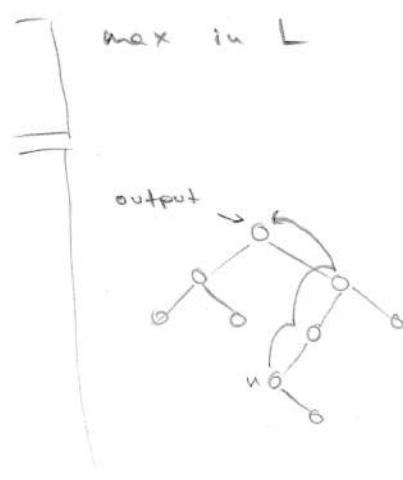
```

if u.left != null
    return Maximum(u.left)

if u.left == null
    p <- u.parent
    while p != null & u = p.left
        u <- p
        p <- p.parent

return p;

```



Einfügen

nech Node suchen und an richtiger Stelle einfügen

Entfernen

Case 1: 0 Kinder = leaf

abschneiden



Case 2: 1 Kind

wie bei verketeter Liste entfernen



Case 3: 2 Kinder

ersetzen Knoten den wir löschen wollen

mit successor / predecessor und löschen

dieses dann (bis es case 1 oder 2 ist)

Laufzeit

Zeitaufwand für

- Suchen
- Einfügen
- Entfernen
- min
- max
- successor
- predecessor

$O(h)$ $h = \text{höhe}$

Wenn Baum zu Liste entartet $O(n)$

Vollständig balancierter Baum (AVL) in $O(\log_2 n)$

Balancierte Pinnreihen: Höhenbalancierung

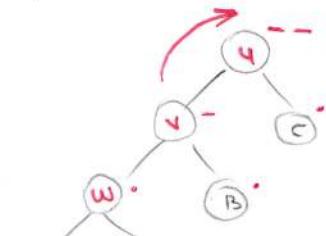
AVL-Bäume

Balance := R.Tiefe - L.Tiefe

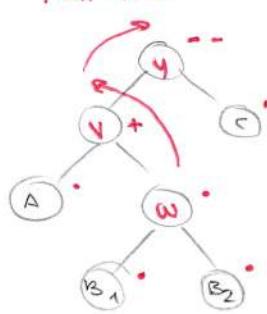
erlaubt: $\{-1, 0, 1\}$

nicht erlaubt: $\{+2, -2\}$

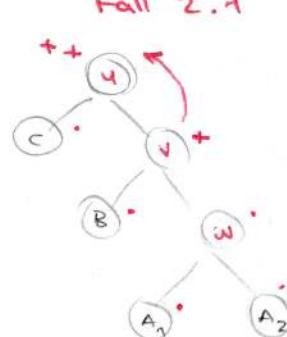
Fall 1.1



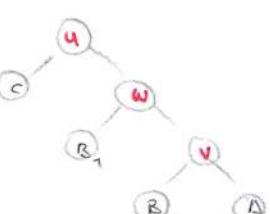
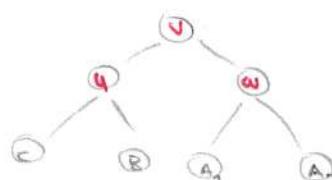
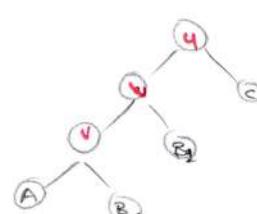
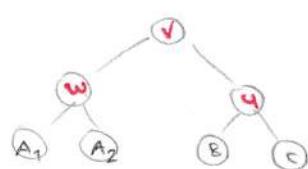
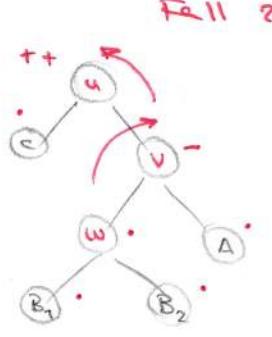
Fall 1.2



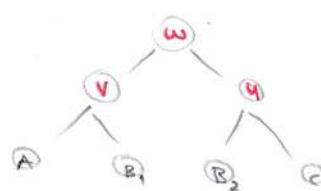
Fall 2.1



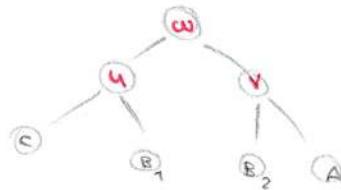
Fall 2.2



↓ Fall 1.1



Fall 2.1



Rotation

Beispielsweise Fall 1.1: r.child [u] wird zu l.child [v]

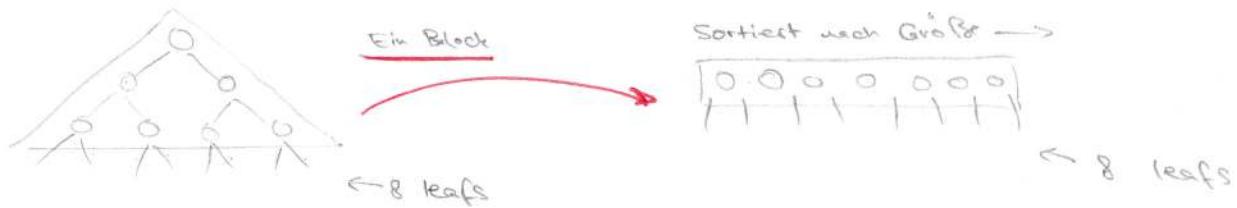
v wird root

logarithmische Laufzeit $O(\log_2 n)$

Bei Einfügen/ Löschen maximal 2 Rotationen

B-Baum

Lesen von Child-Nodes extrem aufwendig bei externen Speichern,
deshalb in einem Block / Node möglichst viel speichern



Constraints:

B-Baum mit Ordnung m (m Kinder, $m-1$ keys (Eintrag) per Node)

1. Alle Leafs sind in derselben Ebene und sind leer

2. Maximal m Kinder und $m-1$ keys aber bei
($k+1$ Kinder : k keys)

3.1. Wurzel hat min 2 Kinder, min 1 key

3.2. Innere Knoten haben min $\lceil \frac{m}{2} \rceil$ Kinder

4. Einträge sind sortiert:

Knoten mit 1 keys und $l+1$ Kindern

s_1, s_2, \dots, s_l v_0, v_1, \dots, v_l

Beispiel



Man erkennt:

Alle Schlüssel in T_{v_i-1} sind $\leq s_i$

s_i ist \leq alle Schlüssel in T_{v_i}

$T_{v_0} \leq s_1$

$s_1 \leq T_{v_1}$

Implementierung

p.l Schlüssel - Anzahl

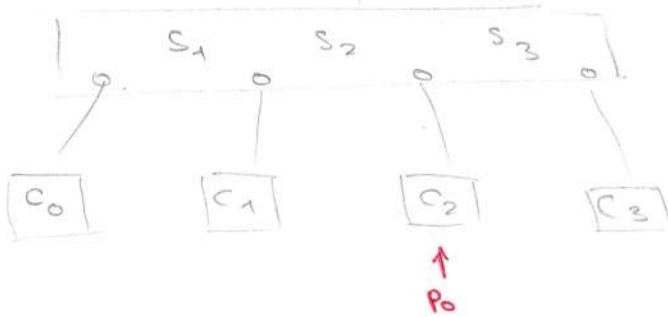
p.key[1] → p.key[l] Schlüssel s_1, \dots, s_l

p.info[1] → p.info[l] Zahlen in s_1, \dots, s_l

p.child[0] → p.child[l] Kinder

> bei Blättern: $p.l = 0$

Einfügen



$w = 4$ → 3 Kreis
→ 4 Kinder

Hier:

$$p.i = 3$$

$$p.key[1] = s_1$$

$$p.key[2] = s_2$$

$$p.key[3] = s_3$$

$$p.child[0] = c_0$$

$$p.child[1] = c_1$$

$$p.child[2] = c_2$$

$$p.child[3] = c_3$$

1. wir suchen im Baum den key den wir einfügen möchten und landen bei Leaf P_0

hier: $P_0 = c_2$

)

2. P ist Vorgänger Node von P_0 und heißt P

P_0 ist c_i von P

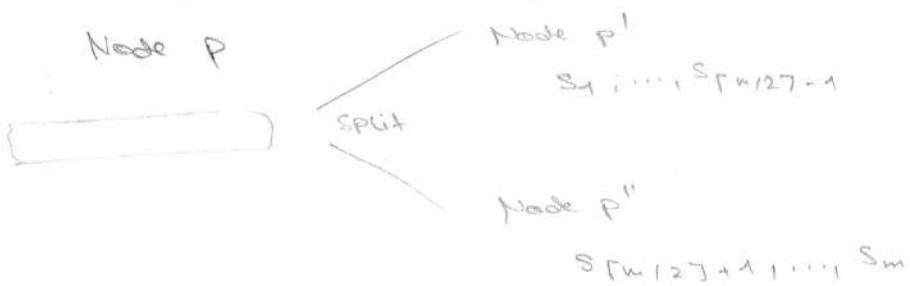
hier: $P_0 = c_2 \quad i=2$ also ist P_0 zwischen s_2 und s_3

3a. Es gibt Platz für mehr keys!

Deshalb zwischen s_i und s_{i+1} einfügen

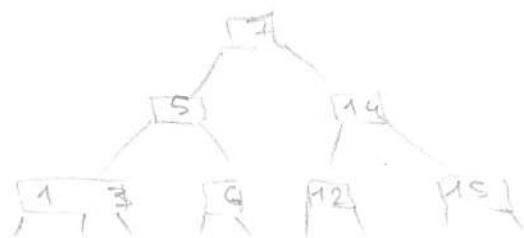
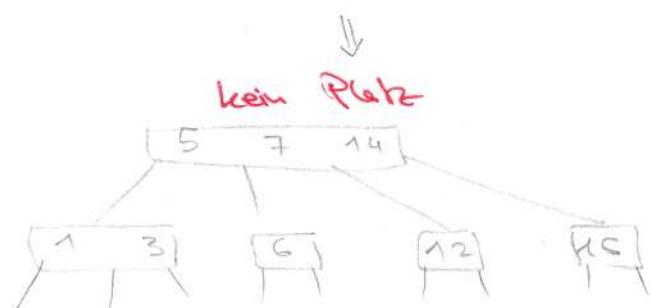
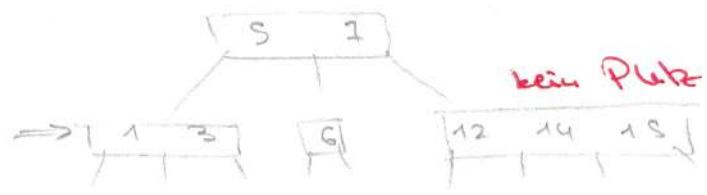
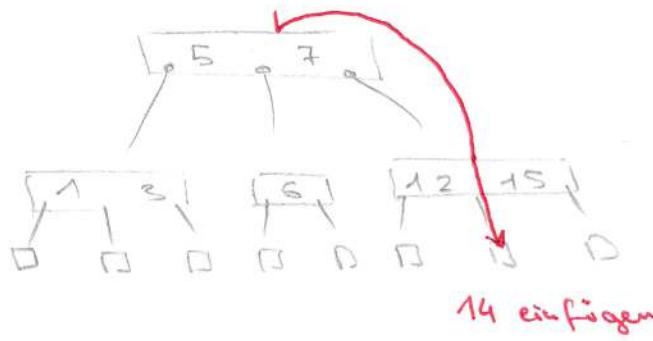


3b. Es gibt keinen Platz, deshalb Node in 2 Teile splitter



und $s_{m/2}$ in Vorgänger von P einfügen
(und das dann wieder splitten)

B-Bäume wachsen immer nur an der Wurzel!



Entfernen

Wir wollen B-Tree optimieren, deshalb pro Node nur $\lfloor \frac{m}{2} \rfloor$ bis m Schlüssel erlaubt.

Wenn mehr: splitten

Wenn weniger: mergen

1. key im Baum suchen: k'

2. Entfernen

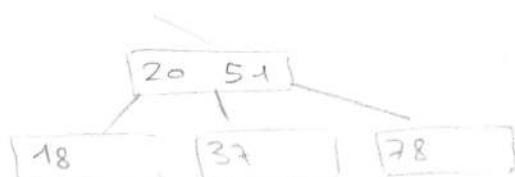
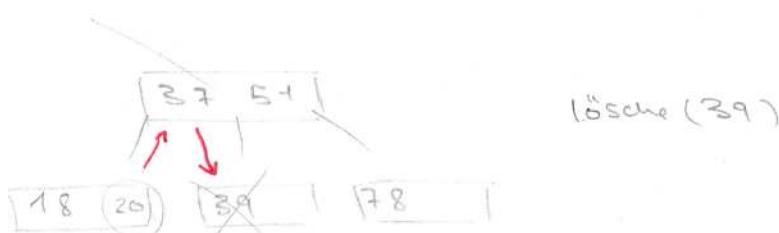
case 1: k' ist in untersten Ebene

nach entfernen nicht weniger als $\lfloor \frac{m}{2} \rfloor$ keys

Case 2: k' ist in unterster Ebene

nach entfernen weniger als $\lfloor \frac{m}{2} \rfloor$ keys

a) Nachbar links oder rechts hat genug keys zum Herausziehen

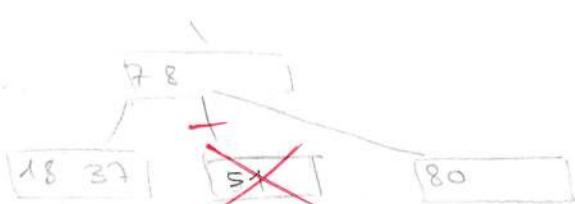
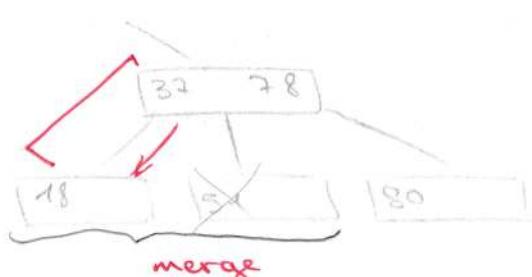


b) Nachbarn haben nicht genug zum Herausziehen

1. Eltern zu Kind

2. leere Node löschen

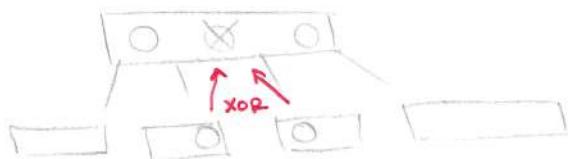
3. mergen



case 3: k' nicht zu unterster Ebene

k' entfernen, mit Ersetzschlüssel ersetzen

Prozess neu auffangen und Ersetzschlüssel löschen



Ersetzschlüssel:

größter im linken Kind

kleinster im rechten Kind

Eigenschaften

$$\# \text{leaves im Baum} + 1 = \# \text{keys}$$

Proof:

Wurzel hat min 1 key, darf aber auch unter $\lfloor \frac{n}{2} \rfloor$ haben

$$\text{Höhe} = 1$$

Wurzel hat k leaves und $k-1$ keys wobei
 $2 \leq k \leq m$

IA

Wir setzen voraus, dass das auch für Höhe h gilt

Wir zeigen, dass es für Höhe $h+1$ gilt:

