

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



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					
K I N D E R	Xaver	Abel	Boole	Church	G A S T F A M I L I E N	Abel	Yvonne	Xaver	Zola
	Yvonne	Boole	Abel	Church		Boole	Xaver	Yvonne	Zola
	Zola	Abel	Boole	Church		Church	Xaver	Yvonne	Zola
<i>Präferenzlisten der Kinder</i>					<i>Präferenzlisten der Familien</i>				

Stable Matching Problem

Stable Matching

keine instabiler Matches (Paare)

instable Matching

Partner-Wechsel vorteilhaft
für Familie und Kind,



Das bessere Paar

Gale-Shapely-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 ($f = \text{frei}$)

Match ($s-f$)

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

Match ($s-f$)

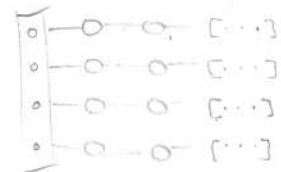
$s' = \text{frei}$

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

$F_{\text{Pref}}[s, i]$ - Array Bestimmt von s , die bevorzugte Familie an Stelle i

$S_{\text{Pref}}[f, i]$ - Array Bestimmt von f , das bevorzugte Kind an Stelle i

S_{Free} - Queue

—
Rang

$\text{Next}[s]$ - Array

Nächste noch nicht überprüfte Stelle im Kind-Array

Nächste Familie: $F_{\text{Pref}}[s, \text{Next}[s]]$

$\text{Current}[f]$

Match von f bzw. "-1" wenn frei

$\text{Ranking}[f, s]$

Wert von allen Matchings

(wird vor Algorithmus erstellt)

Eigentlich $S_{\text{Pref}}[f, i]$ nur statt i kann man direkt nach Kind suchen

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

for $i \leftarrow 0$ bis $n-1$

for $j \leftarrow 0$ bis $n-1$

$\text{Ranking}[i, F_{\text{Pref}}[i, j]] \leftarrow j$

GS-Algorithmus:

Folien 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

$$a_n \in O(b_n) \quad \text{und} \quad b_n \in O(a_n)$$

Daraus folgt

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

Asymptotische Dominanz

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

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

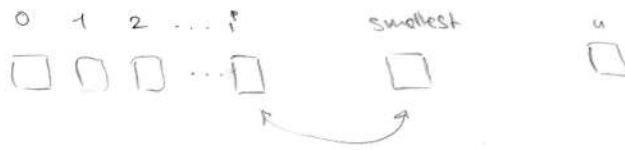
Sortieralgorithmen als Beispiel

Bubblesort $\Theta(n^2)$

bekannt aus EP 1

Selectionsort $\Theta(n^2)$

Suche Minimum aus unsortierten Teil



Vertauscht jede Zahl mit $i \rightarrow \frac{n(n-1)}{2} \in \Theta(n^2)$

Insertionsort $\Theta(n^2)$

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

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

Beispiel für Laufzeitanalyse:

(ALTE
UNTERLAGEN)

Oberer Schranke

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

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

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

$\forall n \geq n_0$ gilt:

$$T(n) \leq c \cdot f(n)$$

konstante Faktoren
ausgeschlossen

$$T(n) = 32n^2 + 17n + 8$$

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

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

$$32n^2 + 17n + 8 \leq c \cdot n^2 \quad | : n^2$$

$$32 + \frac{17}{n} + \frac{8}{n^2} \leq c$$

→ Beispielsweise gültig wenn

$$n \geq 18$$

$$c \geq 34$$

$$n_0 = 18$$

$$c = 34$$

Untere Schranke

$$T(n) = 32n^2 + 17n + 8$$

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

$$\text{---} \text{---} \text{---} T(n) \geq c \cdot f(n)$$

$$32n^2 + 17n + 8 \geq c \cdot n^2$$

$$32 + \frac{17}{n} + \frac{8}{n^2} \geq c$$

→ Beispielsweise gültig wenn

$$n \geq 1$$

$$c \leq 32$$

$$n_0 = 1$$

$$c = 32$$

Scharfe Schranke

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

Hier trifft das wie ~~gen~~ bereits gezeigt zu.

Demnach:

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

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

$$c_1 = 32$$

$$c_2 = 34$$

$$n_0 = 18 \leftarrow \text{Max}(1, 18)$$

✓

Asymptotische Analyse, Beispiel

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

Scharfe Schranke

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

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

Trotzdem mit Rechnung

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:

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

↓

$$c_1 = 32 \quad n_0 = 18 \quad (\max(1, 18))$$

$$c_2 = 39$$

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

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

Behauptung:

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

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

Beweis:

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

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

→ gültig für

$$n \geq 18 \quad c \geq 39$$

Best Case:
(beste Approximation)

$$n_0 = 18$$

$$c = 39$$

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:

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

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

→ gültig für

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

$$\rightarrow n_0 = 1$$

$$c = 32$$

Es gilt

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

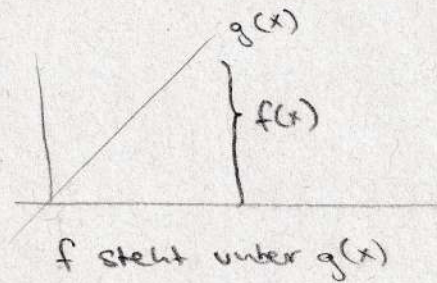
Wichtig:

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

Asymptotische Dominanz

$$\begin{array}{l} g \text{ dominiert } f \\ f \ll g \end{array}$$

$$\begin{array}{l} \rightarrow f(x) \in O(g(x)) \\ g(x) \notin O(f(x)) \end{array}$$



Beispiel

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

$$\begin{array}{l} g(n) = n^3 \\ f(n) = n^2 \end{array} \left\{ \begin{array}{l} f(n) \in O(g(n)) \text{ aber} \\ g(n) \notin O(f(n)) \end{array} \right.$$

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

Sie 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

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

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)

$$O(n^d) \text{ und } \Omega(n^d)$$

Logarithmen

$$a, b > 0$$

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

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

($\log_b a$ ist konstante)

Deshalb gibt man bei asymptotischen Angaben die Basis des Logarithmus 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:

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

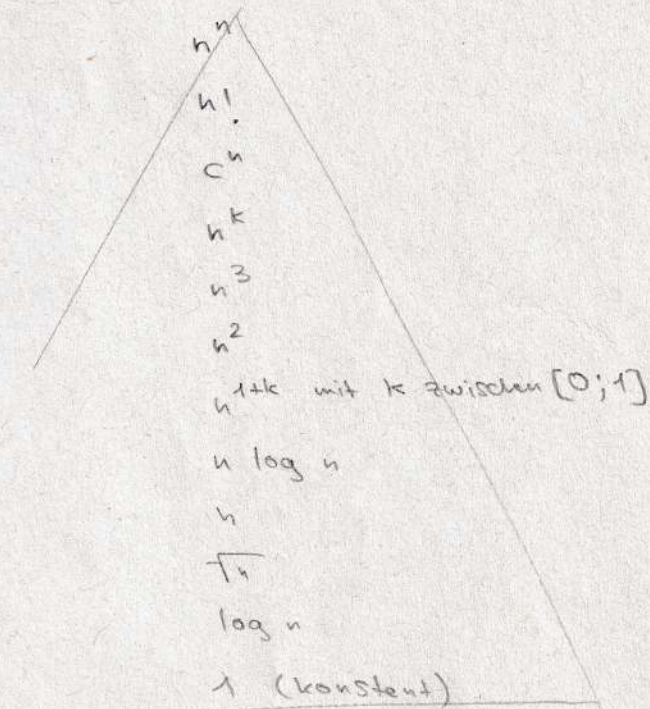
$$c_1^n \ll c_2^n$$

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

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

→ $f(n) \in$ Äquivalenzklasse von $g(n)$ in der alle Funktionen sind die äquivalent sind. „Schranken“

Landau-Notation

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

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

↑
klein-0

Man benutzt die Landaunotation für Zeitkomplexität → „Asymptotische Laufzeit“

Nicht Aufwand auf einem bestimmten Rechner

Die Schranken der Landau Notation:

$$\underline{\Omega}(g(n)) \leq T(n) \leq \underline{O}(f(n))$$

untere
Schranke

tatsächl.
Zeitaufw.

obere
Schranke

Sondern Relation zwischen Eingabemenge und Zeit bei großen Eingaben

„Skalierbarkeit“

⊖ Scharfe Schranke:
wenn obere und untere Schranke identisch sind

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 $\Omega(f(n))$

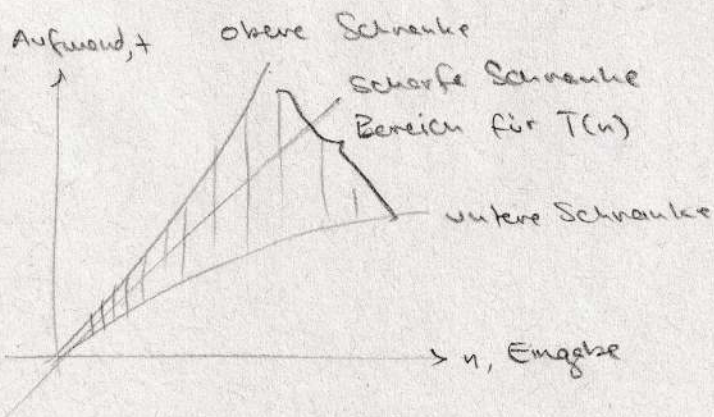
von unten beschr. \rightarrow Menge \forall Funktionen die von $f(n) \cdot c$ unten beschränkt werden asymptotisch

$T(n) \in \Omega(f(n))$
 $\exists n_0 > 0, \exists c > 0$ 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 \forall 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 (1 = Knoten sind adjazent)

Nicht geeignet wenn Graph nicht dicht ist (dichter Graph: zB vollständiger

Graph mit $\frac{n \cdot (n-1)}{2}$ Kanten

nach Handshake Theorem

also $\Theta(n^2)$)

Platzbedarf $\Theta(n^2)$

check von Adjazenz $\Theta(1)$

Aufzählen aller Kanten $\Theta(n^2)$

Adjazenzlisten

Liste an adjazenten Knoten für jeden Knoten

Platzbedarf $\Theta(m+n)$

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

Aufzählen aller Kanten $\Theta(m+n)$

Knoten $|V| = n$

Kanten $|E| = m$

Breitensuche - Ermittlung von Ebenen

BFS - Breadth first search

BFS(G, s)

Level[s] = 0

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

Queue Q ← {s}

While Q ist nicht leer

u ← Q.pop

foreach Knoten adjazent zu u (v)

if Level[v] = -1

Level[v] = Level[u] + 1

Q.add(v)

→ Level L_i hat Distanz i zu Startknoten

} Initialisierung $\Theta(n-1)$

Jeder Knoten der Adjazenzliste wird durchgesehen

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

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

DFS - Depth first search

DFS(G, s):

Discovered[v] = false (für alle Knoten)

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)

} $\Theta(n)$

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

Beide in $O(n+m)$

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

Starke Zusammenhängekomp. mit BFS bestimmen

1. beliebiger Knoten s

2. BFS(G, s)

3. BFS(G^{rev}, s) ← G^{rev} = Alle Kanten verdreht

- ist stark zusammenh.

wenn alle Knoten in Schritt 2 und 3 erreicht werden konnten

gerichtete azyklische Graphen

DAG - directed acyclic graph

```
while  $\exists v \in V$ 
  if  $\exists$  Quelle  $q$ 
    lösche  $q$  und alle inzidenten 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
foreach  $v \in V$ 
  foreach Knoten  $w$  adjazent zu  $v$ 
    count [ $w$ ] = count [ $w$ ] + 1
foreach  $v \in V$ 
  if count [ $v$ ] = 0
     $Q.add(v)$ 
```

