Entwurf und Analyse von Algorithmen

Vorlesung 4 Stunden
WS 2000/01

J. Avenhaus
Universität Kaiserslautern

Inhaltsverzeichnis

1 Einleitung 2
1.1 Motivation und Bezeichnungen 2
1.2 Das Maschinenmodell / Beschreibung und Komplexität von Algorithmen 13
1.3 Techniken für Zeitabschätzungen 19
1.4 Effizienz-Steigerung: Ein Beispiel 31
1.5 Probabilistische Algorithmen 37
1.6 Nicht-deterministische Algorithmen 43
2 Strategie Teilen und Beherrschen
Probleme: Suchen und Sortieren 51
2.1 Binäre Suche 53
2.2 Sortieren 55
2.2.1 Sortieren durch Auswahl: SelectionSort 56
2.2.2 Sortieren durch Mischen: MergeSort 59
2.2.3 Sortieren durch Zerlegen: Quicksort 65
2.2.4 Sortieren mit Bäumen: HeapSort 74
2.2.5 Sortieren mit Hashing: BucketSort 82
2.3 Bestimmung des $\mathbf{k}$-kleinsten Elements 83
2.4 Untere Schranken für Such- und Sortierprobleme . . . 87

2.4.1 Technik der Entscheidungsbäume 88
2.4.2 Untere Schranken für das k-kleinste Element . 92
3 Strategie Dynamisches Programmieren 98
3.1 Multiplikation mehrerer Matrizen 99
3.2 Kürzeste Wege in Graphen 106
3.3 Optimale binäre Suchbäume 109
3.4 Das Problem des Handlungsreisenden 114
3.5 Das 0/1-Rucksackproblem 122
3.6 Effizienz-Verbesserungen 130
4 Die gierige Methode (The greedy method) 132
4.1 Optimale Speicherung auf Bändern 133
4.2 Das Bruchteil-Rucksackproblem 135
4.3 Bäume mit minimaler gewichteter Weglänge 141
4.4 Matroide 152
4.5 Auftragsplanung mit Schlußterminen 156
4.6 Aufspannende Bäume in Graphen 160
4.7 Kürzeste Wege in Graphen 163
4.8 Der Union-Find-Algorithmus 169
5 String Matching 180
5.1 Fingerabdrücke: Das Rabin-Karp-Verfahren 182
5.2 String Matching mit endlichen Automaten 186

5.3 Das Knuth-Morris-Pratt-Verfahren 189
5.4 Suche nach mehreren Mustern 203
5.5 Das Boyer-Moore-Verfahren 208
6 Suchverfahren: Backtracking 217
6.1 Der Suchraum 218
6.2 Das n-Damen-Problem 221
6.3 Graphen-Färbung 228
6.4 Das 0/1-Rucksack-Problem 233
6.5 Hamilton-Zyklen 241
7 Heuristische Suchverfahren 245
7.1 Die Modellierung der Suche 246
7.2 Ein Beispiel: Das $n \times n$-Puzzle 248
7.3 LC-Suche 252
7.4 Auftragsplanung bei unterschiedlichen Bearbeitungszeiten 264
7.5 Das Problem des Handlungsreisenden 269
8 Und-oder-Bäume 283
8.1 Und-oder-Bäume 284
8.2 Max-min-Bäume 290
9 NP-Vollständigkeit 300
9.1 Nicht-deterministische Algorithmen 302

9.2 Simulation eines Rechenmodells auf einem anderen Modell 311
9.3 Entscheidungs- und Optimierungsprobleme 317
9.4 NP-vollständige Probleme 324
9.5 NP-Vollständigkeit von SAT 330
9.6 Einige NP-vollständige Probleme 345
9.6.1 Das Problem CLIQUE 348
9.6.2 Das Problem RSP* 352
9.6.3 Das Problem PART 359
9.6.4 Das Problem RSP 362
9.6.5 Das Problem G-HAMILTON 364
9.6.6 Das Problem U-HAMILTON 365
9.6.7 Das Problem TSP 368
9.6.8 Das Problem G-FÄRB 369
10 Polynomiale Approximation NP-harter Probleme 376
10.1 Approximationen und Approximationsschemata 377
10.2 Absolut approximierbare Probleme 382
10.3 c-approximierbare Probleme 387
10.4 Polynomiale Approximationsschemata 399
10.5 Voll-polynomiale Approximationsschemata 404
Literatur 408

1 Einleitung

1.1 Motivation und Bezeichnungen

Gegenstand der Vorlesung

- Programmieren im Kleinen
- Algorithmen zur Berechnung von Funktionen

also nicht Entwicklung

- großer Systeme
- reaktiver Systeme
- verteilter Systeme

Programmieren im Kleinen bedeutet:

- **Spezifikation eines Problems P**
  Eingabe: Gegeben ist . . .
  Ausgabe: Gesucht ist . . .
  abstrakter:
  P ist beschrieben als Funktion $f : X \to Y$.
- **Erstellen eines Lösungsalgorithmus zu P**
  Verfahren zur Berechnung $x \mapsto f(x)$ für alle $x \in X$.