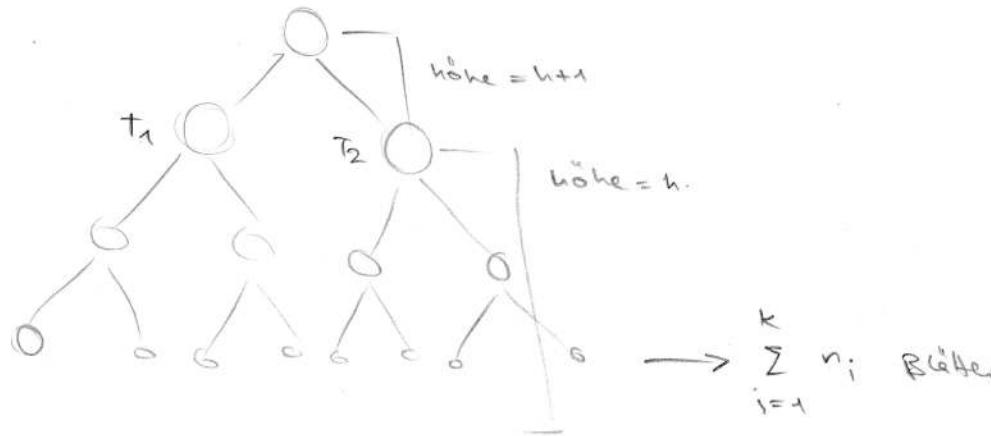
IV

$$\text{Höhe} = h+1$$

k Unterbäume T_1, \dots, T_k mit Höhe h , für die
 das gilt, mit n_1, \dots, n_k Blättern

IS

je $(n_i - 1)$ Schlüssel



$$\sum_{i=1}^k n_i - 1 \text{ Schlüssel}$$

Mit einfachen Worten:

Jede einzelne Ebene des Baumes hat

Pro Node 1 Schlüssel weniger als Blätter

und das trifft deshalb sowieso in jedem Raum \mathbb{R}^n

max # leafs: Wurzel genau 2 children, alle anderen $\lceil \frac{m}{2} \rceil \Rightarrow N_{\min} = 2 \lceil \frac{m}{2} \rceil$

min # leafs: Alle Nodes in children: $m^h \Rightarrow N_{\max} = m^h$

↑
linke und rechte Seite des W

Allgemein gilt:

Wenn Baum insgesamt N Schlüssel und $N+1$ Blätter

Blätter:

$$N_{\min} = 2 \lceil \frac{m}{2} \rceil^{h-1} \leq N+1 \leq N_{\max} = m^h$$

$$\underbrace{\log_m(N+1)}_{\min} \leq h \leq \underbrace{1 + \log_{\lceil \frac{m}{2} \rceil}\left(\frac{N+1}{2}\right)}_{\max}$$

(wenn alle halbgefüllt sind)

Deshalb

$$h = \Theta(\log N)$$

Alternativer Beweis:

Operationen auf balancierten binär-Baum: $O(\log_2(n))$

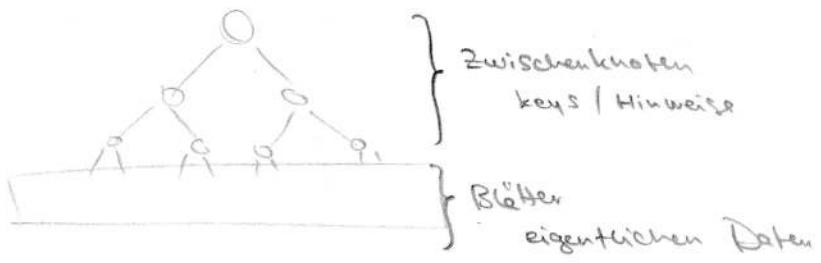
Baum mit 3 Kindern: $O(\log_3 n)$

4 Kinder: $O(\log_4 n)$

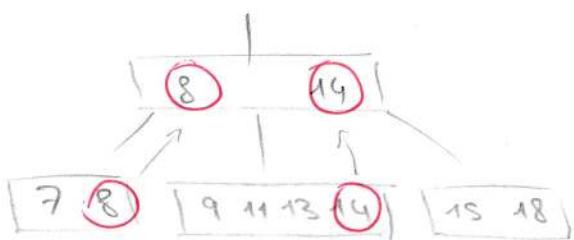
B*-Bäume

Parameter k^* ist von m unabhängig

"klassischer" B-Baum, aber Blätter dürfen $[k^*; 2k^*]$ Einträge haben.



Schlüssel



Schlüssel ist der größte Wert vom linken Kind

In der Praxis:

m wird deutlich höher gesetzt als k^* weil es quasi nur wie ein Telefonbuchindex agiert

und die eigentlichen Daten (z.B. Nummer, Name, Adresse) werden vollständig in leafs gespeichert

Hashing

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019

Letzte Änderung: 7. Mai 2019

Vorlesungsfolien



1 / 41

Hashing

Hashing:

- Alternative Möglichkeit zum Wörterbuchproblem
(siehe Kapitel über Suchbäume).
- Beobachtung: Im Allgemeinen ist lediglich eine kleine Teilmenge K aller möglichen Schlüssel \mathcal{K} gespeichert.
- Idee: Statt in einer Menge von Datensätzen durch Schlüsselvergleiche zu suchen, ermittle die Position eines Elements im Speicher (bzw. einem Array) durch eine arithmetische Berechnung.

2 / 41

Hashing

Wenn Kollision frei dann:

Suchen, Einfügen, Entfernen bei $\Theta(1)$

Sonst worst-case $\Theta(n)$

Hashfunktionen

„Modulo-Methode“

$$h(k) = k \bmod m \quad m \text{ muss Primzahl sein}$$

„Multiplikationsmethode“

irrationale Zahl $A \rightarrow$ optimal: abgerundeter Schritt

$$h(k) = \lfloor m \cdot \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \rfloor$$

Kollisionsbehandlung

„Verketzung der Überläufer“

Bei Kollision in Hashtabelle \rightarrow einfache verketzte Liste

„Offenes Hashverfahren“

1. Grundlegende Idee:

Flags = { frei, besetzt, wieder frei }

bei Kollision, nächste freie Stelle in Array benutzen: „Sondierung“

engl.: probing

Statt Hashfunktion:

\rightarrow Sondierungsfunktion $h(k, i)$

bestimmt Suchroutenfolge nach freien Plätzen

$$i \in [0; m-1]$$

2. Lineare Sondierung

$$h(k, i) = (h'(k) + i) \bmod m$$

Berechnung der \varnothing Zeit für erfolgreiche Suche:

\sum Suchiterationen für alle Einträge

Einträge

Probleme:

Quasi Verstopfung indem belegte Teilstücke immer länger werden: Primary Clustering

$$\text{Belegungsfaktor } \alpha = \frac{\# \text{ Elemente}}{\# \text{ Speicherplätze}}$$

Wenn $\alpha \rightarrow 1$ dann primary clustering wahrscheinlicher
Deshalb ist eine Gleichverteilung (Wahrscheinlichkeitstheorie) wichtig

Uniform hashing = Gleichverteilung
(schwer erreichbar aber gut approximierbar)

3. Quadratisches Sondieren

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Problem: secondary Clustering

4. Double Hashing

$h_1(k)$: hashfunktion 1

$h_2(k)$: hashfunktion 2

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

(fast uniform!)

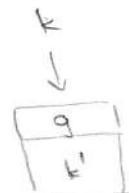
- ↳ Darf nicht m teilen
- ↳ Darf nicht 0 sein

5. Verbesserung nach Brent

$$j = (g + h_2(k)) \bmod m$$

$$j' = (g + h_2(k')) \bmod m$$

- Wenn j frei oder j' besetzt
dann setze k auf j
- Wenn j besetzt und j' frei
dann setze k auf g und
dann setze k' auf j'



Vorteil: auch wenn $\alpha=1 \rightarrow \Theta(1)$

Praktische Datenstrukturen in Java
Ein Überblick

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 9. Mai 2019



Polynomialzeitreduktion

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 14. Mai 2019
Vorlesungsfolien



ALGORITHMS AND
COMPLEXITY GROUP

1 / 81

Effizient lösbar Probleme

Wiederholung aus dem Kapitel „Analyse von Algorithmen“:
Wir bezeichnen ein Problem als effizient lösbar, wenn es in
Polynomialzeit gelöst werden kann. Für ein solches Problem
existiert ein Algorithmus mit einer Laufzeit $O(n^c)$

- n = Eingabegröße (z.B. in Bits)
- c = konstanter Exponent

Effizient lösbare Probleme werden auch als handhabbar (*tractable*)
bezeichnet.

Cobham–Edmonds Annahme:

- Die Annahme, Handhabbarkeit mit Lösbarkeit in
Polynomialzeit gleichzusetzen, geht auf Alan Cobham and
Jack Edmonds zurück, die das in den 1960er-Jahren
vorgeschlagen haben.
- Diese Annahme hat sich weitgehend durchgesetzt und die
Informatikforschung der letzten 50 Jahre geprägt.

2 / 81

Polynomialzeit - Reduktion

Problemtypen

Gibt es Lösung mit Eigenschaft e?

JA/NEIN-PROBLEM (Decision-Problem): returniert nur 1 Bit

FUNKTIONALES PROBLEM: Finde Lösung mit Eigenschaft e!

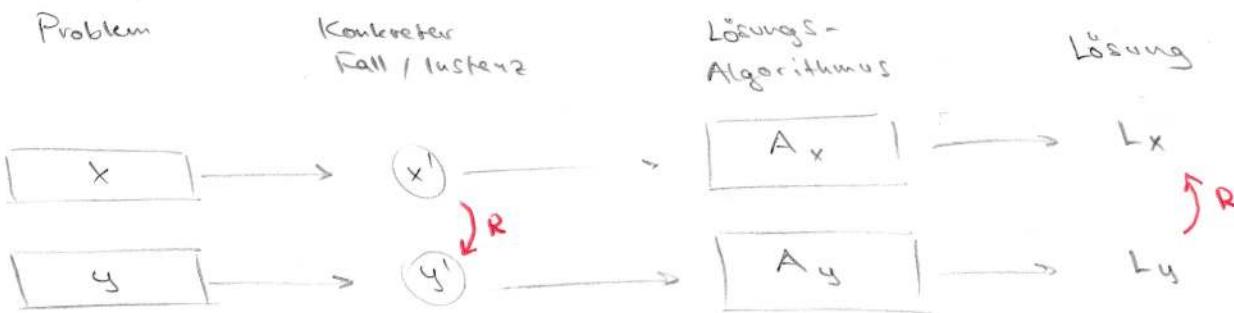
OPTIMIERUNGSPROBLEM: Finde optimale Lösung

Reduktion

Quasi Übersetzung, stellt Verbindung zwischen JA/NEIN-PROBLEMMEN her

Polynomielle Reduktion:

Kann x auf y "übersetzt" werden und wir können effizient y lösen,
dann können wir auch x effizient lösen



(Genauso dann wenn $L_y = \text{JA}$ ist
 $L_x = \text{JA}$)

Wichtig:

R ist selbst ein Reduktions-Algorithmus
mit Eingabe x'

in polynomieller Zeit: Verkettung der Laufzeiten

Reduktion $O(x'^a)$

$A_y \quad O(O(x'^a)^b)$

$O(x'^{a \cdot b}) = \text{Polynomiell}$

Notation:

Problem $X \leq_p$ Problem Y

X ist mindestens genauso-schwer wie Y (sonst ließe es sich nicht mit Y lösen!)

Regeln:

$$1. X \leq_p Y \wedge Y \leq_p X \Rightarrow X \equiv_p Y$$

2. Wenn X nicht polynomiell lösbar und $X \leq_p Y$ dann Y nicht polynomiell lösbar

$$3. X \leq_p Y \wedge Y \leq_p Z \Rightarrow X \leq_p Z$$

Eigenschaften der Reduktion

Korrektheit

Korrekte Abbildung: JA auf JA, NEIN auf NEIN

Polynomielle Laufzeit

$$\mathcal{O}(n^c)$$

Polynomialzeit - Reduktion

Problemtypen

Gibt es Lösung mit Eigenschaft e ?

JA/NEIN-PROBLEM (Decision-Problem): returniert nur 1 Bit

FUNKTIONALES PROBLEM: Finde Lösung mit Eigenschaft e !

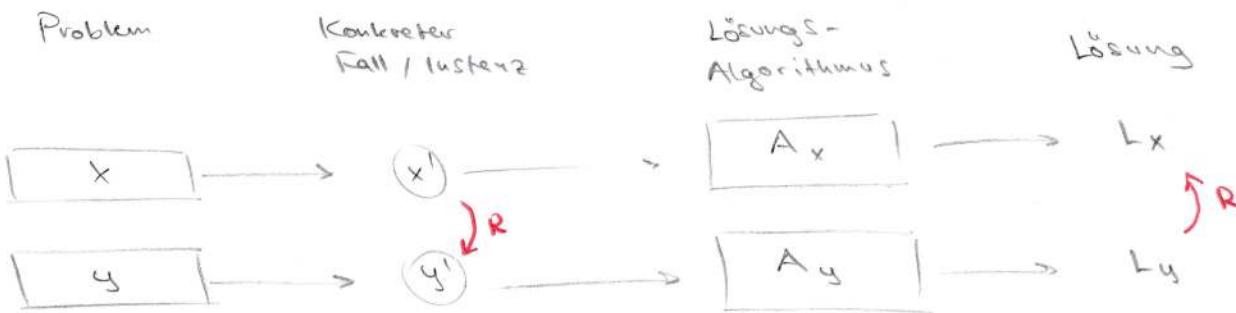
OPTIMIERUNGSPROBLEM: Finde optimale Lösung

Reduktion

Quasi Übersetzung, stellt Verbindung zwischen JA/NEIN-PROBLEMMEN her

Polynomielle Reduktion:

Kann X auf Y "übersetzt" werden und wir können effizient Y lösen,
dann können wir auch X effizient lösen



(Genau dann wenn $L_y = \text{JA}$ ist
 $L_x = \text{JA}$)

Wichtig:

R ist selbst ein Reduktions-Algorithmus
mit Eingabe x'

in polynomialer Zeit: Verkettung der Laufzeiten

Reduktion $O(x^a)$

$A_y \quad O(O(x^a)^b)$

$$O(x^{a+b}) = \text{Polynomial}$$

Notation:

Problem $X \leq_p$ Problem Y

X ist mindestens genauso-schwer wie Y (sonst ließe es sich nicht mit Y lösen!)

Regeln:

$$1. X \leq_p Y \wedge Y \leq_p X \Rightarrow X \equiv_p Y$$

2. Wenn X nicht polynomiell lösbar und $X \leq_p Y$ dann Y nicht polynomiell lösbar

$$3. X \leq_p Y \wedge Y \leq_p Z \Rightarrow X \leq_p Z$$

Eigenschaften der Reduktion

Korrektheit

Korrekte Abbildung: JA auf JA, NEIN auf NEIN

Polynomielle Laufzeit

$$O(n^c)$$

Reduktionsstrategien

(1.) Reduktion durch einfache Äquivalenz

2. Reduktion eines Spezialfalls auf den allgemeinen Fall

3. Reduktion durch Kodierung mit Gadgets

Reduktion durch einfache Äquivalenz

Independent Set

Knotenmenge $\leq k$ in der keine adjazent sind

Vertex Cover

Knotenmenge $\leq k$ in der alle adjazent sind

INDEPENDENTSET \equiv_p VERTEXCOVER

$$\begin{aligned} |V| - |IS| &= |VC| \\ |V| - |VC| &= |IS| \end{aligned} \quad \left\{ \begin{array}{l} \text{"Konversionsternne"} \\ \text{oder} \end{array} \right.$$

Deshalb Reduktion beidseitig mit $(G, n-k)$ $O(1)$ konstant

Nicht-Blockierer (MNB)

Gewichteter Graph:

Maximierung des Kantengewichts der Knoten die entfernt werden

können sodass es min 1 Pfad zwischen allen Knoten gibt bei $\leq k$ Knoten

MST*

MST mit $\leq k$ Kanten

MNB \equiv_p MST*

Konversionsternne

$$K' = \sum \text{aller Gewichte}$$

Reduktion durch

$$(G, K' - k)$$

Reduktion des Spezialfalls auf den allgemeinen Fall

SETCOVER

Mengenüberdeckungsproblem

Gesamtmenge U

Menge an Teilmengen $S = \{S_1, S_2, S_3, S_4, \dots, S_m\}$

Ein SetCover ist eine Menge an Teilmengen $C \subseteq S$, sodass deren Vereinigung

$$C_1 \cup C_2 \cup \dots \cup C_k = U$$

Existiert ein Setcover mit $\leq k$ Teilmengen die gemeinsam U bilden?