Insgesamt:
 $O(n+m)$

```
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

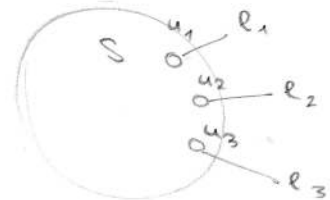
Algorithmus:

1. Initialisiere $S = \{s\}$ $d(s) = 0$
2. Wähle Knoten v wobei $v \notin S$ sodass

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

3. Füge v zu S zu,
setze $d(v) = \min_{e=(u,v): u \in S} d(u) + l_e$

4. Bezeichne Pfad



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

Dijkstra (G, s)

Discovered $[v] = \text{false}$

$d[s] = 0$

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

$Q \leftarrow V$

while Q ist nicht leer \leftarrow (Priority Queue / Heap)

wähle $u \in Q$ mit geringstem Wert $d[u]$

lösche u aus Q

Discovered $[u] = \text{true}$

foreach Knoten adjazent zu u (v)

if !Discovered $[v]$

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

lösche v aus Q

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

$Q.add(v)$

Analyse von Dijkstra

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

$O(u)$ Array initialisierung

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

$O(m)$ foreach (jede Kante max $1 \times$)

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

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

$O(u)$ Array initialisierung

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

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

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

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

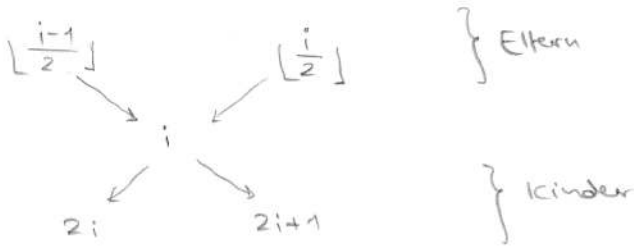
Priority Queue - Datenstruktur:

MIN-HEAD

Ordnung:

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

Darstellung als Array:



Wichtig:

wenn man mit Index 1 anfängt

$$L = 2i$$

$$R = 2i+1$$

Wenn man mit Index 0 anfängt

$$L = 2i+1$$

$$R = 2i+2$$

Operationen

Heap aus Array bilden $O(n)$

for $i = \lfloor \frac{n}{2} \rfloor$ bis 1 \leftarrow 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)

\leftarrow 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)

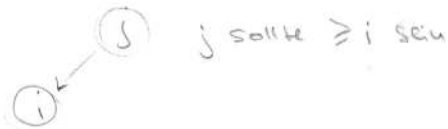
if $i > 1$

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

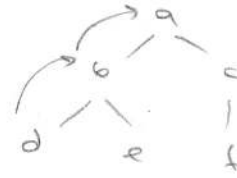
if $A[i] < A[j]$

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

Heapify-up (j)



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 $2i < n$

left = $2i$

right = $2i + 1$

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

else if $2i = n$

$j = 2i$

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

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

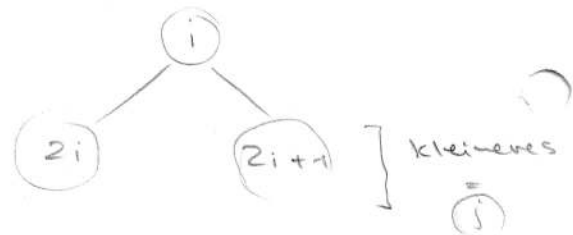
Heapify-down (j)

leaf erreicht

\exists links, \exists rechts

\exists links, \nexists 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 :

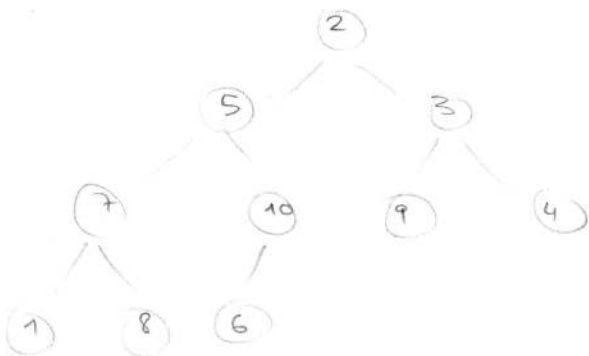
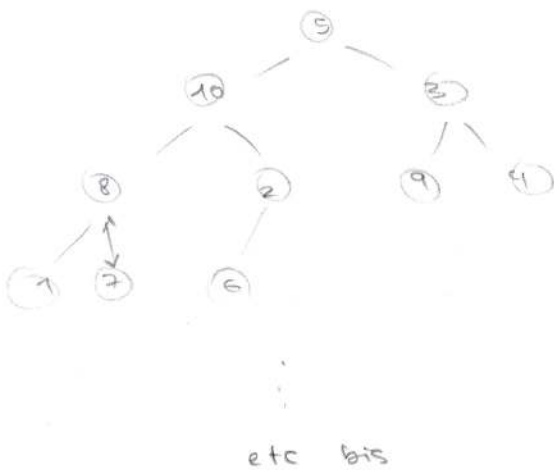
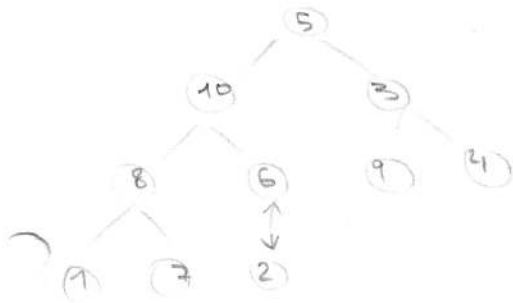
$$\lfloor \frac{n}{2} \rfloor = 5$$

Rechtes Kind : \leq
Linkes Kind : $>$

Lösung: Mit (A, n) :

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

Heapify-down (A, i)



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

Aufangen mit größtem Wert

Interval Scheduling

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

Startzeit s_j

Finishzeit 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

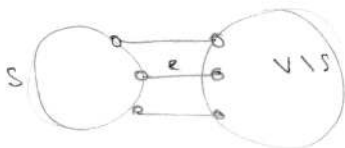
Nicht Hamiltonproblem: geschlossener Pfad, der alle Knoten beinhaltet

Gesucht:

Spannender Baum (offen, azyklisch) zwischen allen Knoten, mit minimalem Kantengewicht.

Basierend auf 3 Lemmata:

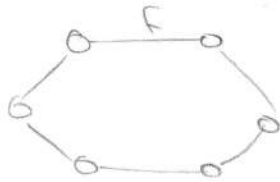
1. Kantschnittlemma



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. Kreislemme



In einem Kreis kann die Kante mit max Gewicht ausgelassen werden

Conclusion:

immer Kante mit min Gewicht nehmen,
keine Kreise bilden

3. Paritätstemme

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

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

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

Laufzeiten:

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 unionfind, operationen in fast $O(1)$

Der zweite Teil fast $O(n)$

Sortieren:

$O(m \log n)$

MST - Algorithmus von Prim

$O(m \log n)$

1. Initialisiere S mit beliebigen Knoten

2. Wende das Kantenschrittverfahren an:

Wähle billigste Kante von Kantenschrittmenge (S) und

füge Knoten auf anderer Seite S hinzu

Prim(G, c)

foreach ($v \in V$)

$A[v] \leftarrow \infty$

Priority Queue $Q \leftarrow \emptyset$

foreach ($v \in V$)

$Q.add(v)$

$S \leftarrow \emptyset$

Initialisierung

while Q ist nicht leer

$u \leftarrow$ minimales Element aus Q

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

foreach Knoten v adjazent zu u

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

$A[v] = c_e$

MST - Algorithmus von Kruskal

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

Kruskal (G, c)

Sortiere Kantengewichte $O(m \log n)$

$T \leftarrow \emptyset$

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)

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

vereine u und v

return T

1. Möglichkeit: BFS

2. Möglichkeit: Union-Find
Datenstruktur

Union-Find - Datenstruktur

DDM „Dynamische diskrete 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(u, w) Merged $S_u \cup S_w \rightarrow u$ 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 in Merge-Array gelegt wird

Laufzeit

$C(n)$... # comparisons bei n Elementen

$$C(n) \leq \begin{cases} 0 & \text{wenn } n=1 \\ C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n & \text{sonst} \end{cases}$$

L
 R
verschmelzen

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

$\Theta(n \log n)$ \rightarrow 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

	Mergesort	Quicksort
Laufzeit:		
worst	$n \log n$	n^2
avg	$n \log n$	$n \log n$
best	$n \log n$	$n \log n$

Speicher:

worst	n	n
avg	n	$\log n$
best	n	$\log n$

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 $\{A, B\}$

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

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

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

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

Merge-and-count (A, B)

Mergesort, dass beim mergen Inversionen zählt

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

Laufzeit

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

Eigentlich = Mergesort

Dichtestes Punktepaar

in euklidischem Raum

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

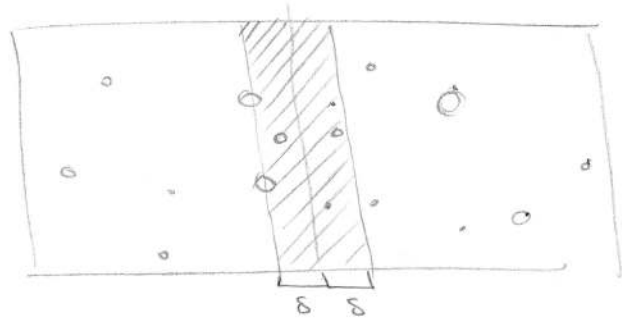
Algorithmus:

1. Teile mit Linie, sodass
in jeder Hälfte ca. gleich
viele sind

2. Berechne Rekursiv L, R

3. Berechne Mitte

4. retourniere $\max(L, R, \text{Mitte})$



→ beobachte nur Teilraum mit Abstand $\min(L, R)$ von Mittellinie = δ