Beispiele von Problemen

a) **Potenz**

Eingabe:	$z \in \mathbb{Z}, n \in \mathbb{N}$	A Matrix, $n \in \mathbb{N}$
Ausgabe:	$\text{pot}(z,n) = z^n$	$\text{pot}(A,n) = A^n$

b) **Sortieren**

Eingabe: $l = (a_1, \ldots, a_n)$, $a_i \in M$ mit $(M, \leq)$ geordnete Menge

Ausgabe: $\text{sort}(l) = (a_{i_1}, \ldots, a_{i_n})$ mit $a_{i_j} \leq a_{i_{j+1}}$ und $(i_1, \ldots, i_n)$ Permutation von $(1, \ldots, n)$.

c) **Kürzeste Wege in Graphen**

Eingabe: $\Delta = (d_{ij})$ $n \times n$-Matrix über $\mathbb{N} \cup \{\infty\}$, $d_{ij} =$ Kantengewicht von $i \rightsquigarrow j$

Ausgabe: $\Delta^* = (d^*_{ij})$, $d^*_{ij} =$ Länge des kürzesten Weges $i \rightsquigarrow j$

d) **Auftragsplanung mit Terminen**

Eingabe: n Aufträge mit Profiten p_i und Terminen t_i, $i = 1, \ldots, n$. Jeder Auftrag benötigt eine Zeiteinheit.

Ausgabe: Teilfolge $(i_1, \ldots, i_m)$ mit $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, $p_{i_1} + \cdots + p_{i_m}$ maximal, Auftrag i_j ist vor Zeit t_j fertig, $j = 1, \ldots, m$.

Ziele der Vorlesung

Fragen

1. Wie beschreibt man sinnvoll Algorithmen?
2. Gibt es Strategien zum Algorithmen-Entwurf?
3. Wie zeigt man die Korrektheit von Algorithmen?
4. Wie definiert man und wie bestimmt man die Komplexität von Algorithmen?
5. Gibt es optimale Algorithmen zu einem Problem P?
6. Was ist die Komplexität eines Problems P?

Warnung

a) Es gibt unlösbare Probleme.
Beispiel: Post'sches Korrespondenz Problem (PCP).

b) Es gibt Probleme, für die jeder Algorithmus beliebig verbesserbar ist. »Gap-Theorem«.
Siehe z. B. M. Blum, J. ACM 14 (1967) 322-336.

Also

1. Wir betrachten hier nur lösbare Probleme.
2. Wir schätzen die Komplexität von Problemen ab.
3. Wir versuchen, optimale Algorithmen zu entwickeln.

Ziele Techniken für

- Entwurf von Algorithmen
- Analyse von Algorithmen
- Komplexitätsabschätzungen von Algorithmen/Problemen

Komplexität eines Algorithmus (1)

Komplexitätsmaße (häufig benutzt)

1) Zeitbedarf
2) Platzbedarf
3) Kombination von 1) und 2)

Granularität des Maßes (für die Zeit)

Sei $\mathcal{A}$ ein Algorithmus, X eine Menge.

- Definiere $\widehat{T}_{\mathcal{A}} : X \to \mathbb{R}_{\geq 0}$ mit

  $$\widehat{T}_{\mathcal{A}}(x) = \text{Zeitbedarf von } \mathcal{A} \text{ bei Eingabe von } x \in X$$

  $\Rightarrow$ zu feines Maß

- Definiere sinnvoll $|x|$ als »Größe von $x \in X$«,
  etwa Potenz: $|(z, n)| = n$
  Sortieren: $|(a_1, \ldots, a_n)| = n$
  Definiere $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{R}_{\geq 0}$ mit

  $$T_{\mathcal{A}}(n) = \max_{|x|=n} \widehat{T}_{\mathcal{A}}(x)$$

  $\Rightarrow$ »Zeitbedarf im schlechtesten Fall«

Komplexität eines Algorithmus (2)

- Definiere $X_n = \{x \in X \mid |x| = n\}$.
  Definiere $T^*_{\mathcal{A}} : \mathbb{N} \to \mathbb{R}_{\geq 0}$ mit Hilfe des Erwartungswerts

  $$T^*_{\mathcal{A}}(n) = E(\widehat{T}_{\mathcal{A}}(x) \mid x \in X_n)$$

  $\Rightarrow$ »Zeitbedarf im Mittel«
  Dies hängt von der Verteilung ab, mit der die $x \in X$ auftreten.
  Z. B. bei Gleichverteilung

  $$T^*_{\mathcal{A}}(n) = \frac{1}{\|X_n\|} \sum_{x \in X_n} \widehat{T}_{\mathcal{A}}(x)$$

Rechenzeiten für verschiedene Komplexitätsfunktionen und Eingabegrößen

Annahme: Maschine, die 10^6 Operationen/sec. durchführt.
(Pentium $5 \cdot 10^8$).