VERTEXCOVER \leq_p SETCOVER

Polynomielle Reduktion: Erzeugung eines Instanz für SetCover

$$k = k$$

$$U = E \text{ aus } G = (U, E)$$

Für jedes $v \in V$

$$S_v = \{ \text{alle Kanten incident zu } v \}$$

S ist die Menge aller Kanten pro Knoten

↓
Können mit k Mengen / Knoten alle Kanten / U abgedeckt werden?

Überprüfung der Reduktion

✓ Polynomiell

✓ Korrekt

Man spricht von „Reduktion des Spezialfall auf den allgemeinen Fall“ da:

$VC \leq_p SC$ immer möglich

~~$SC \leq_p VC$~~ nicht immer möglich, wenn Elemente aus Mengen z.B.
doppelt vorhanden

Reduktion durch "Gadgets"

SAT - Satisfiability - Problem

Literal: boolische Variable, kann true oder false sein (x_i oder \bar{x}_i)

Klausel: $C_3 = (0 \vee 0 \vee 0)$

Konjunktive Normalform (KNF): $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

Wahrheitsbelegung (Truth assignment)

Funktion f , die Φ erfüllt

3-SAT

3-SAT: jede Klausel 3 Literale

→ Mit k Klauseln pro Φ

3-SAT \leq_p INDEPENDENT-SET

Reduktion eines 3-SAT Instanz mit k Klauseln in einem Graph

- Ein Knoten pro Variable
- Alle Variablen in einer Klausel zu Δ verbinden
- Eine Variable zu Negation verbinden falls sie vorkommt

✓ Polynomell

✓ korrekt:

Φ erfüllbar $\Leftrightarrow \exists$ Independent-Set mit $k \geq 3$ gefüllten Knoten
(gefüllt = Literal ist true)

weil

1. Komplementäre Variablen verbunden

$$x_1 = \text{true} \Rightarrow \bar{x}_1 = \text{false}$$

$$x_1 = \text{false} \Leftrightarrow \bar{x}_1 = \text{true}$$

2. Nur 1 wahre Variable pro Klausel weil
3 Ecken

Mit Gadget simuliert man quasi die andere Instanz und lädt sie auf

Zusammenfassung

Einfeche Äquivalenz

$$\text{INDEPENDENT SET} \leq_p \text{VERTEX COVER}$$

$$\text{MNB} \leq_p \text{MST}^*$$

Spezialfall auf allgemeinen Fall

$$\text{VERTEX COVER} \leq_p \text{SET COVER}$$

Gadgets

$$3\text{-SAT} \leq_p \text{INDEPENDENT SET}$$

Beispiel für Transitivität

$$3\text{-SAT} \leq_p \text{IS} \leq_p \text{VC} \leq_p \text{SC}$$

OPTIMIERUNGSPROBLEME

Können JA/NEIN PROBLEME lösen und sich von ihnen lösen lassen
wenn sie „self reduzible“ sind

Beispiele für NP-Probleme

VERTEX-COVER

Ja-Instanz: (Problem welches Ja als Lösung hat)

→ Zertifikat: Vertex Cover Menge S mit der Größe $|S| \leq k$

→ Zertifizierer: Prüft in polynomieller Zeit ob S eine gültige Lösung ist.

1. Prüfe von S die Größe

2. Prüfe ob jede Kante einen Endpunkt in S hat

SAT

Zertifikat: Wahrheitsbelegung die \emptyset erfüllt

→ Zertifizierer: Wahrheitsbelegung logisch kontrollieren

HAM-CYCLE

Gesucht: Kreis welcher alle Knoten genau $1x$ beinhaltet

Zertifikat: Hamiltonkreis

→ Zertifizierer: Überprüfe ob Kreis alle Kanten enthält,
Überprüfe ob Kreis

TSP - Traveling Salesman Problem

Gesucht: Minimale Tour die alle Orte beinhaltet \rightarrow vollständiger Graph,
TSP*: Ja/Nein Problem, existiert Tour mit $\leq k$ Kanten?

Jede Nade eine
Stadt

HAM-CYCLE \leq_p TSP*

Reduktion:

G ist $= (V, E)$ und eine HAM-CYCLE Instanz

1. Erzeuge n Städte mit einer Distanzfunktion

$$d(u, v) = \begin{cases} 1 & \text{wenn } u, v \text{ adjazent} \\ \infty & \text{wenn } u, v \text{ nicht adjazent} \end{cases}$$

KLASSIFIKATION VON PROBLEMEN

P

JA/NEIN-PROBLEME mit polynomiellen Algorithmen

NP

Ein JA/NEIN-PROBLEM $\in NP$ wenn es einen polynomiellen Zertifizierer hat

ZERTIFIKAT / WITNESS

Eingabe Instanz x

Ein Lösungs-Versuch als String t mit polynomieller Länge $|t| \leq p(x)$

Nach mathematischer Definition:

t ist eine Eingabe für eine Funktion „ $\exists t \psi(t)$ “ die Existenzaus sagen überprüft die $\exists t \psi(t) = \text{true}$ ergeben sollte (muss noch von Zertifizierer bewiesen werden)

ZERTIFIZIERER / VERIFICATION ALGORITHM

Polynomialzeit - Algorithmus $C(x, t)$ welchen Ja-Instanzen von x verifizieren kann (aber nicht Nein-Instanzen, weil Problem $\in NP$)

NP-SCHWER

„NP-hard“

Ein Problem ist NP-Schwer, wenn:

Für alle Probleme $X \in NP$: $X \leq_p Y \Rightarrow Y \in \text{NP-Schwer}$

Diese Probleme sind mindestens genauso schwer wie die schwersten NP-Probleme.
Wenn man ein NP-Schweres Problem effizient löst hat man alle NP-Probleme gelöst

NP-VOLLSTÄNDIG

„NP-C“ oder „NP-complete“ JA/NEIN-Probleme

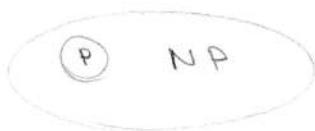
Ein Problem ist NP-Vollständig wenn es gleichzeitig $\in NP$ und $\in NP$ -Schwer ist.

Informell: Kein polynomieller Algorithmus bekannt, aber auch kein Beweis für $\notin NP$

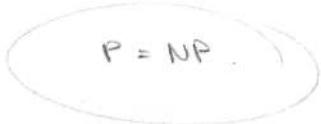
Sind NP-Schweren Problemen ähnlich (lassen sich reduzieren) \rightarrow nicht klassifizierbar

P $\stackrel{?}{=}$ NP

Alle Probleme die sich polynomiell lösen lassen, haben polynomielle Zertifizierer,
also ist $P \subseteq NP$



Wenn aber $P = NP$



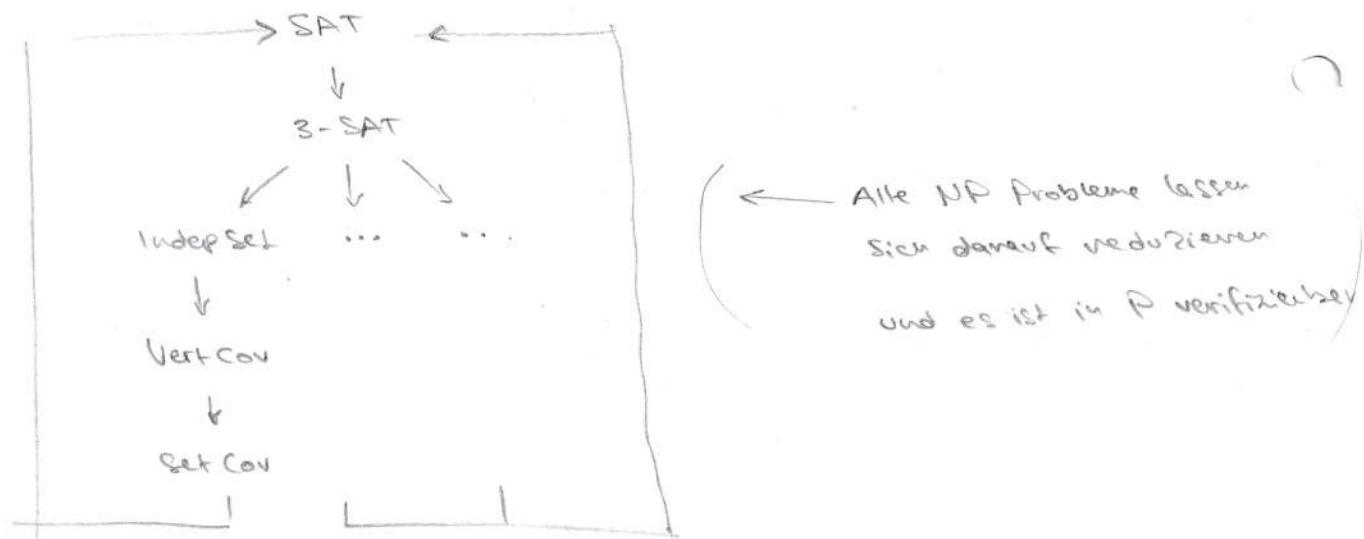
Können sich NP-Vollständige Probleme lösen lassen, da $NP \subseteq P$ sein würde

NP-Vollständigkeit

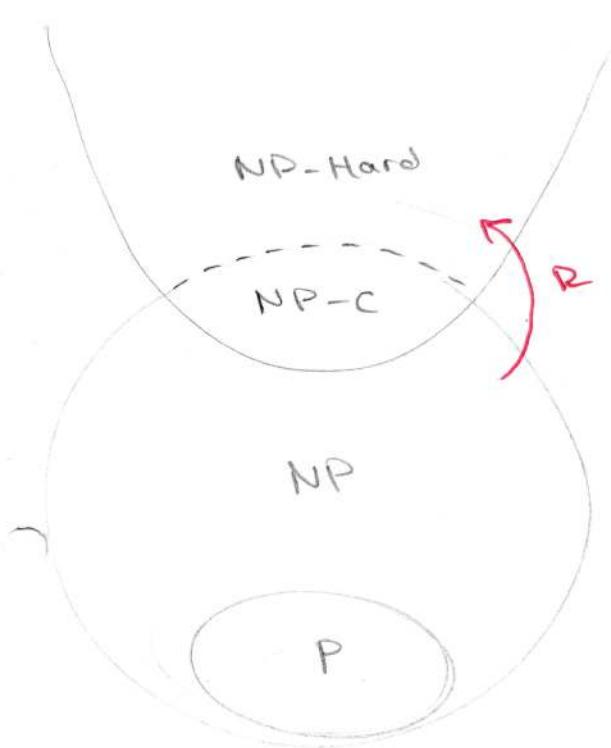
NP-vollständig ↪ $\in NP$: Lässt sich in P-Zeit verifizieren

↪ NP-schwer: Alle NP-Probleme lassen sich darauf übersetzen

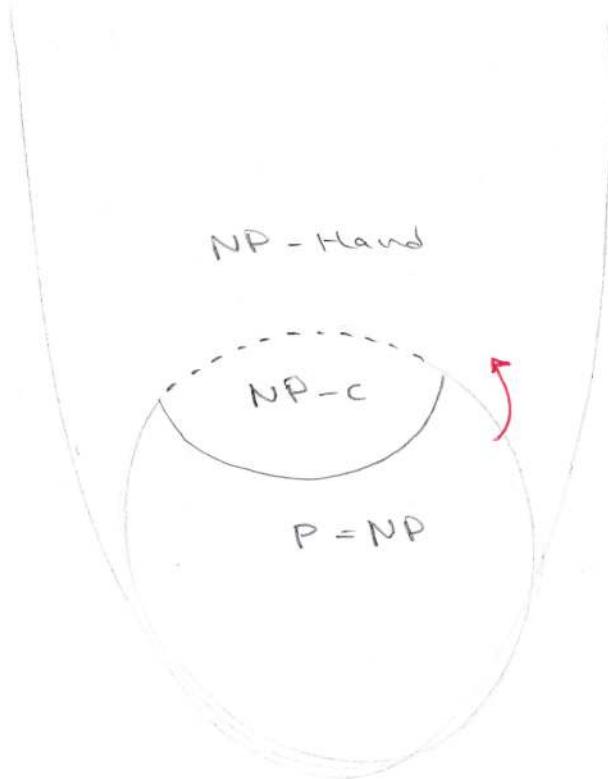
Wenn man ein NP-vollständiges Problem auf ein NP-Problem übersetzt (reduziert), dann können alle NP-C Probleme nur NP oder leichter sein, weil sie sich gegenseitig ineinander übersetzen lassen.



wenn $P \neq NP$



wenn $P = NP$



Wenn man ein NP-schweres (oder NP-vollständiges Problem) in P löst
hat man alle NP Probleme in P gelöst.

Wenn man ein NP-C Problem auf ein NP Problem übersetzt und löst
ist $NP-C = NP$ (beweisbarermaßen unmöglich)

$3\text{-SAT} \leq_p 3\text{-COLOR}$

Wir machen eine Eingabeinstanz für 3-COLOR (basierend auf 3-SAT) die nur färbbar ist, wenn 3-SAT lösbar ist.

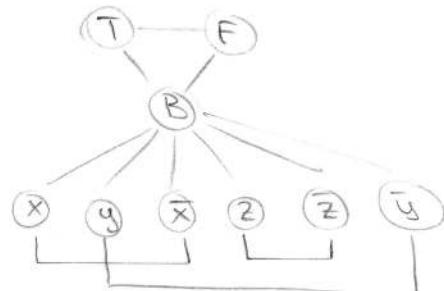
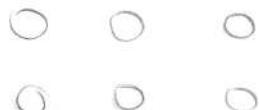
3-SAT:

$$k \text{ Klausel } C_i = (x_1 \vee x_2 \vee x_3)$$

Wobei x true oder false sein kann und in mehreren Klauseln vorkommen darf.