Sortiere Punkte nach y -Koordinate

bestimme Greedy eine Lösung \rightarrow 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 Schlüssel.

2 / 74

Suchbäume

Durchmusterungen (Traversals)

Inorder L N R

Preorder N L R

Postorder L R N

Binäre Suchbäume

$L \leq N \leq R$

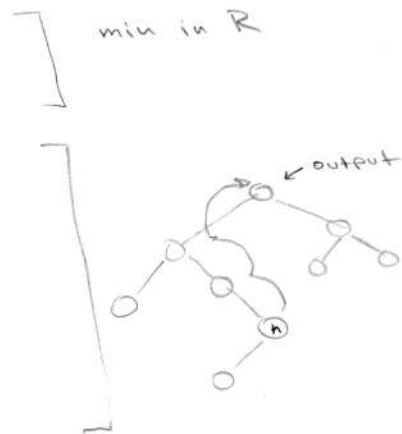
Bestimmung von Vorgänger / Nachfolger

Successor (n):

if $n.right \neq null$
return Minimum ($n.right$)

if $n.right = null$
 $p \leftarrow n.parent$
while $p \neq null \wedge n = p.right$
 $n \leftarrow p$
 $p \leftarrow p.parent$

return p;

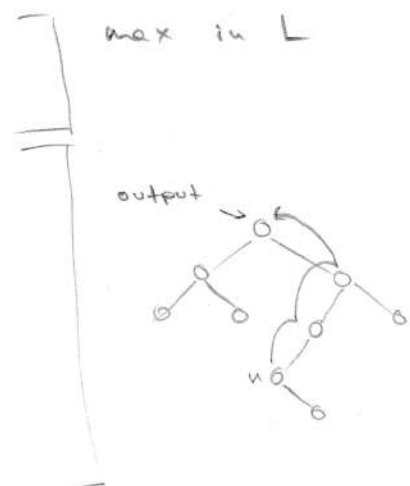


predecessor (n):

if $n.left \neq null$
return Maximum ($n.left$)

if $n.left = null$
 $p \leftarrow n.parent$
while $p \neq null \wedge n = p.left$
 $n \leftarrow p$
 $p \leftarrow p.parent$

return p;



Einfügen

nach Node suchen und an richtiger Stelle einfügen

Entfernen

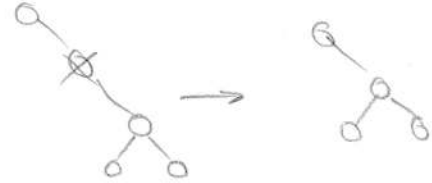
Case 1: 0 Kinder = leaf

abschneiden



Case 2: 1 Kind

wie bei verketteter 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 Rotationsbäume: Höhenbalancierung

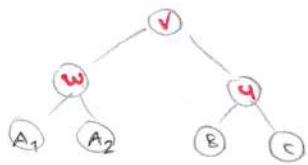
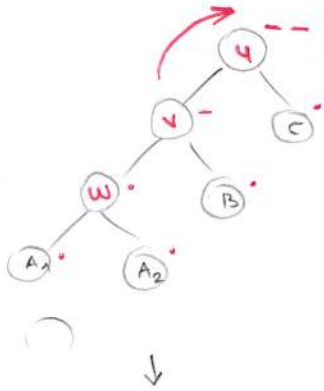
AVL - Bäume

Balance := R.Tiefe - L.Tiefe

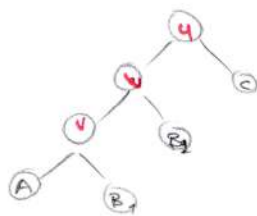
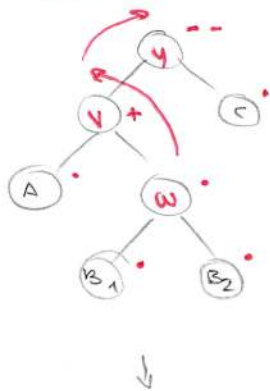
erlaubt: $\{-1, 0, 1\}$

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

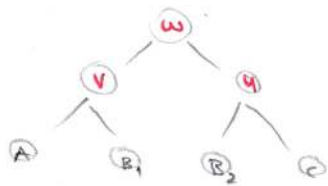
Fall 1.1



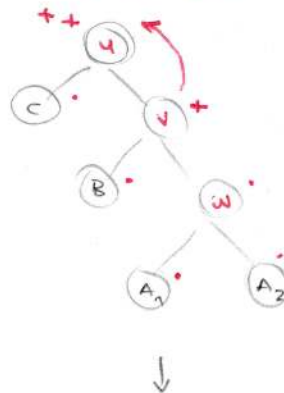
Fall 1.2



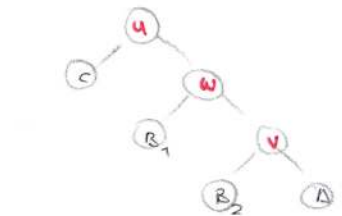
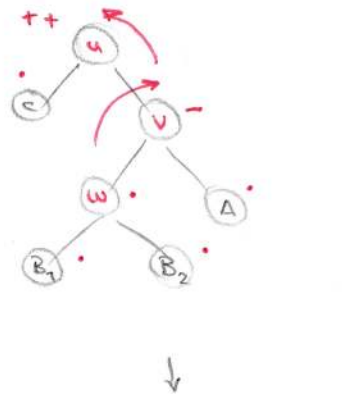
Fall 1.1



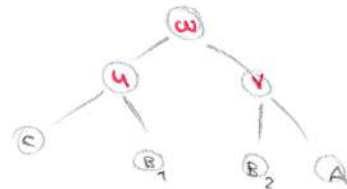
Fall 2.1



Fall 2.2



Fall 2.1



Rotation

Beispielsweise Fall 1.1: r.child[4] wird zu l.child[3]

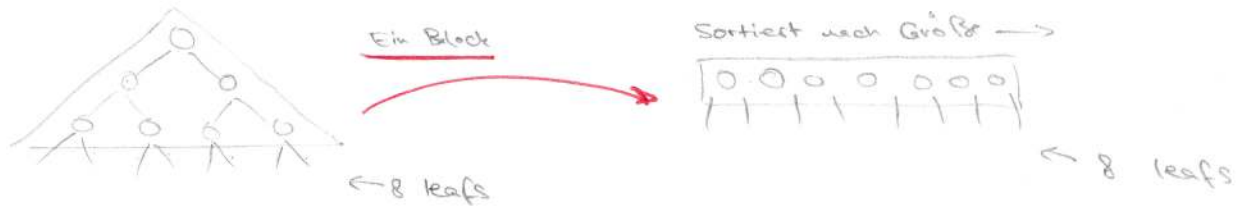
3 wird root

Logarithmische Laufzeit $O(\log_2 n)$

Beim Einfügen/löschen maximal 2 Rotationen

B-Baum

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



Constraints:

B-Baum mit Ordnung m (m Kinder, $m-1$ keys (Einträge) 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



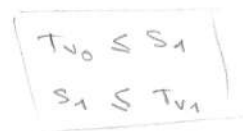
Beispiel



Mau erkennt:

Alle Schlüssel in T_{i-1} sind $\leq S_i$

S_i ist \leq alle Schlüssel in T_{vi}



Implementierung

$p.l$ Schlüssel-Anzahl

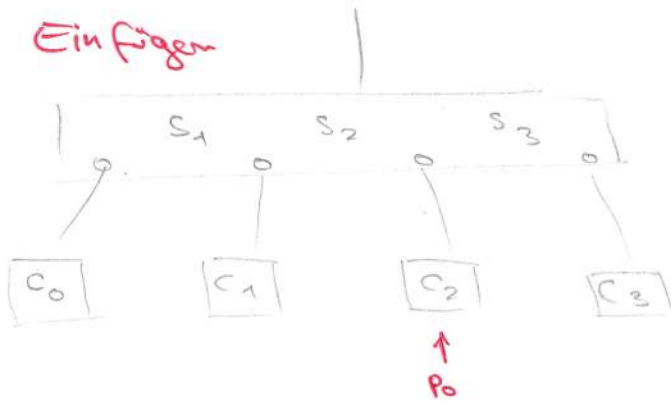
$p.key[1] \rightarrow p.key[l]$ Schlüssel S_1, \dots, S_l

$p.info[1] \rightarrow p.info[l]$ Zahlen in S_1, \dots, S_l

$p.child[0] \rightarrow p.child[l]$ Kinder

\rightarrow bei Blättern: $p.l = 0$

Einfügen



Hier:

$$p.l = 3$$

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

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

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

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

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

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

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

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

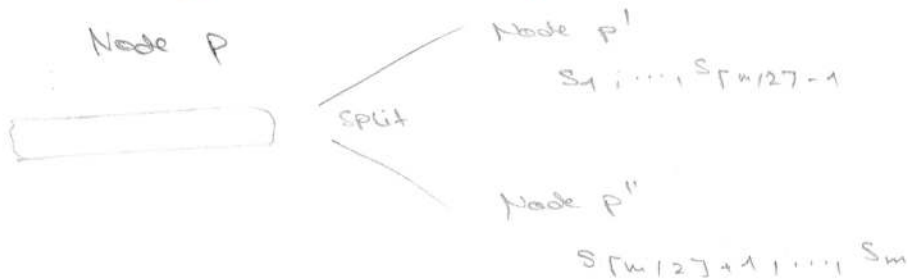
hier: $p_0 = c_2$ ($i=2$) also ist p_0 zwischen s_2 und s_3

3a. Es gibt Platz für mehr keys,

Deshalb zwischen s_i und s_{i+1} einfügen

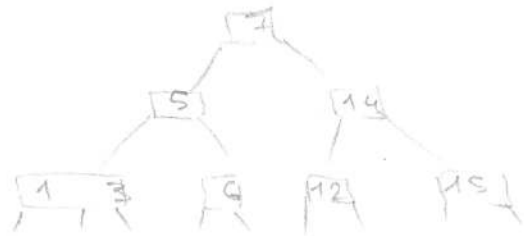
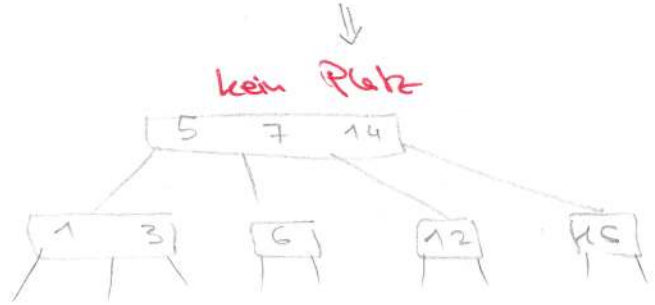
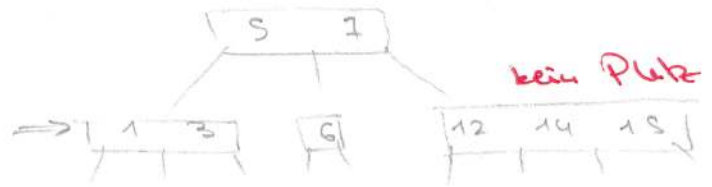
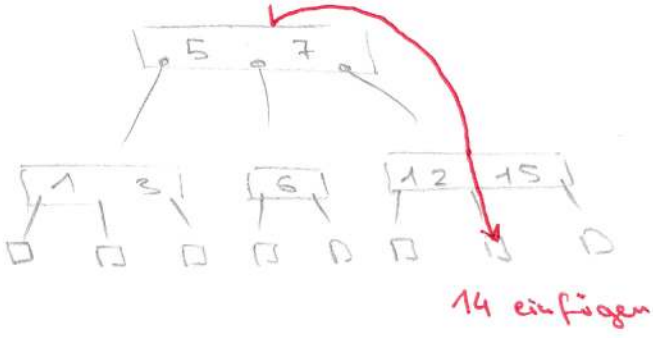


3b. Es gibt keinen Platz, deshalb Node in 2 Teile splitten



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

Bäume wachsen immer von 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

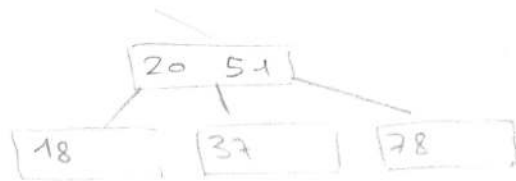
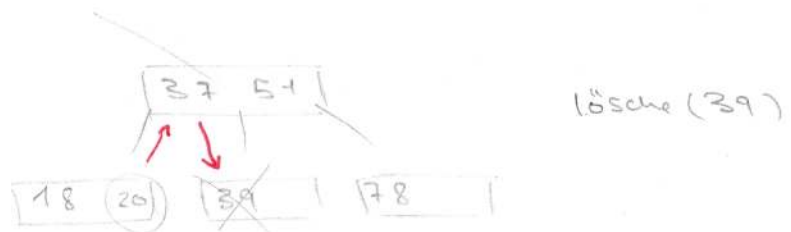
case 1: k' ist in unterster Ebene

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

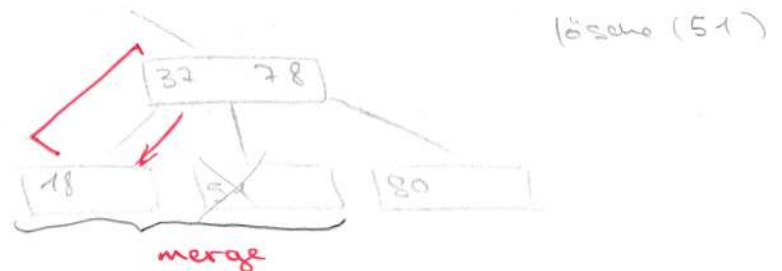
case 2: k' ist in unterster Ebene

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

a) Nachbar links oder rechts hat genug keys zum hergeben



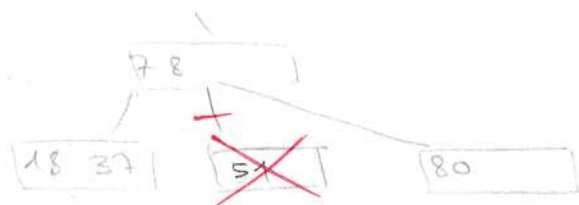
b) Nachbarn haben nicht genug zum hergeben



1. Eltern zu Kind

2. leere Node löschen

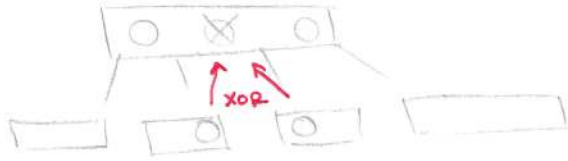
3. mergen



Case 3: k' nicht in unterster Ebene

k' entfernen, mit Ersetzschlüssel ersetzen

Prozess neu aufbauen und Ersetzschlüssel löschen



Ersetzschlüssel:

größter im linken Kind

kleinsten im rechten Kind

Eigenschaften

leaves im Baum + 1 = # keys

Beweis:

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

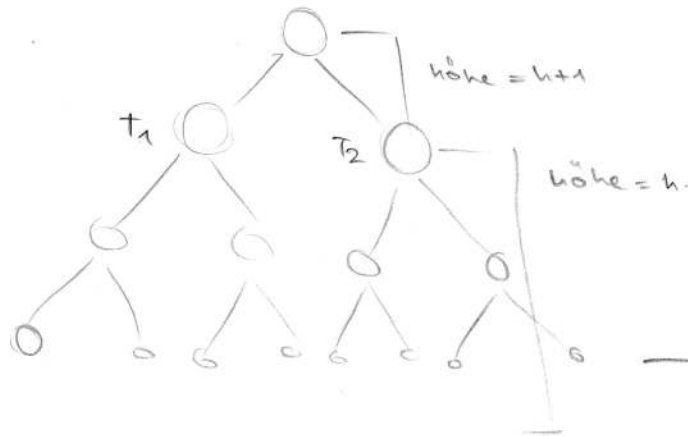
höhe = 1 Wurzel hat k leaves und $k-1$ keys wobei $2 \leq k \leq n$

Wir setzen voraus, dass das auch für Höhe h gilt

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

höhe = $h+1$ k Unterbäume $T_1 \dots T_k$ mit Höhe h , für die das gilt, mit n_1, \dots, n_k Blättern
 \uparrow
 je $(n_i - 1)$ Schlüssel

IA
 IV
 IS



$$\rightarrow \sum_{i=1}^k n_i \text{ Blätter}$$

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

Mit einfachen Worten:

Jede einzelne Ebene des Baumes hat

Pro Node 1 Schlüssel weniger als Blätter

und das trifft deshalb summiert im ganzen Baum zu

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

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

↑
linke und rechte
Seite der W

Allgemein gilt:

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

Blätter:

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

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

↑
Höhe

(wenn alle halbgefüllt sind.)

Deshalb

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

Alternativer Beweis:

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

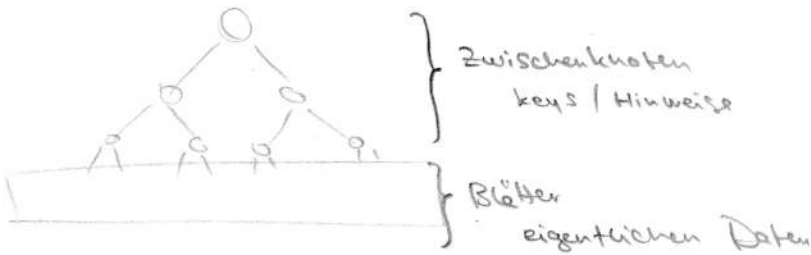
Baum mit 3 Kindern: $O(\log_3 n)$

4 Kindern: $O(\log_4 n)$

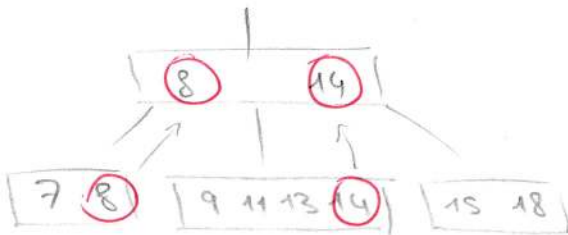
B*-Bäume

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 Keys 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 Kollisionfrei dann:

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

Sonst worst-case $O(n)$

Hashfunktionen

„Modulo-Methode“

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

„Multiplikationsmethode“

irrationale Zahl $A \rightarrow$ optimal: goldenes Schnitt

$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

$\in [0, 1)$

Kollisionsbehandlung

„Verkettung der Überläufer“

Bei Kollision in Hash-Tabelle \rightarrow einfache verkettete 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 Sondiermaßfunktion $h(k, i)$

bestimmt Suchreihenfolge nach freien Plätzen

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

2. Lineare Sondierung

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

Berechnung der \emptyset Zeit für erfolgreiche Suche:

$$\frac{\sum \text{Suchiterationen für alle Einträge}}{\# \text{ Einträge}}$$

Problem:

Quasi-Verstopfung indem belegte Teilstücke immer länger werden: primary clustering

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

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \text{ mod } m$

Problem: secondary Clustering

4. Double Hashing

$h_1(k)$: hashfunktion 1
 $h_2(k)$: hashfunktion 2

$h(k, i) = (h_1(k) + i h_2(k)) \text{ mod } m$
 (fast uniform!)

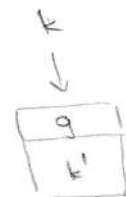
↳ Darf nicht m teilen
 Darf nicht 0 sein

5. Verbesserung nach Brent

$j = (g + h_2(k)) \text{ mod } m$

$j' = (g + h_2(k')) \text{ mod } m$

- 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



1/81

Effizient lösbare 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 Eigensch. e ?

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

FUNKTIONALES PROBLEM: Finde Lösung mit Eigenschaft e !

OPTIMIERUNGS PROBLEM: Finde optimale Lösung

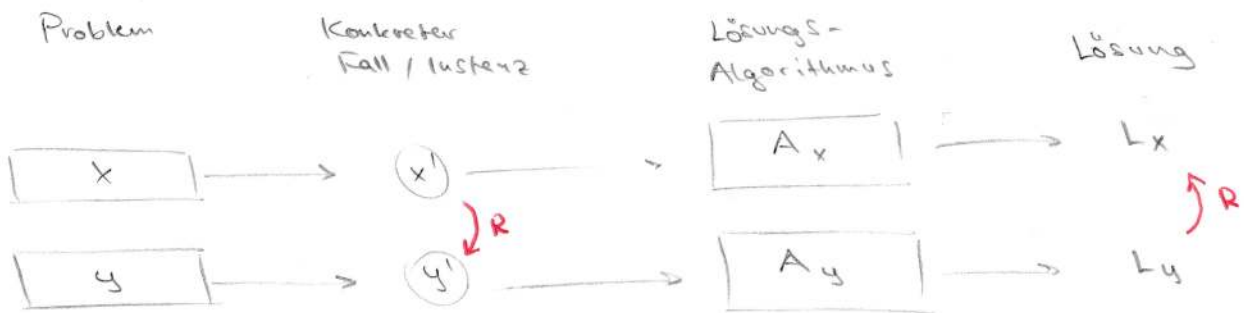
Reduktion

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

Reduktion

Polynomielle Reduktion:

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



(Genau dann wenn $L_y = JA$ ist
 $L_x = JA$)

Wichtig:

R ist selbst ein Reduktions-Algorithmus mit Eingabe x'

in polynomieller Zeit: Verkettung der Laufzeiten

Reduktion $O(x'^a)$

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

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

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

$O(n^c)$

Polynomialzeit-Reduktion

Problemtypen

Gibt es Lösung mit Eigensch. e ?

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

FUNKTIONALES PROBLEM: Finde Lösung mit Eigenschaft e !

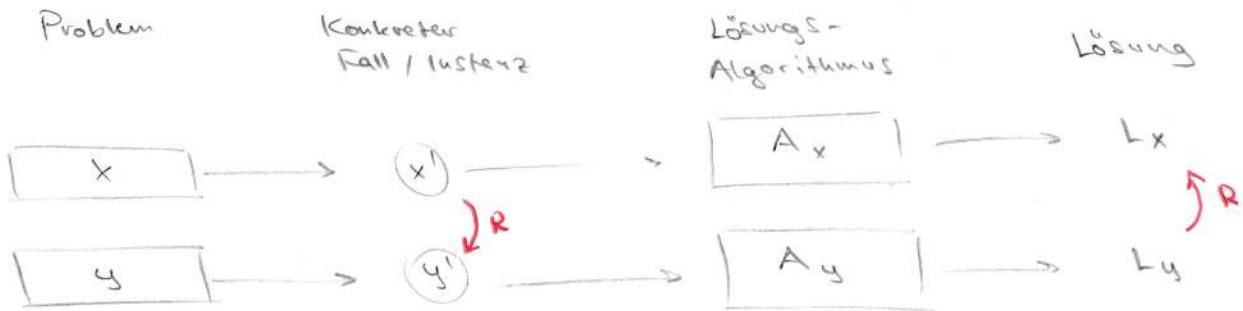
OPTIMIERUNGS PROBLEM: Finde optimale Lösung

Reduktion

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

Polynomialzeit Reduktion:

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



(Genau dann wenn $L_y = JA$ ist
 $L_x = JA$)

Wichtig:

R ist selbst ein Reduktions-Algorithmus mit Eingabe x'

in polynomialer Zeit: Verkettung der Laufzeiten

Reduktion $O(x'^a)$

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

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

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

$$\begin{array}{l} |V| - |IS| = |VC| \\ |V| - |VC| = |IS| \end{array} \quad \left. \vphantom{\begin{array}{l} |V| - |IS| = |VC| \\ |V| - |VC| = |IS| \end{array}} \right\} \text{„Konversionslemme“}$$

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 $\leq k$ Kanten

MST*

MST mit $\leq k$ Kanten

MNB \equiv_p MST*

Konversionslemme

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

Reduktion durch

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

Reduktion des Spezialfalls auf den allgemeinen Fall

SETCOVER

Mengenüberdeckungsproblem

Gesamtmenge U

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

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

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

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

VERTEXCOVER \leq_p SETCOVER

Polynomielle Reduktion: Erzeugung einer Instanz für SetCover

$k = k$

$U = E$ aus $G = (V, E)$

Für jedes $v \in V$

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

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 vorkommen

Reduktion durch Gadget's

SAT - Satisfiability - Problem

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

Klausel: $C_j = (l_1 \vee l_2 \vee l_3)$

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

✓ polynomial

✓ korrekt:

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

weil

1. Komplementäre Variablen verbunden

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

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

2. Nur 1 wahre Variable pro Klausel weil
3 Ecken

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

Zusammenfassung

Einfache Äquivalenz

$\text{INDEPENDENTSET} \equiv_P \text{VERTEXCOVER}$

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

Spezialfall auf allgemeinen Fall

$\text{VERTEXCOVER} \leq_P \text{SETCOVER}$

Gadgets

$3\text{-SAT} \leq_P \text{INDEPENDENTSET}$

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 reducible" 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 Φ 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, Jede Node eine Stadt
- TSP*: Ja/Nein-Problem, existiert Tour mit $\leq k$ Kanten?

$$\text{HAM-CYCLE} \leq_p \text{TSP}^*$$

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

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

Eingabe Instanz x

↓

Nach Mathematischer Definition:

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

ZERTIFIZIERER / VERIFICATION ALGORITHM

Polynomialzeit-Algorithmus $C(x, t)$ welcher 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 \forall Probleme $X \in$ NP : $X \leq_p Y \Rightarrow Y \in$ NP-Schwer

Diese Probleme sind mindestens genauso schwer wie die schwersten NP-Probleme. Wenn man ein NP-schweres Problem effizient lösen 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 \in NP

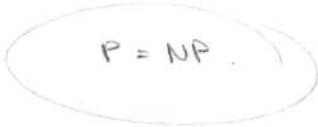
Sind NP-schweren Problemen ähnlich (lassen sich reduzieren) \rightarrow nicht klassifizierbar

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

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

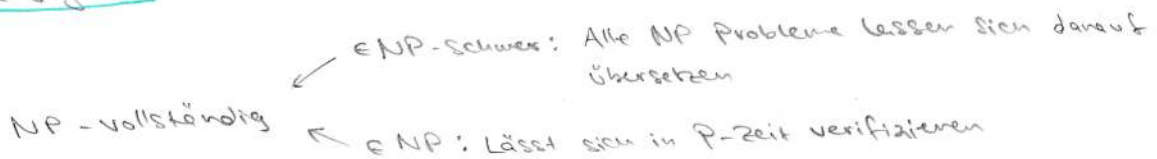


Wenn aber $P = NP$

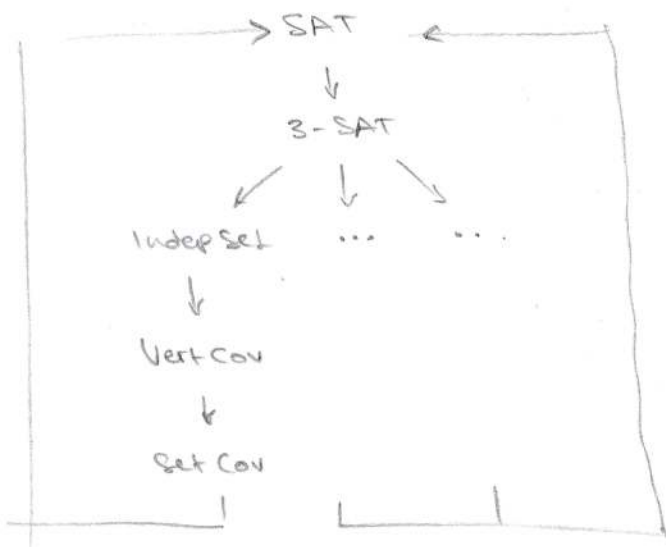


Könnten sich NP-vollständige Probleme lösen lassen, da $NP \subseteq P$ sein würde

NP-Vollständigkeit

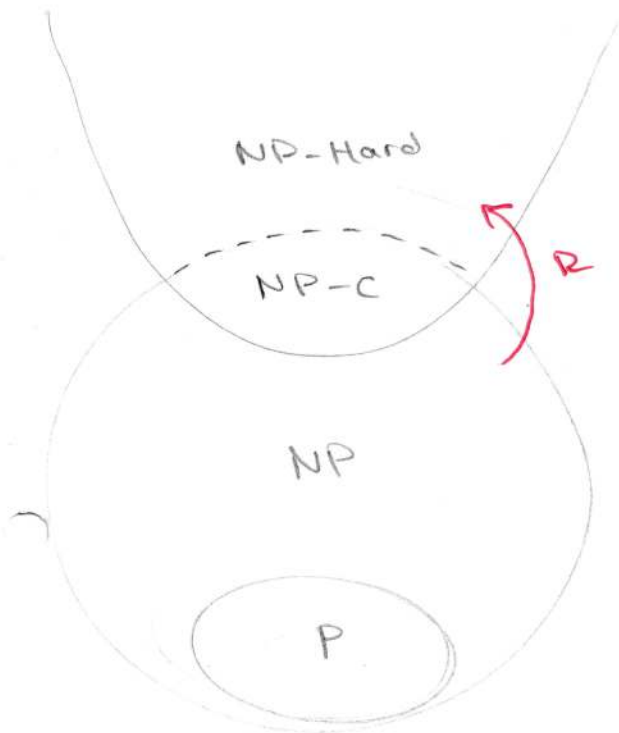


Wenn man ein NP-vollständiges-Problem auf ein NP-Problem übersetzt (reduziert), dann können alle $NP \subseteq$ Probleme nur NP oder leichter sein, weil sie sich alle gegenseitig ineinander übersetzen lassen.

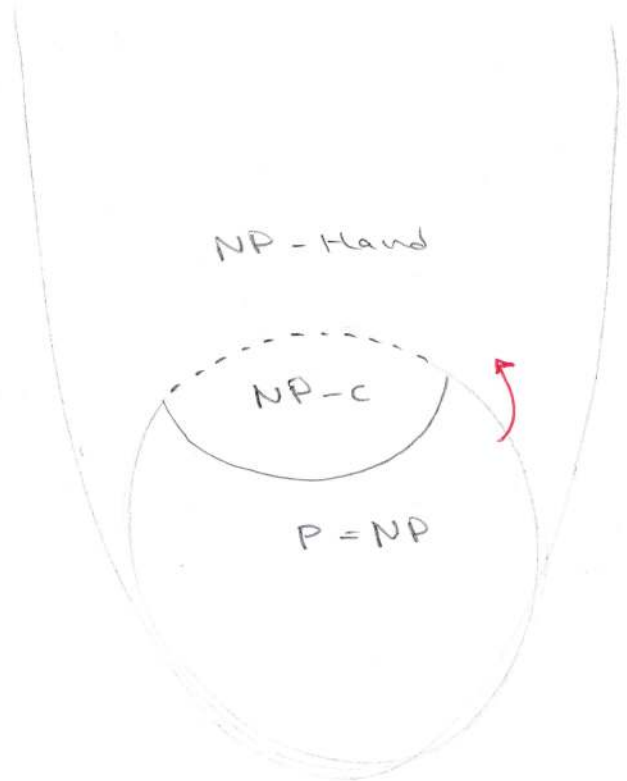


← Alle NP Probleme lassen sich darauf reduzieren und es ist in P verifizierbar

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 $NP-C = NP$ (bewiesenermaßen unmöglich)

3-SAT \leq_p 3-COLOR

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

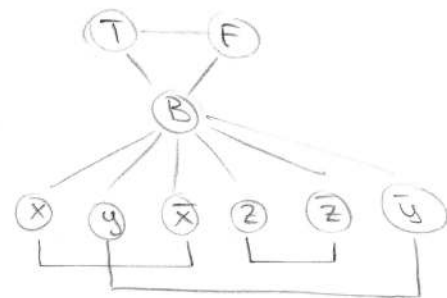
3 SAT:

k 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 Gadget konstruiert mit 6 Knoten:



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

(siehe Rückseite)

Behauptung: Graph ist 3-färbbar wenn Φ erfüllbar

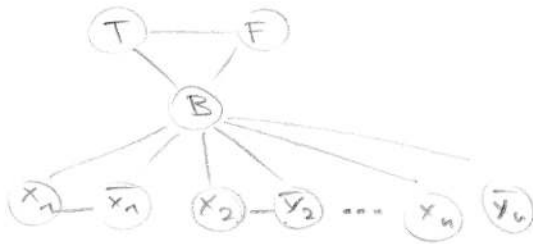
Beweis:

T = grün

F = rot

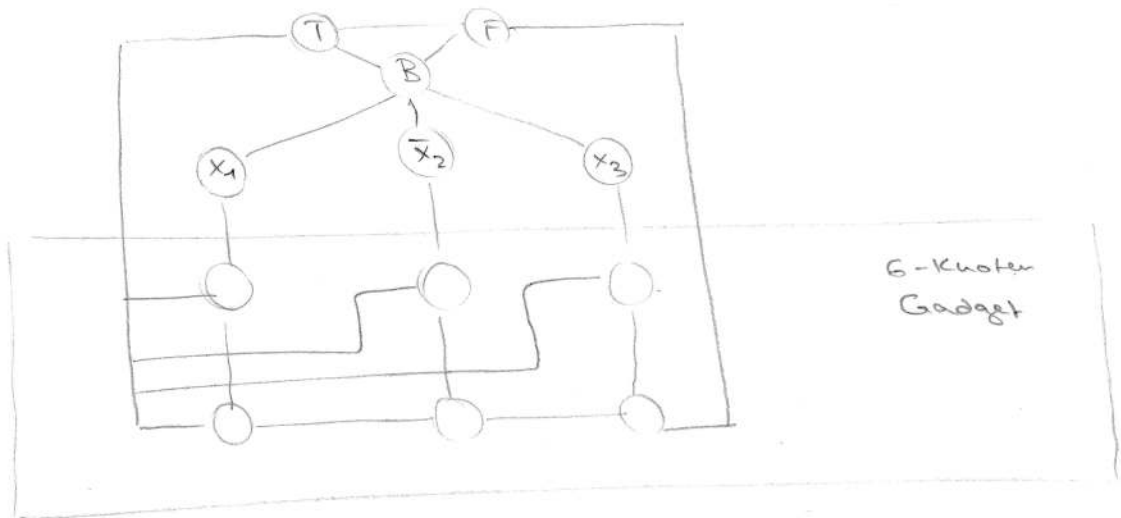
B = blau

→ Alle Variablen $\underbrace{\text{grün}}_{\text{True}}$ oder $\underbrace{\text{rot}}_{\text{False}}$



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

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