Konstruktion:

- Pro Literal 1 Knoten
- 3 neue Knoten T, F, B die verbunden werden, jedes Literal wird mit B verbunden
- Jedes Literal mit eigener Negation verbinden
- Pro Klausel wird ein Gadet konstruiert mit 6 Knoten:



Diese Gadets werden mit den Litervaten in der jeweiligen Klausel und den Knoten T, F, B verbunden

(siehe Rückseite)

Behauptung: Graph ist 3-färbbar wenn Φ erfüllbar

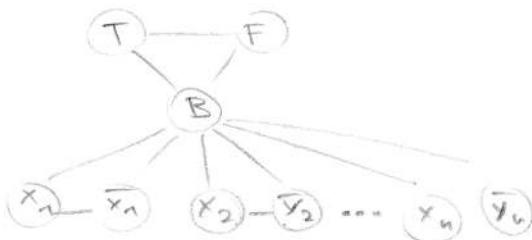
Beweis:

T = grün

F = rot

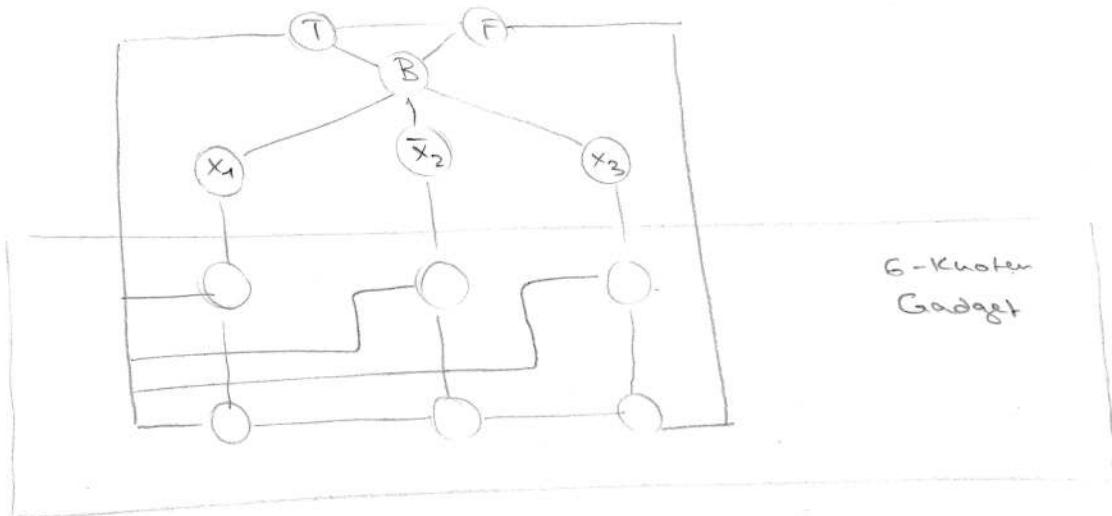
B = blau

Alle Variablen
grün oder rot
True False



- ✓ Dadurch ist es eine gültige Wahrheitsbelegung in der eine Variable nicht gleichzeitig mit der Negation wahr sein kann

Angenommen $C_i = (x_1 \vee \neg x_2 \vee \neg x_3)$



- ✓ Zumindest 1 Literal wird auf true gesetzt, sonst Widerspruch
(siehe Folie)

Anwendung als Beispiel auf Seite 75/81

NP-Vollständigkeit Spezialfälle

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 21. Mai 2019
Vorlesungsfolien



NP-C Probleme sind 21 definierte NP-Hard-Probleme die sich zusätzlich in polynomieller Zeit verifizieren lassen

Dadurch sind sie die schwersten NP-Probleme, auf die sich aber alle NP-Probleme reduzieren lassen

Besondere **Spezialfälle** von NP-C Problemen, lassen sich **in polynomieller Zeit** lösen:

VERTEXCOVER

wenn k angegeben: Spezialfall
/ sonst NP-C

Existiert Teilmenge von n Knoten die mit k Elementen alle Kanten abdeckt?

Brute-Force: $O(n^k) = \binom{n}{k}$

Verifikation: $O(kn)$

$O(k \cdot n^{k+1}) \rightarrow$ undurchführbar

Ziel:

Auf polynomielle Abhängigkeit zu kommen indem k eine kleine Konstante ist

Beispiel:

$$n=1000$$

$$k=10$$

Brute force \rightarrow Never Algorithmus!

$$10^{n^{k+1}} = 10^{39}$$

$$2^k kn = 10^7$$

(unmöglich)

(möglich wenn k klein ist)

Lemme

$$G = (V, E) \quad \textcircled{u} - \textcircled{v}$$

Knoten \textcircled{u} und \textcircled{v} sind adjazent und G hat VertexCover mit k Knoten

Case 1: $G \setminus \{u\}$ hat $|VC| = k-1$

Case 2: $G \setminus \{v\}$ hat $|VC| = k-1$

Lemma: Obere Schranke

$$|E| \leq |\text{VertexCover}| \cdot (|V|-1)$$

Anzahl der Kanten in G ist höchstens so groß wie # Elemente in VC \cdot (# Knoten - 1)

$$|VC| = k$$

$$|V| = n$$

$$|E| \leq k(n-1)$$

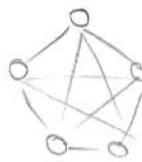


Beweis

Jeder Knoten kann max $\deg(v) = n-1$ haben und $n-1$ Kanten überdecken.

So ist es möglich mehrfache Kanten.

K_5 :



Obere Schranke

Ein vollständiger Graph hat eine maximale Kantenanzahl

$$|V| = 5$$

$$|E| = \frac{n(n-1)}{2}$$

$$|VC| = 5$$

wegen Handshake Lemma /

Algorithmus

VertexCover(G, k)

if $|E| = 0$

return true

if $|E| > k \cdot (n-1)$

return false

$(u, v) \leftarrow$ beliebige Kante

a \leftarrow VertexCover($G \setminus \{u\}, k-1$)

b \leftarrow VertexCover($G \setminus \{v\}, k-1$)

return a oder b

} Alle Kanten abgedeckt $O(1)$

} Nicht alle Kanten abgedeckt $O(n)$

} 2. Knoten entfernen und alle
incidenten Kanten lösen

$\hookrightarrow O(k(n-1)) = O(kn)$
insgesamt

$O(2^k kn)$

binärer Verifikation
Baum,

pro Ebene 2^{i-1} Einfüge

NP-Vollständige Probleme lösen:

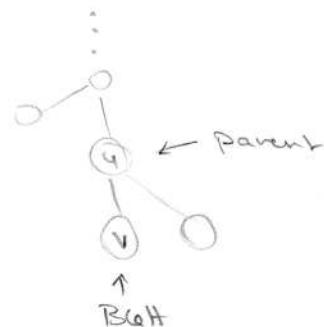
Spezialfall: Ein gegebener Graph ist ein Baum (acyklisch)

INDEPENDENT SET

Definition:

Blatt $v \Rightarrow \deg(v) = 1$

Ein Baum mit 2 Knoten hat 2 Blätter.



Algorithmus

(Geht mit Post-Order durch alle Knoten $O(n) \leftarrow \text{LRN}$, noch effizienter)

Independent-Set-In-A-Forest(T)

$S \leftarrow \emptyset$

while T hat noch 1 Kante

$u \leftarrow$ beliebiges Blatt

$u \leftarrow v \cdot \text{parent}$

$S \leftarrow S \cup \{u\}$

lösche u, v sowie alle incidenten Kanten aus G

$S \leftarrow S \cup v$

return S

Singleton Knoten mit $\deg 0$

Es gibt ein Independent Set im Baum mit max # von Knoten

Case 1: $u \notin \text{MaxIS}$

$v \notin \text{MaxIS}$

Dann ist es aber nicht maximal

Case 2: $u \in \text{MaxIS}$

$v \notin \text{MaxIS}$

Dann kann v den Knoten u ersetzen ohne dass Set kleiner wird

Case 3: $u \notin \text{MaxIS}$

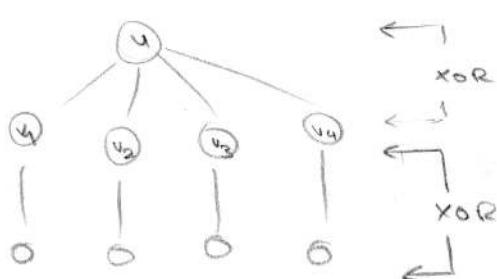
$v \in \text{MaxIS}$

fertig

Conclusion:

Es gibt ein MaxIS mit allen Blättern

Wenn Baum gewichtet ist: Independent Set mit $\max \sum_{v \in S} w_v$



Algorithmus muss entscheiden:

Algorithmus mit dynamischer Programmierung

$$OPT_{in}(u) = w_u + \sum OPT_{out}(v)$$

\uparrow
Kinder

$$OPT_{out}(u) = \sum_{\substack{\text{Alle} \\ \text{Kinder von } u}} \max \{ OPT_{in}(v), OPT_{out}(v) \}$$



Case 1: u nehmen

alle Kinder nicht nehmen

Case 2: u nicht nehmen

für alle Kinder rekursiv entscheiden
ob sie angenommen werden sollten oder nicht
und summieren

$O(u)$

Weighted-Independent-Set-In-A-Tree (T)

Wähle $r \leftarrow \text{root}$ beliebig

foreach Knoten u von T in Postorder (LRN)

if u ist Blatt

$$M_{in}[u] = w_u$$

$$M_{out}[u] = 0$$

else

$$M_{in}[u] \leftarrow w_u + \sum_{\text{Nachfolger } v} M_{out}(v)$$

$$M_{out}[u] \leftarrow \sum_{\text{Nachfolger } v} \max \{ M_{in}[v], M_{out}[v] \}$$

return $\max \{ M_{in}[r], M_{out}[r] \}$

Spezialfall: Graph lässt sich als "Intervallgraphen" darstellen

3-COLOR

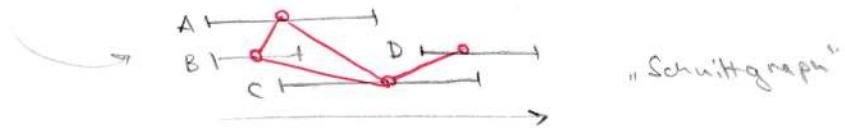
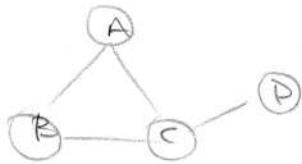
Kann von Graphen-Knoten mit 3 verschiedenen Farben färben,
sodass sich nie 2 gleiche Farben „berühren“ (adjacent sind)

Intervallmenge

Ein Intervall: $a, b \in \mathbb{R} : [a; b] \subseteq \mathbb{R}$

Intervallmenge $I = \{I_v \subset \mathbb{R} | v \in V\}$

Wenn $\exists (u, v) \in E$ dann $\Leftrightarrow I_u \cap I_v \neq \emptyset$ gibt es gemeinsame
Unterintervalle (Überschneidungen)



Wichtig:

1-Color-Problem = {
Independent Set
Interval Scheduling (Greedy-earliest deadline first)}

Definition:

Tiefe $d := \max_{x \in \mathbb{R}} \{|\{I \in I | x \in I\}| \} \equiv$ minimale Anzahl an
Intervallen die übereinander
liegen müssen.

= # Farben die benötigt
wenden

Algorithmus $O(n \log n)$

Graph umwandeln in Intervallgraph und jedem Knoten a und b zuordnen $[a; b]$
und in Liste L speichern

Intervallgrenzen in Liste aufsteigend nach a sortieren

$\text{colmax} = 0$ (max # Farben benötigt)

Queue $Q = \emptyset$

foreach $a \in L$

if a ist Startpunkt

if $Q = \emptyset$

färbe 1 mit colmax

$\text{colmax}++$

if $Q \neq \emptyset$

färbe 1 mit $Q.pop$

if $a \neq$ Startpunkt

$Q.add$ (Farbe von Intervall von 1)

return Färbung der Intervalle und colmax

Sortierung $O(n \log n)$

Färbung $O(n)$

NP-Vollständigkeit meistern (Wiederholung)

NP-C Probleme sind
die schwierigsten NP-Probleme
in die sich alle NP-Probleme
übersetzen lassen

Frage: Angenommen, wir müssen ein NP-vollständiges Problem lösen. Wie sollen wir vorgehen?

Antwort: Die Theorie besagt, dass es unwahrscheinlich ist, einen polynomiellen Algorithmus zu finden.

Man muss eine der gewünschten Eigenschaften opfern:

- Löse Problem optimal
 - Approximationsalgorithmen, Heuristische Algorithmen → „Mit Hausestand“
- Löse Problem in Polynomialzeit
 - Algorithmen mit exponentieller Laufzeit
- Löse beliebige Instanzen des Problems
 - Identifizierte effizient lösbare Spezialfälle

Optimierung – Branch-and-Bound

Algorithmen und Datenstrukturen

VU 186.866, 5.5h, 8 ECTS, SS 2019

Letzte Änderung: 22. Mai 2019

Quiz

Vorlesungsfolien



Rucksack-Problem: Brute-Force / Enumerationsalgorithmus $\mathcal{O}(2^n)$

Aufruf $(0, 0, 0, \vec{x})$

Enum $(z, w_{curr}, q_{curr}, \vec{x})$

if $(q_{curr} \leq G)$

if $(w_{curr} > w_{max})$
 $w_{max} = w_{curr}$
 $\vec{x}_{max} = \vec{x}$

update best solution

for $(i=z+1; i \leq n; i++)$

$x_i = 1$

Enum $(i, w_{curr} + w_i, q_{curr} + q_i, \vec{x})$

$x_i = 1$ Rekursion

$x_i = 0$

$x_i = 0$ Iteration in Schleife

Rucksack-Problem: Verbesserung durch Boundaries

Sortiere Gegenstände nach $\frac{w_i}{q_i}$

Branche nur wenn $w_{max} < U' = w_{curr} + (G - q_{curr}) \cdot \frac{w_i}{q_i}$

Enum $(z, w_{curr}, q_{curr}, \vec{x})$

if $(q_{curr} \leq G)$

if $(w_{curr} > w_{max})$

$w_{max} = w_{curr}$

$\vec{x}_{max} = \vec{x}$

w_{curr} = Eintrag dieser Node
im Baum

w_{max} = größter bisheriger Eintrag

U' = best case Ergebnis
von diesem Node aus

for $(i=z+1; i \leq n; i++)$

$U' = w_{curr} + (G - q_{curr}) \cdot \frac{w_i}{q_i}$

if $w_{max} < U'$

Möglicher Abbruch $w_{max} = U'$

$x_i = 1$

Enum $(i, w_{curr} + w_i, q_{curr} + q_i, \vec{x})$

$x_i = 0$

Verbesserung der unteren Schranke

$L' \neq w_{curr}$ sondern Greedy

mit Greedy das Objekt mit
dem besten $\frac{w_i}{g_i}$ zuerst einpecken
für alle Variablen in \vec{x} die noch
nicht festgelegt sind

$$(1, 0, 1, 0, 1, 0)$$

\underbrace{L}_{Greedy} bestimmen

Verbesserung der oberen Schranke U'

Optimale best-case Lösung ist erreicht wenn sie mit der neu definierten
 $L' = w_{curr}$ übereinstimmt: $L' = U'$

Rucksackproblem: Brute-Force / Enumerationsalgorithmus $O(2^n)$

Aufruf $(0, 0, 0, \vec{x})$ in $\text{Enum}()$

$\text{Enum}(z, w_{\text{curr}}, g_{\text{curr}}, \vec{x})$

if ($g_{\text{curr}} \leq G$)

if ($w_{\text{curr}} > w_{\text{max}}$)
 $w_{\text{max}} = w_{\text{curr}}$
 $\vec{x}_{\text{best}} = \vec{x}$

update best
solution

for ($i = z+1; i \leq n; i++$)

$x_i \in \vec{x} = 1$] $x_i = 1$, Rekursion

$\text{Enum}(i, w_{\text{curr}} + w_i, g_{\text{curr}} + g_i, \vec{x})$

$x_i \in \vec{x} = 0$] $x_i = 0$, Iteration

z : # beteiligten Stoffen

w_{curr} : $\sum w$

g_{curr} : $\sum g$

\vec{x} : binäres Lösungsvektor

Globale Variablen:
 w_{max} und \vec{x}_{best}

Ablauf:

for-Schleife bestimmt Position



$(\cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$ $z=0 \rightarrow i = z+1$
 $y_0 \ y_1 \ y_2 \ y_3 \ y_4 \ y_5$

↓
Entweder abziehen
 $y_0 = 0$

oder mitnehmen
 $y_0 = 1$

nächste Schleifen-Iter.

nächste Rekursionsebene

$(0, \cdot, \cdot, \cdot, \cdot, \cdot)$

$(1, \cdot, \cdot, \cdot, \cdot, \cdot)$

Verbesserung durch boundaries für branches

Man überprüft ob tiefere Rekursion sinnvoll ist

Obere Schranke U' :

$$U' = w_{\text{curr}} + (G - g_{\text{curr}}) \cdot \frac{w_i}{g_i} \quad \frac{\text{Leistung}}{\text{Preis}}$$

↓
 aktuelle Summe # freie Stellen Preis-Leistung
 für nächsten Gegenstand

Einsatz in Enum:

for ($i = z+i$; $z \leq n$; $z++$)

$$U' = w_{\text{curr}} + (G - g_{\text{curr}}) \cdot \frac{w_i}{g_i}$$

if ($U' > w_{\text{max}}$)

$$x_i = 1$$

Enum ($i, w_{\text{curr}} + w_i, g_{\text{curr}} + g_i, \rightarrow$)

$$x_i = 0$$

↓
 Weil absteigend sortiert können folgende Elemente nur kleiner sein

Nur wenn Verbesserungspotential besteht!

Beispiel

$$w_{\text{max}} = 11$$

$$w_{\text{curr}} = 6$$

$$(G - g_{\text{curr}}) = 2 \quad (\text{Freie Stellen})$$

Der nächste Gegenstand zum einfügen \Rightarrow $w_i = 8$
 $g_i = 4$

Aber weil $4 > 2$ passt es nicht in
 unserem Rucksack

Deshalb:

$$\frac{w_i}{g_i} = \frac{8}{4} = \frac{2}{1} \quad \rightarrow$$

Wir stellen uns vor
 $w_i = 2$
 $g_i = 1$

womit obere Schranke
 $6 + 2 \cdot 2 = 10$

und

$$U' = 10 < w_{\text{max}} = 11$$

Breath-and-Bound verallgemeinert für Maximierungsprobleme

Boundaries

Lokale Boundaries werden für jede Node einzeln berechnet

local lower bound L' = Worst case von dieser Node aus (Greedy / Weise)

local upper bound U' = best case von dieser Node aus
(mit „Dualheuristik“ berechnet)

Globale Boundaries halten Ergebnisse von allen Nodes im Baum fest

global lower bound L = worst case bisher ($= w_{\max}$)

Instanz
↓

Breath-and-Bound-Max (I)

$L \leftarrow -\infty$ oder „initiale Lösung“ mit Greedy

$\Pi \leftarrow \{1\}$

while $\exists I' \in \Pi$

Entferne I' aus Π

Berechne U' mit „Dualheuristik“

if $U' > L$

Berechne L' heuristisch

if $L' > L$

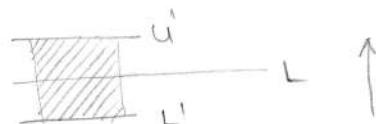
$L = L'$

if $U' > L$

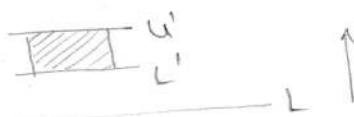
Partitioniere I' und speichere
in Π

return beste Lösung mit Wert L

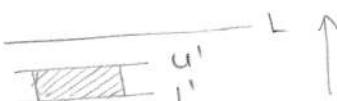
- Es besteht noch verbesserte Potential



Es muss L aktualisiert werden



- Es besteht kein Verbesserungspot.



Branch-and-Bound verallgemeinert für Minimierungsproblem

Boundaries

- local upper boundary U' = worst case von dieser Node aus (Greedy / w_{curr})
- local lower boundary L' = best case von dieser Node aus ("Dualheuristik")
- global upper boundary U = worst case bisher ($= w_{min}$)

Branch-and-Bound-Min()

$U = \infty$ oder initiale heuristische (Greedy) Lsg

$\Pi \leftarrow \{1\}$

while $\exists l' \in \Pi$

entferne l' aus Π

Berechne L' mit "Dualheuristik"

if $L' < U$

Berechne U'

if U' ist eine gültige Lösung

if $U' < U$

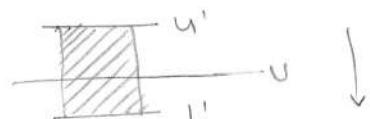
$U = U'$

if $L' < U$

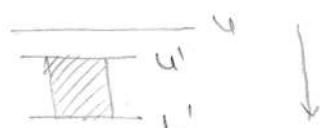
Partitioniere / Partitione

return beste gefundene Lösung mit Wert U

Verbesserung möglich: $L' < U$



update von unten



Verbesserung unmöglich



Auswahlstrategien

Welche Node aus dem Baum sollte als Nächster bearbeitet werden?

Best first

Node mit besten „Dualheuristik“
(best case - Schnelligkeit)

- intuitiver
- schneller aber ungenau

Depth first

Nodes werden einfach wie bei DFS gewählt

- langsam
- bessere Approximation, gegeben
- vollständige Lösung

Brech and Bound : Minimales Vertex Cover

Lösungsweg $C = \emptyset$

Min Vertex Cover (G, C)

$|U| \leq |V| - 1$ (triviale Schranke, man braucht höchstens $n-1$ Knoten)

$\Sigma = \{(G, C)\}$

while $\exists I' \in \Sigma$

Berechne L' mit „Matching Heuristik“

if $L' < U$

Berechne U' mit „Greedy Heuristik“

if $U' < U$

$U = U'$

if $L' < U$

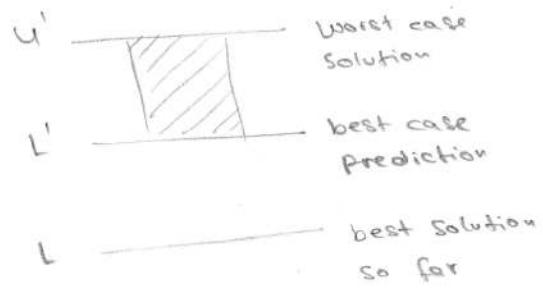
(Start Branching)

$u_{\max} = \text{Knoten mit max deg}$

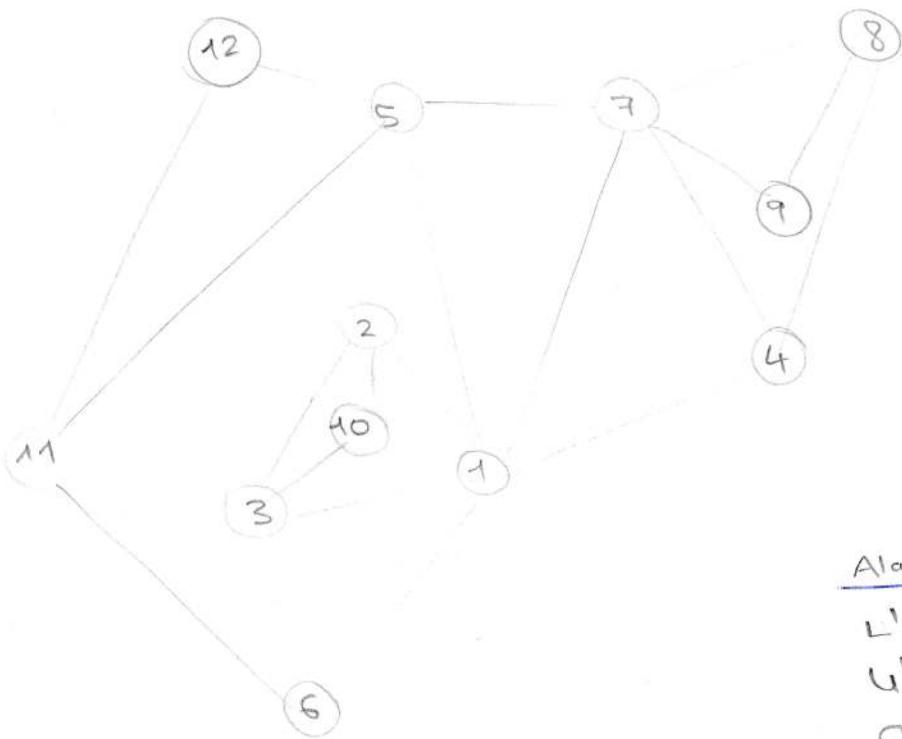
$I'_1 = (G' \setminus \{u_{\max}\}, C \cup \{u_{\max}\})$

$I'_2 = (G' \setminus \{u_{\max}\}, N(u_{\max}), C \cup N(u_{\max}))$

return Lösung mit Wert U



Aufgabe 3) Minimales Vertex-Cover



Algorithmus:

L' : best case: Greedy

U' : worst case: Matching

C : Lösungsmenge

Knoten sortiert nach Greedy:

1	[6]
7	[5]
5	[4]
2, 3, 4, 8, 11	[3]
6, 9, 10, 12	[2]



Knoten	Knotengrade	
1	6	✓
2	3	✓
3	3	✓
4	3	✓
5	4	✓
6	2	✓
7	5	✓
8	3	✓
9	2	✓
10	2	✓
11	3	✓
12	2	✓

Knotenverzweigung

Min Vertex Cover - BranchAndBound (G, c)

$U \leftarrow |V|-1$ //triviale initiale obere Schranke

$M \leftarrow \{G, c\}$ //weil $n-1$ Knoten alle Kanten abdecken

while $\exists I' \in M, I' = (G', c')$

$M \leftarrow M / \{I'\}$

$L' \leftarrow$ lokale untere Schranke mit Matching-Heuristik "best case"

if ($L' < U$)

$U' \leftarrow$ lokale obere Schranke mit Greedy-Heuristik "worst case"

if ($U' < U$)

$U \leftarrow U'$

if ($L' < U$)

$u_{max} \leftarrow$ Knoten mit $\max \deg(v)$

$I''_1 \leftarrow (G' \setminus \{u_{max}\}) \cup \{u_{max}\}$

$I''_2 \leftarrow (G' \setminus \{u_{max}\} \setminus N(u_{max}), C \cup N(u_{max}))$

$M \leftarrow M \cup \{I''_1, I''_2\}$

return Lösung mit west. U .

Branching:

I''_1 aufnehmen in Lösungsmenge

I''_2 ablehnen aus Lösung,
aber Nachbarn
aufnehmen

Wenn best-case für Teilproblem
"kleiner" ist als beste bisher gefundene
Lösung, dann Baum abschneiden

G

$$C = \{\} \quad |C| = 0$$

$L' = 6$ (Max Matching)

$$U' = | \{11, 5, 7, 8, 2, 10, 1\} | + 0 = 7 \text{ (Greedy)}$$

$$U = |V| - 1 = 11$$

$$7 < 11$$

$$U = 7$$

$$C = C \cup \{1\}$$

$$C = C \cup \{N(1)\}$$

$$G = G \setminus \{1\}$$

$$C = \{1\} \quad |C| = 1$$

$$L' = 5 + 1 = 6$$

$$U' = | \{11, 5, 7, 8, 3, 10\} | + 1 = 7$$

$$G = G \setminus \{1, 2, 3, 4, 5, 6, 7\}$$

$$C = \{2, 3, 4, 5, 6, 7\} \quad |C| = 6$$

$$L' = 1 + 6 = 7$$

$$U' = | \{9, 12\} | + 6 = 8$$

$$C = C \cup \{7\}$$

$$C = C \cup \{N(7)\}$$

$L' = U$
ABBRUCH

$$G = G \setminus \{1, 7\}$$

$$C = \{1, 7\} \quad |C| = 2$$

$$L' = 4 + 2 = 6$$

$$U' = | \{11, 10, 8, 3, 5\} | + 2 = 7$$

$$G = G \setminus \{1, 7, 5, 4, 8, 9\}$$

$$C = \{1, 5, 4, 8, 9\} \quad |C| = 5$$

$$L' = 2 + 5 = 7$$

$$U' = | \{10, 3, 11\} | + 5 = 8$$

$$C = C \cup \{11\}$$

$$C = C \cup \{N(11)\}$$

$L' = U$
ABBRUCH

$$G = G \setminus \{1, 7, 11\}$$

$$C = \{1, 7, 11\} \quad |C| = 3$$

$$L' = 3 + 3 = 6$$

$$U' = | \{8, 12, 2, 10\} | + 3 = 7$$

$$G = G \setminus \{1, 7, 11, 12, 5, 6\}$$

$$C = \{1, 7, 12, 5, 6\} \quad |C| = 5$$

$$L' = 2 + 5 = 7$$

$$U' = | \{8, 2, 10\} | + 5 = 8$$

$L' = 4$
ABBRUCH

$$C = C \cup \{8\}$$

1

$$G = G \setminus \{1, 7, 11, 8\}$$

$$C = \{1, 7, 11, 8\} \quad |C| = 4$$

$$L' = 2 + 4 = 6$$

$$U' = | \{12, 10, 2\} | + 4 = 7$$

$$C = C \cup \{N(8)\}$$

$$G = G \setminus \{1, 7, 11, 8, 9, 4\}$$

$$C = \{1, 7, 11, 9, 4\} \quad |C| = 5$$

$$L' = 2 + 5 = 7$$

$$U' = | \{12, 10, 2\} | + 5 = 8$$

$$C = C \cup \{10\}$$

1

$$G = G \setminus \{1, 7, 11, 8, 10\}$$

$$C = \{1, 7, 11, 8, 10\} \quad |C| = 5$$

$$L' = 2 + 5 = 7$$

$$U' = | \{12, 2\} | + 5 = 7$$

$$C = C \cup \{N(10)\}$$

$L' = U$
ABBRUCH

$$G = G \setminus \{1, 7, 11, 8, 10, 2, 3\}$$

$$C = \{1, 7, 11, 8, 9, 4\} \quad |C| = 6$$

$$L' = 1 + 6 = 7$$

$$U' = | \{12\} | + 6 = 7$$

$L' = U$ und $L' = U'$

ABBRUCH

$L' = U$ und $L' = U'$

ABBRUCH

Alle Lösungsmengen mit Kardinalität von 7:

$$\{11, 5, 7, 8, 2, 10, 1\}$$

$$\{11, 5, 7, 8, 3, 10, 1\}$$

$$\{11, 5, 7, 8, 3, 10, 1\}$$

$$\{11, 5, 7, 8, 3, 10, 1\}$$

$$\{11, 12, 7, 1, 8, 12, 10\}$$

$$\{1, 7, 11, 8, 12, 10, 2\}$$

$$\{1, 7, 11, 8, 10, 12, 2\}$$

$$\{1, 7, 11, 8, 10, 12, 2\}$$

$$\{1, 7, 8, 9, 4, 11, 12\}$$

Die obere Schranke zuletzt gesetzt
von:

Optimierung – Dynamische Programmierung

Algorithmen und Datenstrukturen

VU 186.866, 5.5h, 8 ECTS, SS 2019

Letzte Änderung: 24. Mai 2019

Vorlesungsfolien

→ Siehe
max weight
[independent set]



1 / 93

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung: Dynamische Programmierung kann dann eingesetzt werden, wenn das Problem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung sich aus optimalen Lösungen der Teilprobleme zusammensetzt.