✓ 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 2-1 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:

$n = 1000$
 $k = 10$

\rightarrow Brute force

\rightarrow Neuer Algorithmus:

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

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

(unmöglich)

(möglich wenn k klein ist)

Lemma

$G = (V, E)$ $(u) - (v)$

Knoten (u) und (v) sind adjazent und G hat Vertex Cover mit k Knoten

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

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

Lemma: Obere Schranke

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

$$|V| = n$$

$$|V| = n$$

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

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



Beweis

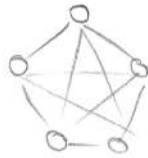
Jeder Knoten kann max $\deg(v) = n - 1$ haben und $n - 1$ Kanten überdecken.

Sonst gibt es Mehrfachkanten

Obere Schranke

Ein vollständiger Graph hat eine Maximale Kantenzahl

K_5 :



$$|V| = 5$$

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

wegen Handshake Lemma

$$|V| = 5$$

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 inzidenten Kanten löschen

\leftarrow boolsche Aussage

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

$O(2^k kn)$

binärer Verifikation Baum,

pro Ebene 2^{i-1} Einträge

NP-vollständige Probleme lösen:

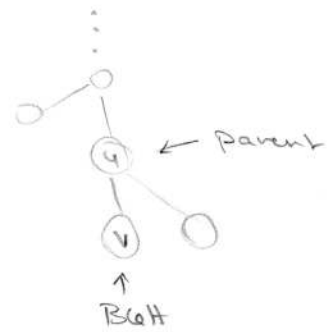
Spezialfall: Eingabegraph ist ein Baum (acyclisch)

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 OU) \leftarrow LRN, noch effizienter)

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

$S \leftarrow \emptyset$

while T hat noch 1 Kante

$v \leftarrow$ beliebiges Blatt

$u \leftarrow v.$ parent

$S \leftarrow S \cup \{v\}$

lösche u, v sowie alle inzidenten 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: $v \notin \text{MaxIS}$

$u \in \text{MaxIS}$

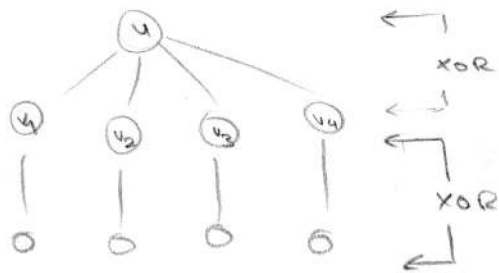
fertig

Conclusion:

Es gibt ein MaxIS mit allen Blättern

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

$w_v > 0$



Algorithmus muss entscheiden:

Algorithmus mit dynamischer Programmierung

$$OPT_{in}(u) = w_u + \sum_{\text{Kinder}} OPT_{out}(v)$$

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

Case 1: u nehmen
alle Kinder nicht nehmen

Case 2: u nicht nehmen
für alle Kinder rekursiv entscheiden
ob sie genommen werden sollten oder nicht
und summieren

$O(n)$

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

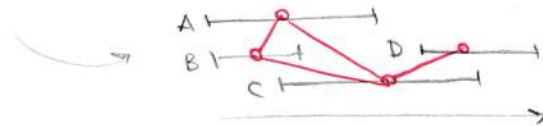
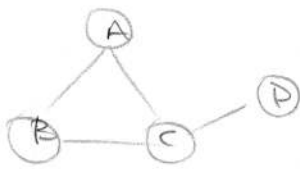
Kann man Graphen-Knoten mit 3 verschiedenen Farben färben,
so dass sich nie 2 gleiche Farben "berühren" (adjazent sind)

Intervallmenge

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

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

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



"Schnittgraph"

Wichtig:

1-Color-Problem = $\begin{cases} \text{Independent Set} \\ \text{Interval Scheduling (Greedy-earliest deadline first)} \end{cases}$

Definition:

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

= # Farben die benötigt
werden

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

colmax = 0 (max # Farben benötigt)

Queue $Q = \emptyset$

foreach $a \in L$

if a ist Startpunkt

if $Q = \emptyset$

färbe l mit colmax

colmax++

if $Q \neq \emptyset$

färbe l mit $Q.pop$

if $a \neq$ Startpunkt

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

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 Hausverstand“
- Löse Problem in Polynomialzeit
→ Algorithmen mit exponentieller Laufzeit
- Löse beliebige Instanzen des Problems
→ Identifiziere 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 $O(2^n)$

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

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

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

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

} $x_i = 1$ Rekursion

$x_i = 0$

} $x_i = 0$ Iteration in Schleife

Rucksack-Problem: Verbesserung durch Bound (aries)

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

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

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

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

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

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

if $w_{max} < U'$

} Möglicher Abbruch $w_{max} = U'$

$x_i = 1$

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

$x_i = 0$

Verbesserung der unteren Schranke

$L^1 \neq w_{curr}$ sondern Greedy

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

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

└──────────┘
Greedy bestimmen

Verbesserung der oberen Schranke

u^1

optimale best-case Lösung ist erreicht wenn sie mit der neu definierten

$L^1 = w_{curr}$ übereinstimmt: $L^1 = u^1$

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

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

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

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

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

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

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

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

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

z : # belegten Stellen

w_{curr} : $\sum w$

g_{curr} : $\sum g$

\vec{x} : binärer Lösungsvektor

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

Ablauf:

for-Schleife bestimmt Position

↓
 $(\cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$ $z=0 \rightarrow i=z+1$
 $x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5$

Entweder ablehnen

$x_0 = 0$

nächste Schleifen-Iter.

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

Oder mitnehmen

$x_0 = 1$

nächste Rekursionsebene

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

Verbesserung durch boundaries für branches

Man überprüft ob tiefere Rekursion sinnvoll ist

Obere Schranke U' :

$$U' = \underbrace{w_{curr}}_{\text{aktuelle Summe}} + \underbrace{(G - g_{curr})}_{\# \text{ freie Stellen}} \cdot \underbrace{\frac{w_i}{g_i}}_{\substack{\text{Leistung} \\ \text{Preis}}} \quad \text{Preis-Leistung für nächsten Gegenstand}$$

Einsetz in Enum:

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

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

If ($U' > w_{max}$)

$$x_i = 1$$

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

$$x_i = 0$$



Weil absteigend
Sortiert können
folgende Elemente
nur kleiner sein

Nur wenn Verbesserungspotential besteht

Beispiel

$$w_{max} = 11$$

$$w_{curr} = 6$$

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