Approximation(algorithmen)

Heuristische Verfahren

2 / 93

Grundlagen

Dynamische Programmierung: Teile das Problem in eine Folge von überlappenden Teilproblemen auf und erstelle und speichere Lösungen für immer größere Teilprobleme unter Verwendung der abgespeicherten Lösungen.

Zwischenfragen

Optimalitätsprinzip von Bellman: Dynamische Programmierung führt zu einem optimalen Ergebnis genau dann, wenn es sich aus den optimalen Ergebnissen der Subprobleme zusammensetzt.

Effizienz: Hängt von der Vorgehensweise bei der Aufteilung und Ermittlung der Lösungen für die einzelnen Teilprobleme ab.

Wesentlicher Aspekt: Speicherung (memoization) von Ergebnissen für Subprobleme zur Wiederverwendung.

3 / 93

Beispiel: Weighted Independent Set auf Bäumen

Bestimmt mit M_{IN} (Node \checkmark , Nachbarn \times)

M_{OUT} (Node \times , Nachbarn \checkmark)

Kinder:

$$\begin{aligned} 3 & \quad M_{IN}: 3 + 4 + 6 + 0 = 17 \\ & \quad M_{OUT}: 0 + 14 + 6 = 20 \end{aligned}$$

$$5 + M_{OUT} \text{ von Kindern} = M_{IN}: 5 + 0 = 5$$

$$0 + \text{größerer Wert von Kindern} = M_{OUT}: 4 + 4 + 6 = 14$$

$$\begin{aligned} 6 & \quad M_{IN}: 6 + 0 \\ & \quad M_{OUT}: 0 + \text{keine Kinder} \end{aligned}$$

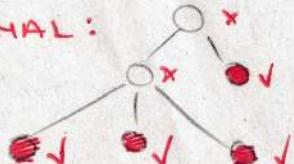
$$4 + M_{OUT} = M_{IN}: 4$$

$$M_{OUT}: 0 + 0 = 0$$

$$6 + M_{OUT} = M_{IN}: 6$$

$$M_{OUT}: 0 + 0 = 0$$

OPTIMAL:



4 / 93

Dynamische Programmierung

Memoization von Lösungen für Teilprobleme

Fibonacci - Rekursiv : „top down“

```

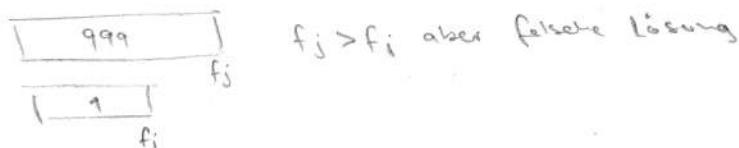
Fibonacci(n)
if F[n] = -1
    if n=1 v n=2
        F[n] = 1
    else
        F[n] = Fibonacci(n-1) + Fibonacci(n-2)
return F[n]

```

Beide mit Laufzeit $O(n)$

Gewichtetes Intervall Scheduling

Greedy scheitert wenn gewichtet:

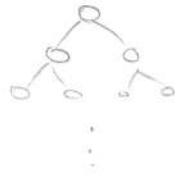


Brute force

$$OPT(j) = \begin{cases} 0 & \text{wenn } j=0 \\ \max\{w_j + OPT(p(j)), OPT(j-1)\} & \text{sonst} \end{cases}$$

$\xrightarrow{\quad \quad \quad \quad \quad}$

OPT_{in} OPT_{out}



Rekursiv „top down“

Sortiere Jobs nach Beendigungszeit $f_j < f_i$ wenn $j > i$

Berechne $p(\forall n)$

$OPT(j)$

if $M[j] = \text{leer}$

$$M[j] = \max\{w_j + OPT(p(j)), OPT(j-1)\}$$

return $M[j]$

Iterativ „bottom-up“

```

Fib(n)
F[1] = 1
F[2] = 1
for i=3 bis n
    F[i] = F[i-1] + F[i-2]
return F[n]

```

Iterativ „bottom-up“

```

M[0] = 0
for i=1 bis n

```

$$M[i] = \max(w_i + M[p(i)], M[i-1])$$

Laufzeit für Top-down

Sortierung $O(n \log n)$

Binäre Suche für Berechnung von Vorausrechner $O(n \log n)$

Aufruf von $\text{OPT}(n)$

Liefert entweder Lösung in $O(1)$

oder berechnet neuen Eintrag durch Verzweigung

$\text{OPT}(p(j))$
 $\text{OPT}(j-1)$

Laufzeit für Bottom-up

$O(n)$

Fortschritts-Maß "y"

Am Anfang 0,

Durch jede Baumebene $y++$



$O(n)$ teil

$0 \leq y \leq n$

Backtracking

Backtracke (j)

if $j=0$

keine Ausgabe

else if $w_j + M[p(j)] > M[j-1]$

$\underbrace{\quad}_{\text{OPT}_{in}}$ $\underbrace{\quad}_{\text{OPT}_{out}}$

gib j aus

Backtracke ($p(j)$)

$O(n)$ insgesamt

else

Backtracke ($j-1$)

Segmented Least Squares

Allgemeines Problem mit 1 Gerade: least squares

Finde bei n Punkten die Gerade $y = \boxed{a}x + \boxed{b}$ mit minimalem Abstand zu allen Punkten

$$\min Err = \sum_{i=1}^n (y_i - (ax_i + b))^2 \text{ minimiert}$$

Wirklich
keit = y (Guess)



(lässt sich mit Formel berechnen)

Mit mehreren Geraden ist $\underbrace{\text{Genauigkeit}}$ und $\underbrace{\text{Sparsamkeit}}$ wichtig

$\min Err$ $\min \# \text{Geraden}$

Wir definieren ein Verhältnis:

$$E = \text{Error}^2$$

$$L = \# \text{Geraden}$$

$$c = \text{Konstante} > 0$$

„Trade off function“

$$\Rightarrow E + cL = \text{konstante}$$

$\uparrow c \Rightarrow$ weniger Geraden

$\downarrow c \Rightarrow$ mehr Geraden,
weniger Fehler

Algorithmus

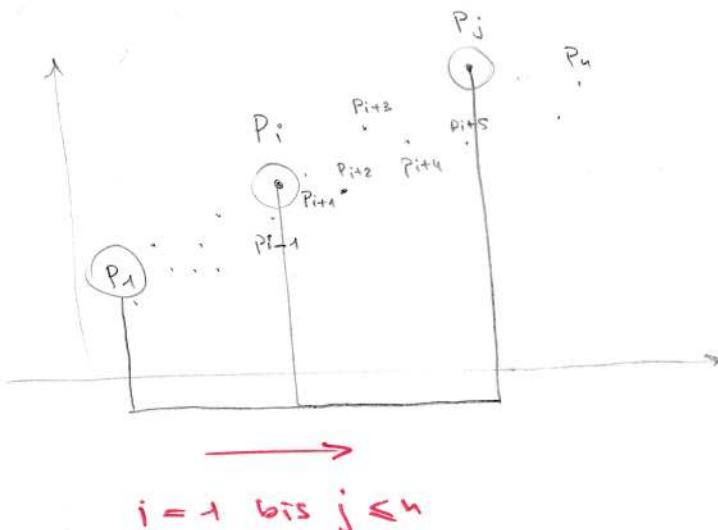
Eingabe: n Punkte $\{p_1, p_2, \dots, p_n\}$

$$\Rightarrow \text{OPT}(j) = \min_{i=1 \text{ bis } j} \left\{ \text{OPT}(i-1) + \min \text{Err}(i; j) + c \right\}$$

wobei $j \leq n$

\downarrow
 $e(i; j)$

löst mit i Geraden



Algorithmus Berechnet:

Minimum der folgenden Gleich.:

$$\text{OPT}(0) + \min \text{Err}(1; j) + c$$

$$\text{OPT}(1) + \min \text{Err}(2; j) + c$$

$$\text{OPT}(j-1) + \min \text{Err}(j; j) + c$$

Algorithmus, iterativ: „bottom-up“

Segmented-Least-Squares ($P = \{p_1 \dots p_n\}$)

$M[0] = 0$

for $j=1$ bis n

for $i=1$ bis n

berechne $e(i,j)$ $O(n)$

$\left. \right] O(n^2)$

(Least Squares $(i,j) = e(i,j)$)

Löse Problem mit einer Linie mit min Err

for $j=1$ bis n

$M[j] = \min_{1 \leq i \leq j} (M[i-1] + e(i,j)) + c$

$\left. \right] O(n^2)$

return $M[n]$

Insgeamt:

Laufzeit $O(n^3)$ (Kann verbessert werden zu n^2)

Rucksackproblem

Greedy: keine optimale Lösung

Parench and Bound: $O(2^n)$

Memoization: $O(nG)$ wobei $G = \text{Rucksackplatz}$

Benötigt mehr Speicher, da Problem für alle $g \leq G$ gelöst wird
für alle Elemente von i bis n

$$OPT(i, g) = \begin{cases} 0 & \text{wenn } i=0 \\ OPT(i-1, g) & \text{wenn } g_i > g \quad \text{Kapazität übersch.} \\ \max \{ OPT(i-1, g), w_i + OPT(i-1, g - g_i) \} & \text{sonst} \end{cases}$$

OPT_{out} OPT_{in}

Algorithmus iterativ - „bottom up“

Rucksack: $(n+1) \times (G+1)$ Array (eine zusätzliche Stelle für index 0)

for $g=0$ bis G
 $M[0, g] = 0$

for $i=1$ bis n
 for $g=0$ bis G
 if $g_i > g$
 $M[i, g] = M[i-1, g]$

else $M[i, g] = \max \{ M[i-1, g], w_i + M[i-1, g - g_i] \}$

return $M[n, G]$

$\boxed{O(nG)}$

Laufzeit & Speicher: $O(nG)$

Backtracking (M -Array)

$A = \emptyset$
 while $n > 0$ and $G > 0$
 $OPT_{\text{in}} \leftarrow$ if $M[i, G] \neq M[i-1, G]$
 $A = A \cup \{i\}$
 $G = G - g_i$
 $i = i-1$
 return A

Problem: $O(uG)$ \rightarrow „Pseudo Polynomell“

Polynomell in n

G ist exponentiell in Länge weil Zahlen binär kodiert werden

Es gilt:

Rucksackproblem kann nicht im P gelöst werden bei $P \neq NP$

Verbesserung des Algorithmus:

Array nicht füllen wenn $OPT(i, g) = OPT(i-1, g)$

Kürzeste Pfade mit negativen Kanten gewichten, ohne negativen Kreisen

Definition Pfad

Weg / (Menge an Knoten) in der sich kein Knoten wiederholt

"Finden eines kürzesten Pfades mit negativen Kanten" ist NP-C (Dijkstre nicht anwendbar)

Problem:

Wenn es negative Kanten gewichte gibt, gilt es auch negative Kreise

Dadurch werden Algorithmen "dazu verleitet" sich im Kreis zu drehen und Knoten doppelt zu besuchen

Bellman's Gleichungen

Wenn keine negativen Kreise im Graph



OPT(w) = Länge des kürzesten Pfades von w nach t

Es gilt:

$$OPT(t) = 0$$

$$OPT(v) = \min_{(v,w) \in E} \{ c_{vw} + OPT(w) \} \text{ für } \forall v \in V \setminus \{t\}$$

Implementierung:

OPT(i,v) ist Länge des kürzesten v-t Pfades mit höchstens i Kanten

Entweder: \exists direkter Weg vom Knoten
mit weniger Kanten

$$OPT(i,v) = OPT(i-1,v)$$

Oder \exists indirekter Weg

$$OPT(i,v) = OPT(i-1,w) + c_{vw}$$

$$OPT(i,v) = \begin{cases} 0 & \text{wenn } i=0 \text{ und } v=t \\ \infty & \text{wenn } i=0 \text{ und } v \neq t \\ \min \left\{ OPT(i-1,v), \min_{(v,w) \in E} \{ c_{vw} + OPT(i-1,w) \} \right\} & \text{sonst} \end{cases}$$

Wichtig:

$\text{OPT}(u-1, v) = \text{OPT}(u)$ wenn es keine Kreise gibt, da worst case: 0000...0

u Knoten
v-1 Knoten

Iterative Bottom-up Implementierung

Shortest-Path (G, c, t)

$M [Kanten \#; Knoten]$

foreach $v \in V$

$$M[0, v] = \infty$$

$$M[0, t] = 0$$

for $i = 1$ bis $n-1$

foreach $v \in V$

$$M[i, v] = M[i-1, v]$$

} Wert von $i-1$ übernehmen für i

foreach $(v, w) \in E$

$$\rightarrow M[i, v] = \min(M[i, v], c_{vw} + M[i-1, w])$$

return $M[n-1, t]$

Laufzeit:

Initialisierung: $O(|V|) = O(n)$

For Schleife: $O(n-1)$

foreach : $O(|V|) = O(n)$

foreach : $O(\deg(v)) \rightarrow$ insgesamt,
unabhängig von
for-Schleife in
 $O(2n)$ wegen
Handshake-Lemma

Speicherplatz:

$n \times n$ -Array

Laufzeit:

$$O(n \cdot (n-1) \cdot n + 2n) = O(n + n^2 + 2n) = O(n^2 + n)$$

Durch effiziente Implementierung $O(nm)$ auch möglich

Negative Kreise erkennen

Durch Bellman-Ford-Algorithmus

Wenn $\text{OPT}(u, v) < \text{OPT}(u-1, v)$ dann wurde Pfad mit mehr Kanten kürzer
also \exists negativer Kreis im Pfad zu +

In $O(nm)$ wegen Bellman-Ford-Algorithmus

Effiziente Implementierung des Bellman-Ford-Algorithmus:

Push-Based-Shortest-Path(G, s, t):

foreach $v \in V$

$M[v] = \infty$

$\text{Next}[v] = \emptyset$

→ um Backtracking zu ermöglichen

$M[t] = 0$

for $i=1$ bis $n-1$

foreach $(v, w) \in E$

if $M[v] > M[w] + c_{vw}$

$M[v] = M[w] + c_{vw}$

$\text{Next}[v] = w$

$O(n \cdot 2m) =$

$O(n \cdot m)$

if $M[w]$ ändert sich in dieser Iteration :

break

return $M[s]$

Backtracking:

Wenn $\text{Next}[v] = w$ dann gilt: $M[v] \geq c_{vw} + M[w]$

Optimierung - Approximation

Algorithmen und Datenstrukturen

VU 186.866, 5.5h, 8 ECTS, SS 2019

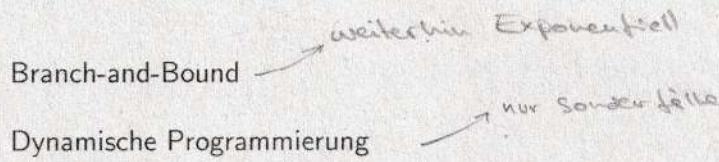
Letzte Änderung: 28. Mai 2019

Vorlesungsfolien



1 / 58

Optimierung: Roadmap



Approximation(salgorithmen): Erzeuge in polynomieller Zeit eine Näherungslösung, die eine Gütegarantie besitzt. Die Güte eines Algorithmus sagt etwas über die Fähigkeit aus, optimale Lösungen gut oder schlecht anzunähern.