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

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

Deshalb:

$$\frac{w_i}{g_i} = \frac{8}{4} = 2$$



Wir stellen uns vor

$$w_i = 2$$

$$g_i = 1$$

womit obere Schranke

$$6 + 2 \cdot 2 = 10$$

und

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

Branch 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 / Wurst)

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
↓
Branch-and-Bound-Max (I)
 $L \leftarrow -\infty$ oder „initiale Lösung“ mit Greedy
 $\Pi \leftarrow \{I\}$

while $\exists I' \in \Pi$

Entferne I' aus Π

Berechne U' mit „Dualheuristik“

if $U' > L$

Berechne L' heuristisch

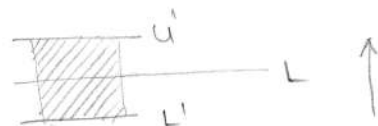
if $L' > L$
 $L = L'$

if $U' > L$

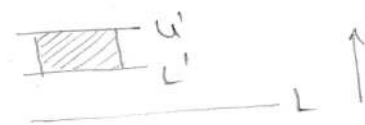
Partitioniere I' und speichere
in Π

return beste Lösung mit Wert L

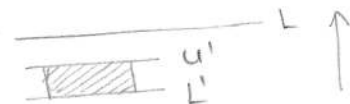
- Es besteht noch verbesser.- Potential



Es muss L geupdated werden



- Es besteht kein Verbesserungspot.



Branch-and-Bound verallgemeinert für Minimierungsproblem

Boundaries

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

Branch-and-Bound-Min(1)

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

$II \leftarrow \{1\}$

While $\exists I' \in II$

entferne I' aus II

Berechne L' mit „Dualheuristik“

if $L' < U$

Berechne U'

if U' ist eine gültige Lösung

if $U' < U$

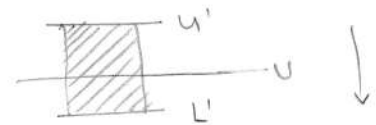
$U = U'$

if $L' < U$

Partitioniere / Berechne

return beste gefundene Lösung mit Wert U

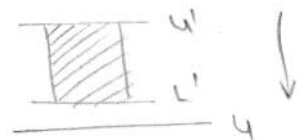
Verbesserung möglich: $L' < U$



update notwendig



Verbesserung unmöglich



Auswahlstrategien

Welche Node aus dem Baum sollte als nächstes bearbeitet werden?

Best first

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

- intuitiver
- Schneller aber ungenauer

Depth first

Nodes werden einfach wie bei DFS gewählt

- langsamer
- bessere Approximation, genauer
- vollständige Lösung

Branch and Bound: Minimales Vertex Cover

Lösungsmenge $C = \emptyset$

Min Vertex Cover (G, C)

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

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

While $\exists I' \in \Pi$

Berechne L' mit "Matching Heuristik"

if $L' < U$

Berechne U' mit "Greedy Heuristik"

if $U' < U$

$U = U'$

if $L' < U$

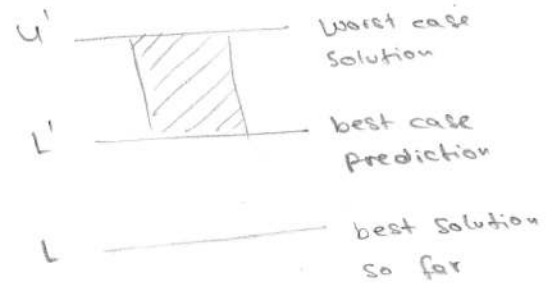
(Start Branching)

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

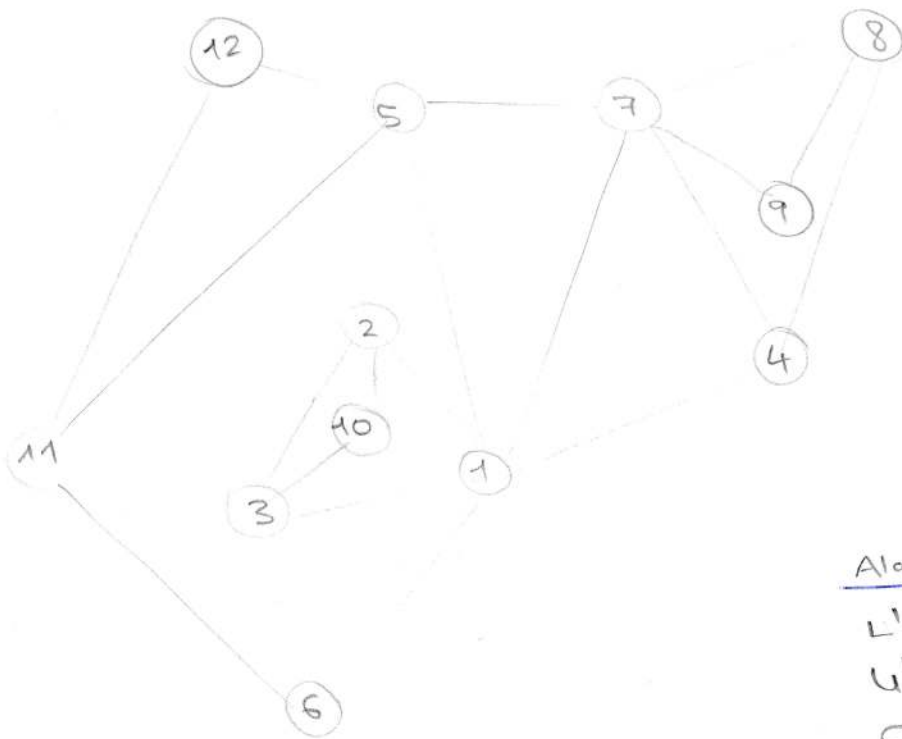
$I_1 = (G' \setminus \{u_{max}\}, C \cup \{u_{max}\})$

$I_2 = (G' \setminus \{u_{max}\}, C \cup N(u_{max}))$

return Lösung mit Wert U



Aufgabe 3) Minimales Vertex-Cover



Algorithmus:

- L' : best case: Greedy
- U' : worst case: Matching
- C : Lösungsmenge

Knoten sortiert nach Grad:

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

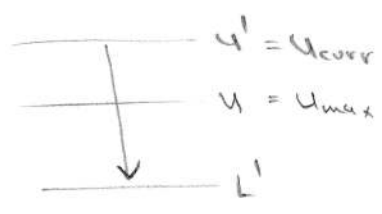


Knoten	Knotengrad	
1	6	✓
2	3	✓
3	3	✓
4	3	✓
5	4	✓
6	2	✓
7	5	✓
8	3	✓
9	2	✓
10	2	✓
11	3	✓
12	2	✓

Knotenmenge

Mit n Vertex Cover - Branch And Bound (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 wert U .

Branching:

I_1'' aufnehmen in Lösungsmenge

I_2'' ablehnen aus Lösungsmenge, aber Nachbarn aufnehmen

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

G
 $C = \{\}$ $|C| = 0$
 $L' = 6$ (Max Matching)
 $U' = |\{1, 5, 7, 8, 2, 10, 1\}| + 0 = 7$ (Greedy)

$u = |V| - 1 = 11$
 $7 < 11$
 $u = 7$

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

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

$G = G \setminus \{1\}$
 $C = \{1\}$ $|C| = 1$
 $L' = 5 + 1 = 6$
 $U' = |\{1, 5, 7, 8, 3, 10\}| + 1 = 7$

$G = G \setminus \{1, 2, 3, 4, 5, 6, 7\}$
 $C = \{2, 3, 4, 5, 6, 7\}$ $|C| = 6$
 $L' = 1 + 6 = 7$
 $U' = |\{9, 12\}| + 6 = 8$

$L' = U$
ABBRUCH

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

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

$G = G \setminus \{1, 7\}$
 $C = \{1, 7\}$ $|C| = 2$
 $L' = 4 + 2 = 6$
 $U' = |\{11, 10, 8, 3, 5\}| + 2 = 7$

$G = G \setminus \{1, 7, 5, 4, 8, 9\}$
 $C = \{1, 5, 4, 8, 9\}$ $|C| = 5$
 $L' = 2 + 5 = 7$
 $U' = |\{10, 3, 11\}| + 5 = 8$

$L' = U$
ABBRUCH

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

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

$G = G \setminus \{1, 7, 11\}$
 $C = \{1, 7, 11\}$ $|C| = 3$
 $L' = 3 + 3 = 6$
 $U' = |\{8, 12, 2, 10\}| + 3 = 7$

$G = G \setminus \{1, 7, 11, 12, 5, 6\}$
 $C = \{1, 7, 12, 5, 6\}$ $|C| = 5$
 $L' = 2 + 5 = 7$
 $U' = |\{8, 2, 10\}| + 5 = 8$

$L' = U$
ABBRUCH

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

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

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

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

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

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

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

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

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

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

$L' = U$
ABBRUCH

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

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

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

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

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

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

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

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

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

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

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

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

Alle Lösungsmengen mit Kardinalität von 7:

- $\{11, 5, 7, 8, 2, 10, 1\}$
- $\{11, 5, 7, 8, 3, 10, 1\}$
- ~~$\{11, 5, 7, 8, 3, 10, 1\}$~~
- ~~$\{11, 5, 7, 8, 3, 10, 1\}$~~
- $\{11, 12, 7, 1, 8, 2, 10\}$
- ~~$\{1, 7, 11, 8, 12, 10, 2\}$~~
- ~~$\{1, 7, 11, 8, 10, 12, 2\}$~~
- ~~$\{1, 7, 11, 8, 10, 12, 2\}$~~
- $\{1, 7, 8, 9, 4, 11, 12\}$

die obere Schwanke zuletzt gesetzt
von:

ident

ident

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.

Zwischenspeichern

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.

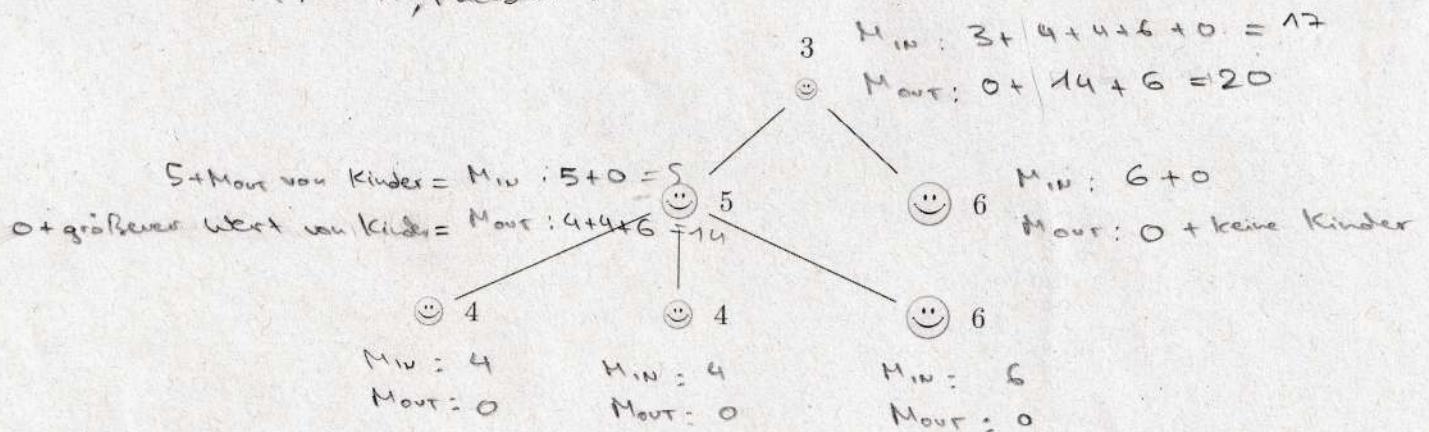
Wesentlicher Aspekt: Speicherung (memoization) von Ergebnissen für Subprobleme zur Wiederverwendung.

3/93

Beispiel: Weighted Independent Set auf Bäumen

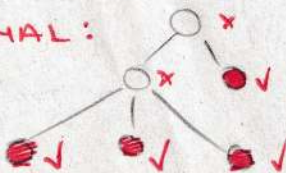
Bestimmt mit M_{in} (Node \checkmark , Nachbarn \times)

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



4/93

OPTIMAL:



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]