Heuristische Verfahren

Nicht bei
heuristischen
Verfahren

2 / 58

Approximationsalgorithmen

Annäherung mit Gütegarantie in Polynomzeit

$$\frac{C_A(x)}{C_{opt}(x)} \geq \varepsilon$$

Minimierung $\varepsilon \geq 1$
Maximierung $0 \leq \varepsilon \leq 1$

VERTEX COVER

MINIMALES-VERTEX-COVER ist NP-Schwer

2-Approx-Vertex-Cover(G)

$$C = \emptyset$$

while $E \neq \emptyset$

wähle zufällig Kante $(u, v) \in E$

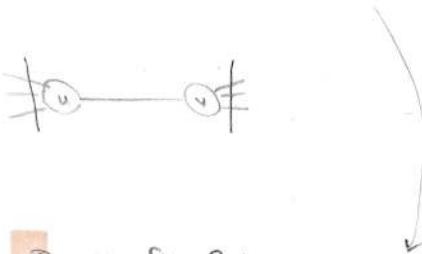
$$C = C \cup \{u, v\}$$

entferne alle incidenten Kanten von u und v

return C

$O(n+m) \rightarrow$ ist ein Matching

$$\varepsilon = 2$$



Beweis für ε :

AVC-Algorithmus bildet Matching und bildet obere Schranke mit $C_{opt}(x) \leq 2 \cdot C_{AVC}(x)$

Bei vollständigen Graphen mit gerader Knotenanzahl:

$$C_{opt}(x) = 2 \cdot C_{AVC}(x)$$

Greedy-Approx-Vertex-Cover(G)

$$C = \emptyset$$

while $E \neq \emptyset$

wähle Knoten mit max deg

$$C = C \cup \text{Node}$$

Entferne alle incidenten Kanten zu Node

return C

$$\varepsilon = \log n$$

Beweis kompliziert

MST-Heuristik für das symmetrische Traveling Salesman Problem

$c_{ij} = c_{ji}$

Wert von beiden Eltern gleich

Der Graph ist:

- ungerichtet
- vollständig, alle Knoten miteinander verbunden
- schlicht, keine Schlingen
- gewichtet aber nicht negativ

Gesucht:

Tour mit allen Knoten mit minimaler Gewichts-Summe

MST-Heuristik

1. Bestimme MST(V, E)
2. Verdopple alle Kanten und tolle Graph (V, E')
3. Bestimme Euler-Tour E in (V, E')
(Das bedeutet jede Kante nur \times)
4. Gebe Euler-Tour eine Orientierung

Wähle Startpunkt $s \in V$

marked [s] = true

$p = s$

p = letzter besuchter Knoten

5. Wenn bereits alle Knoten markiert

$$T = T \cup \{(p, s)\}$$

retourniere T

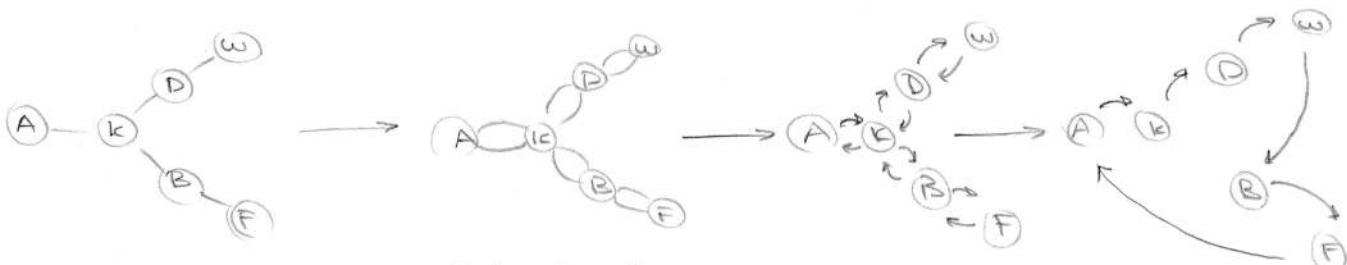
6. sonst traversiere von p entlang der Tour bis q , wobei $\text{marked}[q] = \text{false}$

$$T = T \cup \{(p, q)\}$$

marked [q] = true

$p = q$

→ Wiederhole ab Schritt ⑤



Euler-Tour kann gefunden werden da alle $\deg(v)$ sind gerade

Laufzeit $O(n^2)$

Aufwändiger Teil: MST-Bestimmung mit Prim in $O(n^2)$

Bestimmung der Euler-Tour in $O(n+m)$

Bestimmung einer möglichen Euler-Tour:

Algorithmus von Hierholzer

1. Wähle beliebigen $v_0 \in V$

konstruiere geschlossenen Pfad von v_0 namens Z , in der keine Kante doppelt vorkommt

2. Wenn Z alle Knoten beinhaltet:

terminiere

3. Sonst lösche alle Kanten E_Z aus E

4. Wähle beliebigen $v_2 \in Z$ mit $\deg(v_2) > 0$
bilde neuen Zyklus Z'

5. Füge Z in Z' ein

Güte-Garantie für:

MST-Heuristik für symmetrisches TSP

CASE 1: TSP ist nicht metrisch

Wenn $P \neq NP$ dann \exists Gütegarantie $\epsilon \leq 2$ weil

HAM-CYCLE \leq_p SYMMETRICAL-TSP

Transformation:
 $G(V, E) \rightarrow G(V, E^1, c)$

Gewichtung für alle Kanten:

$$c_{u,v} = \begin{cases} 1 & \text{wenn } (u,v) \in E \text{ von Hamilton} \\ 2n+1 & \text{sonst einfach beliebig hohe Kosten} \end{cases}$$

Weil HAM-CYCLE nicht unbedingt ein vollständiger Graph sein muss.

Daraus folgt, dass:

Ja-Instanz \rightarrow Kreis mit $n=|V|$ Kosten

Nein-Instanz \rightarrow Kreis mit mindestens

$$(2n+1) + (n-1) = 3n > 2n$$

falsche
Kante

Conclusion:

- Für beliebiges $\epsilon > 1$ kann es keinen polynomiellen ϵ -Approximationsalgorithmus geben, da TSP-SYM NP-HARD ist

$$c(u,v) = \epsilon \cdot u + 1 \quad \text{wenn } (u,v) \notin E$$

CASE 2: TSP ist metrisch // "Euklidisches TSP"

Dreiecksungleichung gilt:

$$c_{ik} \leq c_{ij} + c_{jk} \quad \text{für Knoten } i, j, k \in V$$

Es gibt einen Approximationsalgorithmus mit $\epsilon=2$ (in polynomieller Zeit) $O(n^2)$

Beweis:

c_{ST} MST - Heuristik wobei doppelte Kanten nach Euler-Tour entfernt wurden

c_{VI} MST - Heuristik, verdoppelte Kanten

c_{MST} Minimales Spannbaum

c_{OPT} Optimale Lösung

$$c_{ST} \leq c_{VI} = 2 \cdot c_{MST} \leq 2 c_{OPT}$$

wegen Weil MST kein Zyklus ist muss es kleiner sein
Dreiecks-Ungleichung

Lastverteilung - „Load Balancing“

Scheduling Problem mit mehreren Maschinen zugleichen

- m Maschinen
- n Jobs $\{j_1 \dots j_n\}$ werden mit j_i der Maschine $1 \leq i \leq m$ zugewiesen
- t_{ij} Bearbeitungszeit
- L_i Last von Maschine $i = \sum_{j \in J_i} t_{ij}$ Summe der Arbeitszeit
- L „Makespan“ = $\max_i L_i$

Ziel: „Makespan“ bzw. Bearbeitungsdauer minimieren

Algorithmus

Ordne n Jobs absteigend

List-Scheduling $(m, n, t_1, t_2, \dots, t_n)$

for $i=1$ bis m

$L_i = 0$ (Last auf Maschine i)

$J_i = \emptyset$ (Jobs von Maschine i)

for $j=1$ bis n

$i = \arg\min_{k=1, \dots, m} L_k$

$O(\log m)$
mit Heap

Laufzeit

$O(n \log m)$

$J_i = J_i \cup \{j\}$

$L_i = J_1 \dots J_m$

Über Algorithmus:

- Greedy

- E = 2

Beweis:

Angenommen L ist makespan von Algo

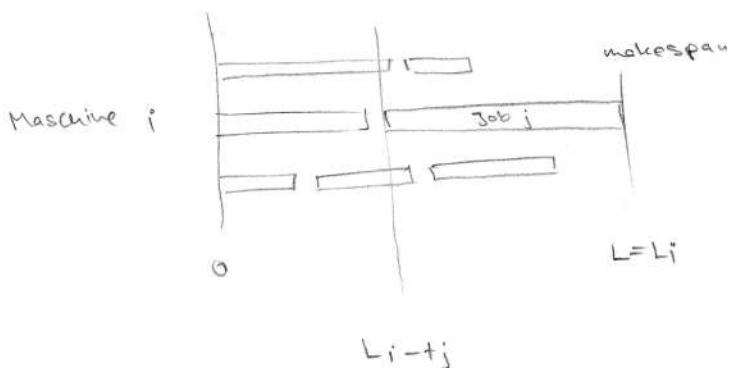
L^* ist optimale Lösung

Lemma: $L^* \geq \max_j t_j$ (längster Job)

Lemma: $L^* \geq \frac{1}{m} \sum_j t_j$ (Durchschnitt)

Lemma: $E=2$ für Algorithmus

Beweis: Angenommen Maschine i definiert Makespan und ist Flaschenhals
Bevor der Maschine i der letzte Job j zugewiesen wurde hatte
Maschine i die geringste Last unter allen



Die Last $Li - t_j$ war die kleinste unter allen Maschinen

$$Li - t_j \leq L_k \text{ für alle } k \in [1; m]$$

Bzw.

$$Li - t_j \leq L_1 \quad \left. \begin{array}{l} \text{Last von Maschine 1} \\ \text{(alle vor Maschine i)} \end{array} \right\}$$

$$Li - t_j \leq L_2 \quad \left. \begin{array}{l} \\ \end{array} \right\}$$

:

$$Li - t_j \leq L_i$$

$$Li - t_j \leq L_{i+1} \quad \left. \begin{array}{l} \\ \text{(alle nach Maschine i)} \end{array} \right\}$$

$$Li - t_j \leq L_m$$

$$m \circ (Li - t_j) \leq \sum_{k=1}^m L_k$$

$$Li - t_j \leq \frac{1}{m} \sum_{k=1}^m L_k \quad \text{Summe aller Gesamtlasten durch } m$$

$$= \frac{1}{m} \sum_{j=1}^n t_j \quad \text{Summe aller Jobs-Zeiten durch } m$$

$$\leq L^*$$

Daraus folgt:

$$Li - t_j \leq \frac{1}{m} \sum_{j=1}^n t_j \leq L^* \quad \rightarrow \quad Li = \underbrace{(Li - t_j)}_{\leq L^*} + \underbrace{\max t_j}_{\leq L^*} \leq 2L^*$$

$$\max t_j \leq L^*$$

Center Selection

Eingabe n Standorte $S = \{s_1, \dots, s_n\}$
 $k > 0$

Gesucht k Mittelpunkte $C = \{c_1, \dots, c_k\}$

So dass $r(C)$ minimiert wird

↓

Definiert durch $\text{dist}(x, y)$ in euklidischer Ebene:

$\text{dist}(x, y)$

$$\underline{\text{dist}}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c)$$

$$r(C) = \max \text{dist}(s, C)$$

Greedy Algorithmus

Greedy-Center-Selection (k, S)

$C = \text{beliebiges } s_i \in S$

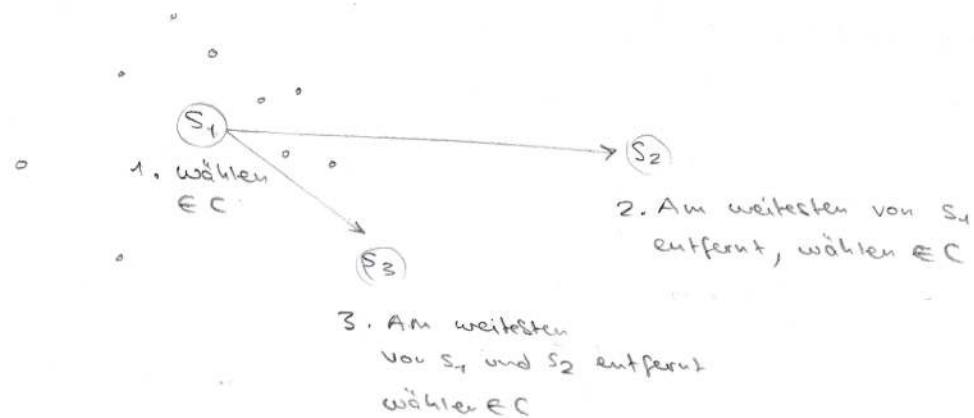
Wiederhole $k-1$ Mal:

Wähle s_i mit $\max \text{dist}(s_i, C)$

$$C = C \cup \{s_i\}$$

return C

} wähle s_i mit größtem Abstand zu allen C



Im Worst Case verbessert sich $r(C)$ nicht nach der ersten Iteration und alle Standorte haben exakt gleiche Distanz

Aber es gilt:

Abstand zwischen 2 $c \in C$ ist nach Algorithmus immer $\geq r(C)$

↑

letzte Iteration

Analyse des Greedy Algorithmus

$$C_{\text{Greedy}} = C$$

$$C_{\text{opt}} = C^*$$

Wir wollen beweisen, dass $r(C) \leq 2r(C^*)$ bzw. $\frac{1}{2}r(C) \leq r^*(C)$

Deshalb müssen wir widerlegen, dass $\frac{1}{2}r(C) > r^*(C)$

Beweis durch Widerspruch:

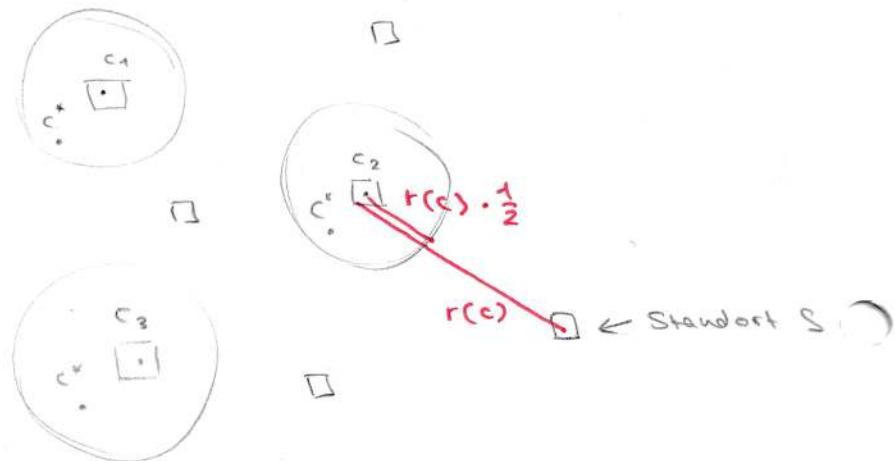
- Alle Mittelpunkte c die zugleich Standorte sind werden mit $\frac{1}{2}r(C)$ von der optimalen Lösung C^* abgedeckt
- Deshalb muss im Radius $\frac{1}{2}r(C)$ von jedem S mindestens ein c^* sein
- Weil für $c_1, c_2 \in C$ gilt:
 $\text{dist}(c_1, c_2) \geq r(C)$

Wissen wir, dass im Radius $\frac{1}{2}r(C)$ nicht $2 c \in C$ sein können!

$$\text{dist}(c_1, c_2) > \frac{1}{2}r(C)$$

$$[k=3]$$

sowohl bei C
als auch bei C^*

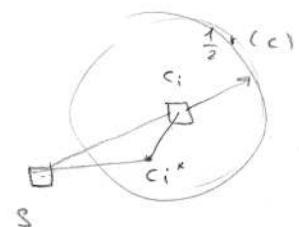


$$\underbrace{\text{dist}(S, c^*)}_{C^* \text{ muss zu } S \text{ näher sein}} \leq \text{dist}(S, c_i) \leq \underbrace{\text{dist}(S, c_i^*)}_{\leq r(C^*)} + \underbrace{\text{dist}(c_i^*, c_i)}_{\leq r(C^*)} \leq 2r(C^*)$$

als C

$$\text{dist}(S, c_i) \leq \text{dist}(S, c_i^*) + \text{dist}(c_i^*, c_i)$$

Dreiecksungleichung



(Siehe Seite 58)

Greedy-Algorithmus - Analyse

Theorem:

C^* optimale Menge an Mittelpunkten

C greedy Menge an Mittelpunkten

Wir nehmen an, dass $r(C) \leq 2r(C^*)$

Beweis durch Widerspruch:

(Wir wollen widerlegen, dass $\frac{1}{2}r(C) > r(C^*)$ weil dadurch)
 $\Rightarrow \frac{1}{2}r(C) \leq r(C^*)$ und $r(C) \leq 2r(C^*)$

Annahmen $\frac{1}{2}r(C) > r(C^*)$:

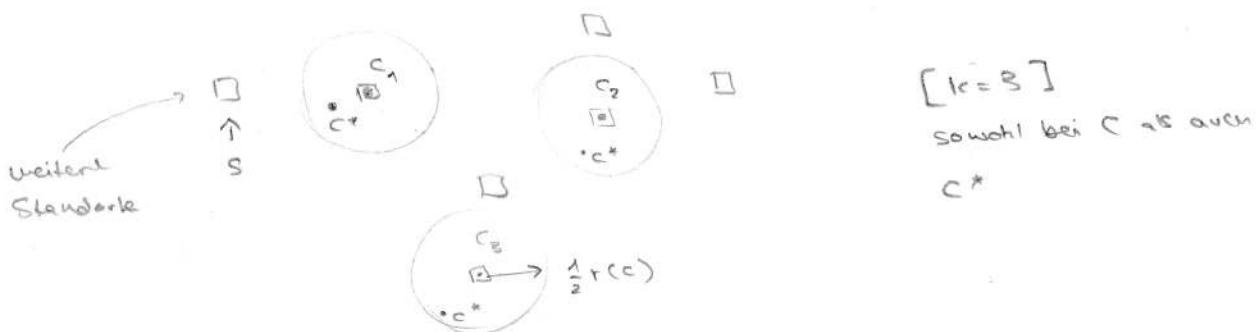
- Man beachte die k Mittelpunkte in C die zugleich Standorte sind,
mit einem halben Radius: $\frac{1}{2}r(C)$

weil für $c_1, c_2 \in C$ gilt $\text{dist}(c_1, c_2) \geq r(C)$

wissen wir mit Sicherheit, dass sich Radien von $\frac{1}{2}r(C)$ nicht überdecken:

$$\text{dist}(c_1, c_2) > \frac{1}{2}r(C)$$

- Weil $r(C^*) < \frac{1}{2}r(C)$ müssen $c_i^* \in C^*$ aber innerhalb dieser Radien sein



- Man nehme Standort außerhalb von $\frac{1}{2}r(C)$ von c_i^*
und nenne sie S .

$$\text{dist}(S, c_i^*) \leq \underbrace{\text{dist}(S, c_i)}_{\substack{\text{Mittelpunkt } c^* \\ \text{muss näher sein als } C}} + \underbrace{\text{dist}(c_i^*, c_i)}_{\substack{\text{Dreiecksungleichung}}} \leq 2r(C^*)$$

$\frac{1}{2}r(C) \leq r(C^*)$

$$\frac{1}{2}r(C) \leq r(C^*)$$

Dreiecksungleichung



Wenn $r(c) \leq r(c^*) + 2$

$$\frac{1}{2} r(c) \leq r(c^*)$$

dann $\downarrow 2 \cdot r(c) \cdot \frac{1}{2} > r(c^*)$

Optimierung - Heuristische Verfahren

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, SS 2019
Letzte Änderung: 25. Juni 2019
Vorlesungsfolien



1 / 69

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung

Approximation(salgorithmen)

Praxis-
Näher

{ Heuristische Verfahren: Erzeuge in polynomieller Zeit eine
Näherungslösung. In der Praxis kann eine solche Lösung häufig
sehr gut oder sogar optimal sein, es gibt aber keine Garantie dafür.

„Gütegarantie“

2 / 69

Heuristische Verfahren

1. Konstruktionverfahren

Poly Zeit ohne Gütekriterium, intuitiv

Ungewisse Greedy Heuristiken zur Abschätzung

Beispiel:

Anwendung in TSP (sym, metrisch)

Spanning Tree Heuristik

MST

Kanten verdoppeln

Euler Tour

kürzen

Nearest Neighbour Heuristik

Gehe zum nächsten unbesuchten Knoten

(ungenau: Viele Kreuzungen) \rightarrow 2-opt: 2 exchange

Insertion Heuristik

Partie konvexe Hülle mit N äußeren Knoten

Füge iterativ alle restlichen Knoten ein

a) zufällig \rightarrow schnell

b) best first \rightarrow langsam

2. Lokale Suche (Verbesserungsheuristiken)

Verbessern iterativ durch kleine Änderungen (Züge, Moves) durch Bestimmung der Nachbarschaft \rightarrow deutliche Verbesserung durch richtige Auswahl einer Nachbarschaft

x = Ausgangslösung

while (...)

$x' \in N(x)$

if x' besser als x

$x = x'$

Definition der Nachbarschaft bei Vertex Cover

Lokale Suche: remove 1

entferne 1 Knoten, sodass noch gültige Lösung

$O(n)$

Problem: steckt im lokalen Optimum

Lokale Suche: add 1, remove 2 (erweiterte Nachbarschaft)

Problem: aufwändiger

$O(n^3)$

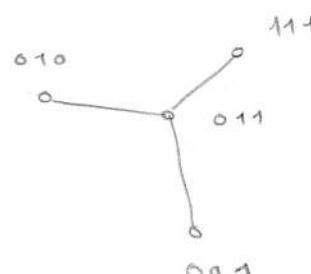
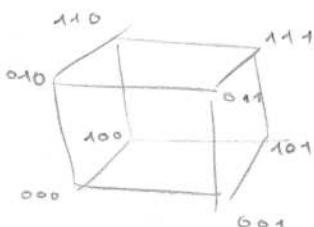
Wege um globales Optimum zu erreichen

- größere Nachbarschaften
- randomisierte Startpunkte
- Kombination

Definition der Nachbarschaft bei MAX-SAT

Lösungsvektor: 2^n Möglichkeiten $\{0,1\}^n$

Lokale Suche: Flip k Bits



Flips 1 bit:

3 Dimensionen
Würfel

k Dimensionen
Hyperwürfel

Auswahl der Nachbarn

- best improvement
- next improvement (erste Lösung die besser ist)
- random

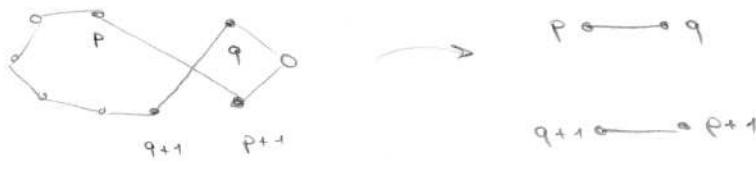
Abbruchkriterium

- keine Verbesserung möglich
(bei Random nicht möglich)
- nach x Iterationen

Definition der Nachbarschaft für TSP - Heuristiken

2-exchange oder 2-opt:

Tausche wenn sinnvoll



$O(n^2)$

Schritte zur Bestimmung einer Nachbarlösung

Worst case:

$O(n!)$ Iterationen mit je $O(n^2)$ Schritten

r-exchange oder r-opt

↳ Viel mehr Kombinationen möglich bei $r \geq 2$

Laufzeitanalyse:

$\binom{n}{r} \in O(n^r)$ Möglichkeiten r Kanten aus einer Tour zu entfernen

$O(r!)$ Möglichkeiten zurückzusetzen

$$INC(x) \in O(n^r \cdot r!) \in O(n^r)$$

Verschieben eines Knotens

Liu - Kernighan - Heuristik

↳ Heute die beste Variante

Maximaler Schnitt

MAX CUT

Ist NP-Vollständig

Teile gewichteten Graphen mit Gewichten ≥ 0 in 2 Partitionen A, B

So dass $w(A, B) := \sum_{\substack{u \in A \\ v \in B}} w_{uv}$ maximiert wird

Bestimmung eines lokalen Optimums mit „Flip-Nachbarschaft“:

$N(x)$: Verschiebe Knoten in andere Partition

Bedingung für lokales Optimum:

$$\forall u \in A: \sum_{v \in B} w_{uv} \leq \sum_{v \in A} w_{uv}$$

$$\forall u \in B: \sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Aufsummiert:
(wegen Handshake alle Kanten doppelt betrachtet)

$$2 \sum_{\substack{u \in A \\ v \in B}} w_{uv} \leq \sum_{\substack{u \in A \\ v \in A}} w_{uv} + \sum_{\substack{u \in B \\ v \in B}} w_{uv} = w(A, B)$$

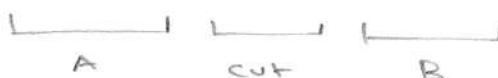
$$2 \sum_{\substack{u \in B \\ v \in A}} w_{uv} \leq \sum_{\substack{u \in A \\ v \in A}} w_{uv} + \sum_{\substack{u \in B \\ v \in B}} w_{uv} = w(A, B)$$

Davon folgt:

$$\sum_{\substack{u \in A \\ v \in A}} w_{uv} \leq \frac{1}{2} w(A, B)$$

$$\sum_{\substack{u \in B \\ v \in B}} w_{uv} \leq \frac{1}{2} w(A, B)$$

$$\sum_{e \in E} w_e = \sum_{\substack{u \in A \\ v \in A}} w_{uv} + \sum_{\substack{u \in A \\ v \in B}} w_{uv} + \sum_{\substack{u \in B \\ v \in B}} w_{uv} \leq \underline{2 w(A, B)}$$



$$\leq \frac{1}{2} w(A, B) \quad w(A, B) \quad \leq \frac{1}{2} w(A, B)$$

$$\frac{1}{2} \sum_{e \in E} w_e \leq w(A, B)$$

Wir haben bewiesen, dass:

$$w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e$$

Und damit auch für das globale Optimum: $w(A^*, B^*)$

$$w(A^*, B^*) \geq \frac{1}{2} \sum_{e \in E} w_e$$



Theorem im Skriptum:

$$w(A, B) \geq \frac{1}{2} w(A^*, B^*) \quad \text{weil } \varepsilon = \frac{1}{2}$$

Formel im Skriptum:

$$w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} w(A^*, B^*)$$

_____ |

$$\text{wenn } w(A^*, B^*) \geq \frac{1}{2} \sum_{e \in E} w_e$$

Dann:

$$\frac{1}{2} w(A^*, B^*) \leq \frac{1}{2} \sum_{e \in E} w_e$$

3. Metaheristiken

Erweiterung der lokalen Suche

„problem-unabhängig“

Simulated Annealing SA

Um lokale Optima zu vermeiden

Simuliert Abkühlung um stabile Kristallstruktur zu erreichen:

- | Akzeptanz von schlechten Nachbaren zu unterschiedlichen Zeiten
- | mit verschiedenen Wahrscheinlichkeiten

Pseudo-Zufallszahl $z \in [0; 1]$

„Metropolis-Kriterium“:

$$z < e^{-|f(x') - f(x)| / T} \leftarrow \text{Temperatur}$$

wenn das zutrifft wird schlechtere Lösung auch akzeptiert

> Eigenschaft:

geringfügig schlechtere Lösung \rightarrow höhere WSL

viel schlechtere Lösung \rightarrow niedrigere WSL

Abkühlungsplan

Wie sich T nach jeder Iteration ändern soll

(kann sich dem Fortschritt anpassen)

Zusammenfassung:

- Einfache Implementierung
- Bereit Parameterierung

Anwendungsbeispiel:

Graph-Bi-Partitionierung (Quasi MINCUT)

Lösungsdarstellung: $\vec{y} = \{0,1\}^n$

Bool'scher Vektor:

$$n = |V|$$

$$x_i = 0 \rightarrow y_i \in A$$

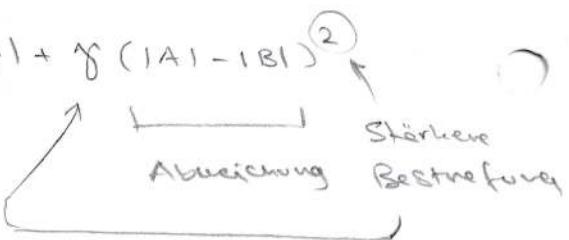
$$x_i = 1 \rightarrow y_i \in B$$

Nachbarschaft: Bit-Flip $O(n^2)$

Anwendung von 'Simulated Annealing':

- Zufällige Anfangslösung
- „Geometrisches Abkühlen“
- Ungültige Lösungen erlaubt aber in Zielfunktion „bestrafen“

$$f(A, B) = |\{(u, v) \in E \mid u \in A, v \in B\}| + \gamma (|A| - |B|)^2$$



Tabu-Suche

- Best-Improvement (bester Nachbar, auch wenn schlechter als aktuelle Lösung) Iteration
- Vermeidung von Zyklen
- Gedächtnis (History): Speichert bisherige Moves von t_L Iterationen
Sehr viel Speicherverbrauch

Algorithmus:

Tabu-Suche ()

$x_{best} = x = \text{Ausgangslösung}$

TabuListe $\leftarrow \{x\}$

while (...)

$x' \leftarrow \text{Teilmenge von } N(x) \text{ abhängig von TabuListe}$

] Nachbarschaft bestimmen

$x' \leftarrow \text{beste Lösung aus } N(x)$

TabuListe = TabuListe $\cup \{x'\}$

Löse Elemente aus TabuListe die älter als t_L Iterationen sind

$x \leftarrow x'$

Wichtigster Parameter:

if $x > x_{best}$

„TabuListenlänge“

$x_{best} = x$

zu kurz: Zyklen

zu lang: Steckenbleiben

Man bestimmt auch
TabuAttribute die verboten
Sind

Nach

Aspirationskriterien werden
manche Tabu-Lösungen
trotzdem erlaubt wenn
Sie gut genug sind

Evolutionäre Algorithmen

→ Vorteil: Parallelisierung möglich

1. Bestimmung der Population

Ausgangslösungen

2. Selektion

Bessere Lösungen → Höhere Selektionswahrscheinlichkeit

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^n f(x_j)} \quad \begin{array}{l} \text{Fitness von } x_i \\ \sum \text{ andere} \end{array}$$

wird gesteuert über
lineare Funktion

$$g(x_i) = [a] f(x_i) + [b]$$

"Selektionsdruck"

$$S = \frac{P(x_{\max})}{P}$$

Zu niedrig:
Gleichverteilung
Zu hoch:
Tyrannei

Alternative:

Tournament - Selektion

(keine Skalierung notwendig)

Selektionsdruck = Gruppengröße

3. Rekombination

Vererbung

4. Mutation