```

Iterativ - „bottom-up“

```

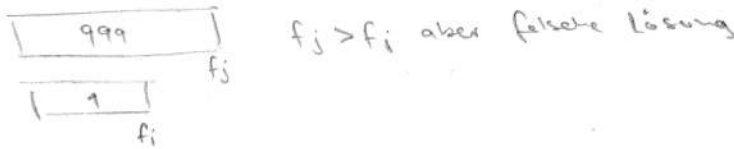
Fib(n)
F[1] = 1
F[2] = 1
for i = 3 bis n
  F[i] = F[i-1] + F[i-2]
return F[n]

```

Beide mit Laufzeit $O(n)$

Gewichtetes Intervall Scheduling

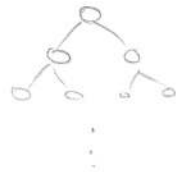
Greedy scheitert wenn gewichtet:



Brute force

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

$\underbrace{\hspace{10em}}_{OPT_{in}}$
 $\underbrace{\hspace{10em}}_{OPT_{out}}$



Rekursiv / „top down“

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

Berechne $P(\forall n)$

```

OPT(j)
if M[j] = leer
  M[j] = max { w_j + OPT(p(j)), OPT(j-1) }
return M[j]

```

Iterativ / „bottom-up“

```

ME[0] = 0
for j = 1 bis n
  ME[j] = max (w_j + ME[p(j)], ME[j-1])

```

Laufzeit für Top-down

Sortierung $O(n \log n)$

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

Aufruf von $OPT(u)$

Liefert entweder Lösung in $O(1)$

oder berechnet neuen Eintrag durch Verzweigung $\begin{cases} OPT(p(j)) \\ OPT(j-1) \end{cases}$

Laufzeit für Bottom-up

$O(n)$

Backtracking

Backtrack(j)

if $j=0$

keine Ausgabe

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

$\underbrace{\hspace{1.5cm}}_{OPT_{in}} \quad \underbrace{\hspace{1.5cm}}_{OPT_{out}}$

gib j aus

Backtrack(p(j))

$O(n)$ insgesamt

else

Backtrack(j-1)

↓
Fortschritts-Maß "y"
1

Am Anfang 0,

Durch jede Baumebene $y++$

↓

$O(n)$ weil

$0 \leq y \leq n$

Segmented Least Squares

Allgemeines Problem mit 1 Gerade: least squares

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

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

$\underbrace{\hspace{10em}}_{\text{Wirklichkeit}} \quad \underbrace{\hspace{10em}}_{= y \text{ (Guess)}}$



(lässt sich mit Formel berechnen)

Mit mehreren Geraden ist Genauigkeit und Sparsamkeit wichtig
 min Error min # Geraden

Wir definieren ein Verhältnis:

- $E = \text{Error}^2$
- $L = \# \text{ Geraden}$
- $c = \text{Konstante} > 0$

„Trade off function“

$\rightarrow \underline{E + cL = \text{konstante}}$

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

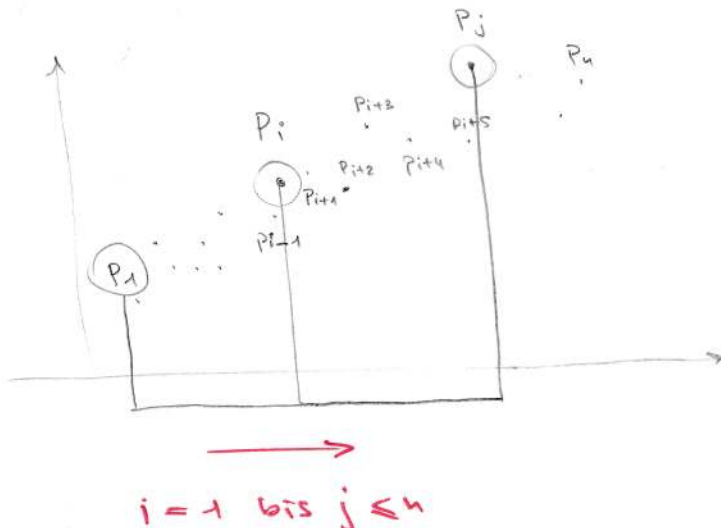
Algorithmus

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

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

wobei $j \leq n$

(löst mit 1 Gerade)



Algorithmus Berechnet:

Minimum der folgenden Gleich.:

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

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

$$\vdots$$

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

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 $e(i, j)$ $O(n)$] $O(n^2)$

(Least Squares $(i, j) = e(i, j)$
Löse Problem mit einer Linie mit min Err

for $j = 1$ bis n

$$M[j] = \min_{1 \leq i \leq j} (M[i-1] + e(i, j) + c) \quad] \quad O(n^2)$$

return $M[n]$

Insgesamt:

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

Rucksackproblem

Greedy: keine optimale Lösung

Branch 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 1 bis i

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

Algorithmus iterativ - „bottom up“

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

for $g=0$ bis G
 $M[0, g] = 0$] $O(G)$

for $i=1$ bis n

for $g=0$ bis G

if $g_i > g$

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

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

] $O(n, G)$

return $M[n, G]$

Laufzeit & Speicher: $O(nG)$

Backtracking (M-Array)

$A = \emptyset$

while $n > 0$ and $G > 0$

← OPT_{in} if $M[i, G] \neq M[i-1, G]$

$A = A \cup \{i\}$

$G = G - g_i$

$i = i - 1$

return A

Problem: $O(nG)$ \rightarrow „Pseudo Polynomiell“

Polynomiell in n

G ist exponentiell in Länge weil Zahlen binär kodiert werden

Es gilt:

Rucksackproblem kann nicht in 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 Kantengewichten, 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 (Dijkstra nicht anwendbar)

Problem:

Wenn es negative Kantengewichte gibt, gibt 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



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

Es gilt:

$$OPT(t) = 0$$

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

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

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

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

Wichtig:

$OPT(u-1, v) = OPT(u)$ wenn es keine Kreise gibt, der worst case: $0-0-0-0 \dots 0$
u Knoten
u-1 Kanten

Iterative Bottom-up Implementierung

Shortest-Path (G, s, t)

M [Kanten #, Knoten]

foreach $v \in V$

$M[0, v] = \infty$

$M[0, s] = 0$

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 $(v, w) \in E$

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

return $M[n-1, s]$

Laufzeit:

Speicherplatz:

$u \times u$ - Array

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

For Schleife: $O(u-1)$

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

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

Laufzeit:

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

Durch effiziente Implementierung $O(u \cdot m)$ auch möglich

Negative Kreise Erkennen

Durch Bellman-Ford-Algorithmus

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

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

$M[s] = 0$

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

foreach $(v, w) \in E$

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

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

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

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

break

return $M[s]$

$O(n \cdot 2m) =$

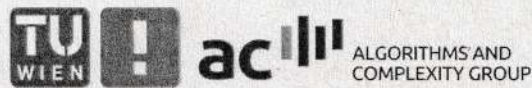
$O(n \cdot m)$

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

Branch-and-Bound \rightarrow weiterhin Exponentiell
Dynamische Programmierung \rightarrow nur Sonderfälle

Approximation(algorithmen): 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 \downarrow
Nicht bei
Heuristischen
Verfahren

2 / 58

Approximationsalgorithmen

Annäherung mit Gütegarantie in Polynomialzeit

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

Minimierung $\epsilon \geq 1$

Maximierung $0 \leq \epsilon \leq 1$

VERTEX COVER

MINIMALES-VERTEX COVER ist NP-Schwer

2-Approx-Vertex-Cover (G)

$C = \emptyset$

while $E \neq \emptyset$

wähle random Kante $(u, v) \in E$

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

entferne alle inzidenten Kanten von u und v

return C

Greedy-Approx-Vertex-Cover (G)

$C = \emptyset$

while $E \neq \emptyset$

wähle Node mit max deg

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

Entferne alle inzidenten Kanten zu Node

return C

$\epsilon = \log n$

Beweis kompliziert

$O(u+m) \rightarrow$ ist ein Matching
 $\epsilon = 2$



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
genereller Knoten Anzahl:

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

MST-Heuristik für das symmetrische Traveling Salesman Problem

$$c_{ij} = c_{ji}$$

Wert von beiden Seiten 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 bilde Graph (V, E')
3. Bestimme Euler-Tour E in (V, E')
(Das bedeutet jede Kante nur 1X)
4. Gebe Euler-Tour eine Orientierung

Wähle Startpunkt $s \in V$
marked[s] = true
 $p = s$

$p =$ letzter besuchter
Knoten

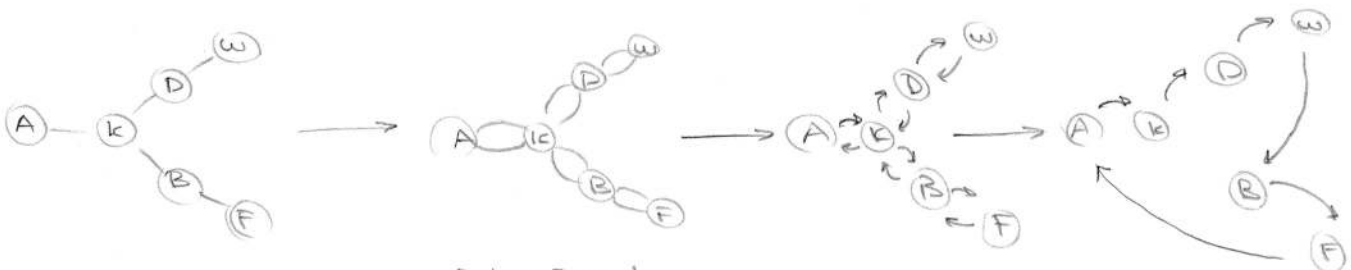
5. wenn bereits alle Knoten markiert

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

6. sonst traversiere von p entlang der Tour bis q wobei marked[q] = 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 $O(u^2)$

Aufwändigster Teil: MST-Bestimmung mit Prim in $O(u^2)$

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

Bestimmung eines möglichen Euler-Tours:

Algorithmus von Hierholzer

1. Wähle beliebigen $v_0 \in V$
konstruiere geschlossenen Pfad von v_0 namens Z , in dem 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 \nexists Gütegarantie $\epsilon \leq 2$ weil

HAM-CYCLE \leq_p SYMMETRICAL-TSP

Transformation:

$$G(V, E) \rightarrow G(V, E', C)$$

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 $n = |V|$ Kanten

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

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

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

$C_{V'}$ MST-Heuristik, verdoppelte Kanten

C_{MST} Minimaler Spannbau

C_{OPT} Optimale Lösung

$$C_{ST} \leq C_{V'} = 2 \cdot C_{MST} \leq 2 C_{OPT}$$

wegen
Dreiecks-Ungleichung

weil MST kein Zyklus ist muss es kleiner sein

Lastverteilung - „Load Balancing“

Scheduling Problem mit mehreren Maschinen zugleich

m Maschinen
 n Jobs $\{j_1, \dots, j_n\}$ werden mit J_i der Maschine $1 \leq i \leq m$ zugeteilt
 t_j Bearbeitungszeit
 L_i Last von Maschine $i = \sum_{j \in J_i} t_j$ 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

$i = \operatorname{argmin}_{k=1, \dots, m} L_k$

$O(\log m)$ mit Heap
] Laufzeit
 $O(n \log m)$

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

$L_i = J_1 \dots J_m$

Über Algorithmus:

- Greedy

- $\epsilon = 2$

Beweis:

Angenommen L ist makespan von Algo

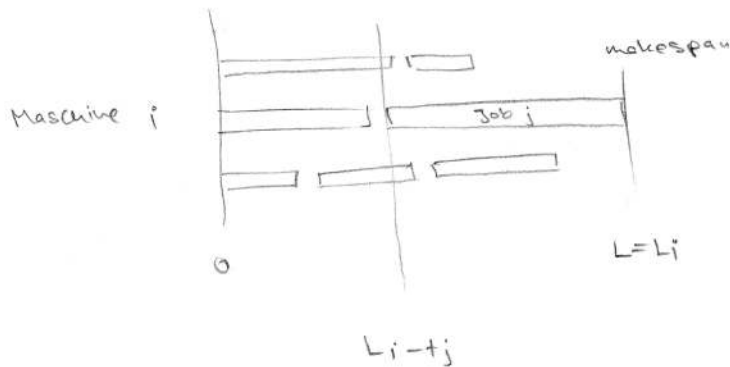
L^* ist optimale Lösung

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

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

Lemme: $\epsilon = 2$ 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 $L_i - t_j$ war die kleinste unter allen Maschinen
 $L_i - t_j \leq L_k$ für alle $k \in [1; m]$

Bzw.

$$\begin{array}{l}
 L_i - t_j \leq L_1 \\
 L_i - t_j \leq L_2 \\
 \vdots \\
 L_i - t_j \leq L_i \\
 L_i - t_j \leq L_{i+1} \\
 \vdots \\
 L_i - t_j \leq L_m
 \end{array}
 \left. \begin{array}{l}
 \text{Last von Maschine 1} \\
 \text{(alle vor Maschine i)} \\
 \\
 \text{(alle nach Maschine i)}
 \end{array} \right\}$$

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

$$L_i - t_j \leq \frac{1}{m} \sum_{k=1}^m L_k$$

Summe aller Gesamtlasten durch m

$$= \frac{1}{m} \sum_{j=1}^n t_j$$

Summe aller Job-Zeiten durch m

$$\leq L^*$$

Daraus folgt:

$$L_i - t_j \leq \frac{1}{m} \sum_{j=1}^n t_j \leq L^*$$

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

$$\rightarrow L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{\max t_j}_{\leq L^*} \leq 2 L^*$$

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:

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

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

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

Greedy Algorithmus

Greedy-Center-Selection (k, S)

$C =$ beliebiges $s_1 \in S$

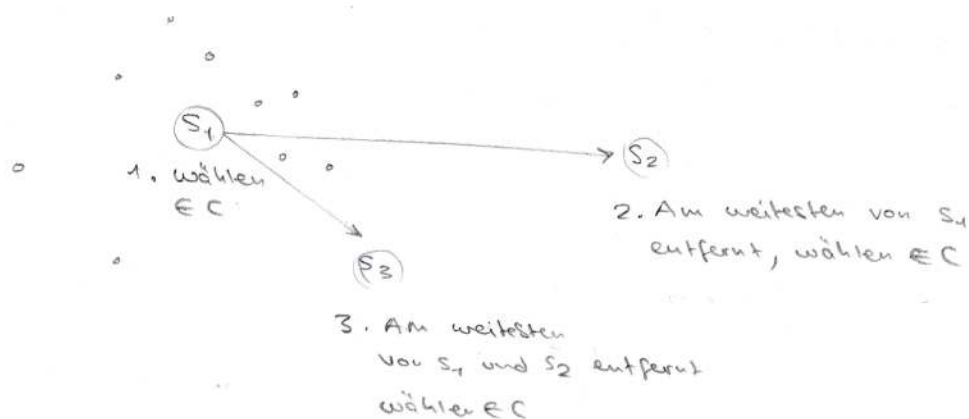
Wiederhole $k-1$ Mal:

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

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

} wähle s_i mit größtem Abstand zu allen C

return C



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

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

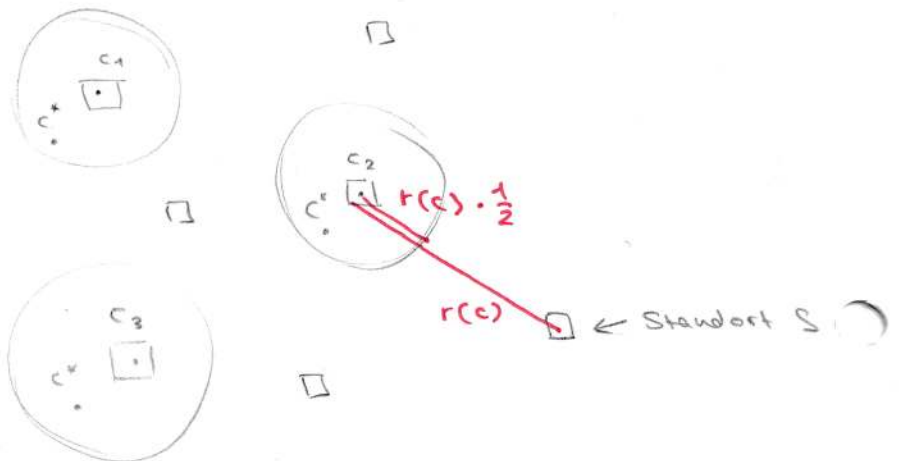
Beweis durch Widerspruch:

- 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^*

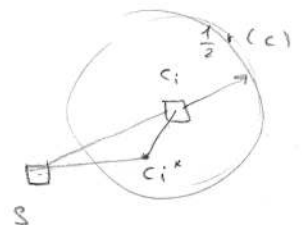


$$\underbrace{\text{dist}(S, c^*)}_{C^* \text{ muss zu } S \text{ näher sein als } C} \leq \underbrace{\text{dist}(S, c_i)}_{\leq r(C)} \leq \underbrace{\text{dist}(S, c_i^*)}_{\leq r(C^*)} + \underbrace{\text{dist}(c_i^*, c_i)}_{\leq r(C^*)} \leq 2r(C^*)$$

C^* muss zu S näher sein
 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 $r(C) \leq 2r(C^*)$)

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

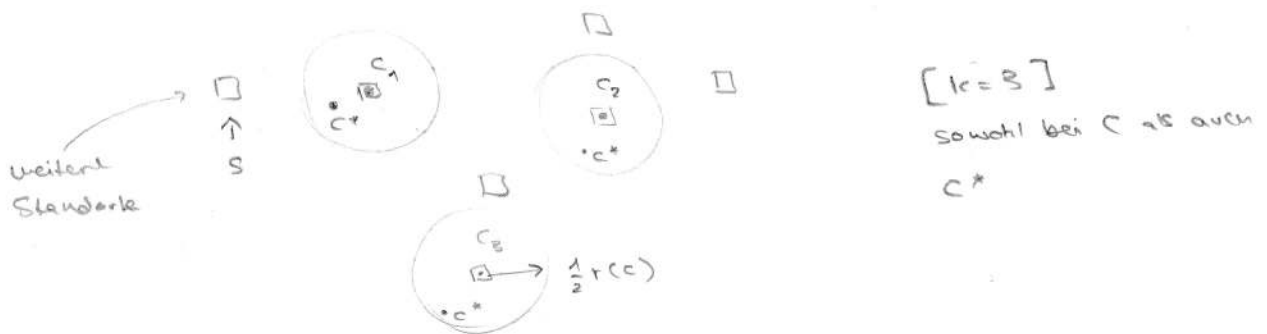
- Man beachte die k Mittelpunkte in C die zugleich Standorte sind,
 mit einem halben Radius: $\frac{1}{2}r(C)$

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



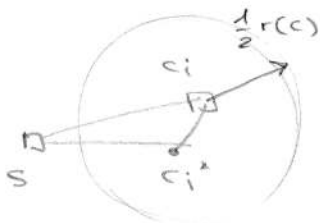
- Man nehme Standort außerhalb von $\frac{1}{2}r(C)$ von c_i
 und nenne sie s .

$$\text{dist}(s, c^*) \leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$$

Mittelpunkt c^*
 muss näher sein als c

Dreiecksungleichung

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



$$\text{Wenn } r(c) \leq r(c^*) \cdot 2$$

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

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

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(algorithmen)

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

Poly Zeit ohne Gütekriterium, intuitiv

1. Konstruktionsverfahren

Ungenaue 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

Bilde konvexe Hülle mit V 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 Knoten, sodass noch gültige Lösung $O(n)$

Problem: steckt im lokalen Optimum

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

Problem: aufwändiger $O(n^3)$

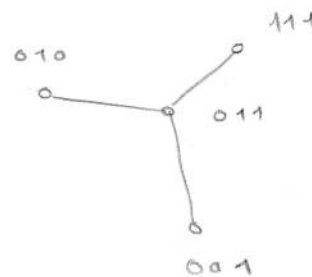
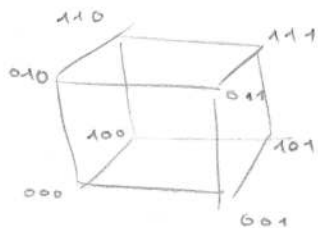
Wege um globales Optimum zu erreichen

- 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

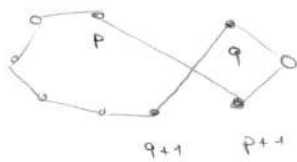
Abbruchkriterium

- 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

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

Verschieben eines Knotens

Liu-Kernighan-Heuristik

Heute die beste Variante

Maximaler Schnitt

MAXCUT

Ist NP-Vollständig

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

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

Bestimmung eines lokalen Optimums mit "Flip-Nachbarschaft":

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

Bedingung für lokales Optimum:

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

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

Aufsummiert:

(wegen Handshake alle Kanten doppelt betrachtet)

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

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

Daraus folgt:

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

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

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



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

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

3. Metaheuristiken

Erweiterung der lokalen Suche

„problem-unabhängig“

Simulated Annealing SA

Um lokale Optima zu vermeiden

Simuliert Abkühlung um stabile Kristallstruktur zu erreichen:

Akzeptanz von schlechten Nachbarn 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 WSK

viel schlechtere Lösung \rightarrow niedrigere WSK

Abkühlungsplan

Wie sich T nach jeder Iteration ändern soll

(kann sich dem Fortschritt anpassen)

Zusammenfassung:

- Einfache Implementierung
- Braucht Parameter tuning

Anwendungsbeispiel:

Graph-Bi-Partitionierung
(Quasi MINCUT)

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

Boolscher Vektor:

$$n = |V|$$

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

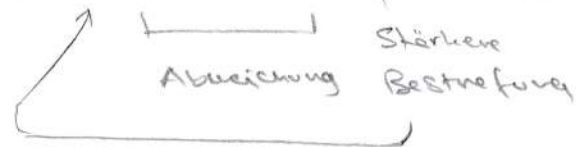
$$x_i = 1 \rightarrow v_i \in B$$

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

Anwendung vor Simulated Annealing:

- Zufällige Anfangslösung
- „Geometrisches Abkühlen“
- Ungültige Lösungen erlauben, 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 Speicher verbrauchen

Algorithmus:

Tabu-Suche ()

$x_{best} = x$ = Ausgangslösung

TabuListe $\leftarrow \{x\}$

while (...)

$x' \leftarrow$ Teilmenge von $N(x)$ abhängig von TabuListe] Nachbarschaft bestimmen

$x' \leftarrow$ beste Lösung aus $N(x)$

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

Lösche Elemente aus TabuListe die älter als t_L Iterationen sind

$x \leftarrow x'$

if $x > x_{best}$

$x_{best} = x$

Wichtigster

Parameter:

„TabuListenlänge“

Zu kurz: Zyklen

Zu lang: stecken bleiben

Man bestimmt auch

Tabuattribute die verboten sind

Nach

Aspiration skriterien werden

manchmal Tabu-Lösungen

trazieren erlaubt wenn

so gut genug sind

Evolutionäre Algorithmen

→ Vorteil: Parallelisierung möglich

1. Bestimmung der Population

Ausgangslösungen

2. Selektion

Bessere Lösungen → Höhere Selektionswahrscheinlichkeit

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

wird gesteuert über
lineare Funktion

$$g(x_i) = \boxed{a} f(x_i) + \boxed{b}$$

„Selektionsdruck“

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

Zu niedrig:
Gleichverteilung

Zu hoch:
Tyranie

3. Rekombination

Verberbung

4. Mutation

Alternative:

Tournament-Selektion

(keine Skalierung notwendig)

Selektionsdruck = Gruppengröße