Rechenzeit (i. Allg. in Sekunden)					
Zeit-komplexität	Eingabegröße n n=10	n=20	n=30	n=40	n=50
$\lfloor \log_2(n) \rfloor$	0.000003	0.000004	0.000004	0.000005	0.000005
n	0.00001	0.00002	0.00003	0.00004	0.00005
$n\lfloor \log_2(n) \rfloor$	0.00003	0.00008	0.00012	0.0002	0.00025
n^2	0.0001	0.0004	0.0009	0.0016	0.0025
n^3	0.001	0.008	0.027	0.064	0.125
2^n	0.001024	1.0	17 Min	12 Tage	35 J.
$n!$	3.6	77146J	·	·	·

Behandelbare Größen von Problemen in gegebener Zeitschranke

Annahme: Maschine, die 10^6 Operationen/sec. durchführt.

Maximale Problemgröße			
Zeit-komplexität	Rechenzeit 1 Sekunde	1 Minute	1 Stunde
$\lfloor \log_2 n \rfloor$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{3.6 \cdot 10^9}$
n	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$
$n\lfloor \log_2 n \rfloor$	62746	2800000	130000000
n^2	1000	7746	60000
n^3	100	390	1500
2^n	20	25	31
$n!$	9	11	12

Fazit

Von der Entwicklung schnellerer Rechner profitieren vor allem die effizienten Algorithmen, also die mit kleinerem Zeitbedarf.

Konstanten in Zeitschranken

Beispiel

Seien $\mathcal{A}_i, i = 1, \ldots, 4$, Algorithmen mit Zeitschranken T_i:

$\mathcal{A}_1 :$	$T_1(n) = 1.000 \cdot n$	am schnellsten für $100 \leq n$
$\mathcal{A}_2 :$	$T_2(n) = 200 \cdot n \cdot \log n$	nie
$\mathcal{A}_3 :$	$T_3(n) = 10 \cdot n^2$	$10 \leq n \leq 100$
$\mathcal{A}_4 :$	$T_4(n) = 2^n$	$n < 10$

T_1 wächst linear, T_3 quadratisch, T_4 exponentiell.
$\Rightarrow$ für große Eingaben ist
$\mathcal{A}_1$ am schnellsten, $\mathcal{A}_4$ am langsamsten.

Fazit

Erst asymptotisches Wachstum bestimmen
$\rightsquigarrow$ wichtig für qualitatives Wachstum.
Dann Konstanten beachten
$\rightsquigarrow$ wichtig für realistische Eingabegrößen.

Bezeichnung für asymptotisches Wachstum (1)

Bezeichnungen (intuitiv)

$g \in O(f)$	heißt g wächst nicht schneller als f
$g \in \Omega(f)$	heißt g wächst nicht langsamer als f
$g \in \Theta(f)$	heißt g wächst gleich schnell wie f
$g \in o(f)$	heißt g wächst langsamer als f

Bezeichnungen (formal)

Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

$$g \in O(f) \quad \text{gdw} \quad \bigvee_{c>0} \bigvee_{n_0 \geq 1} \bigwedge_{n \geq n_0} g(n) \leq c \cdot f(n)$$

$$g \in \Omega(f) \quad \text{gdw} \quad \bigvee_{c>0} \bigvee_{n_0 \geq 1} \bigwedge_{n \geq n_0} g(n) \geq c \cdot f(n)$$

$$g \in \Theta(f) \quad \text{gdw} \quad g \in O(f) \wedge g \in \Omega(f)$$

$$g \in o(f) \quad \text{gdw} \quad \bigwedge_{c>0} \bigvee_{n_0 \geq 1} \bigwedge_{n \geq n_0} g(n) \leq c \cdot f(n)$$

Sei $c \in \mathbb{R}_+$ und $\bigotimes \in \{O, \Omega, \Theta, o\}$.

- $c \cdot \bigotimes(f) := \{c \cdot f' \mid f' \in \bigotimes(f)\}$
- $\bigotimes(f) + \bigotimes(g) := \{f' + g' \mid f' \in \bigotimes(f), g' \in \bigotimes(g)\}$

Bezeichnung für asymptotisches Wachstum (2)

Beispiele

a) $f(n) = \frac{1}{2}n(n+1),\ g(n) = n^2,\ h(n) = n^3$

$$\begin{aligned} \Rightarrow \quad & f \in \Theta(g) \\ & g \in \Theta(f) \\ & f \in o(h) \\ & f \in O(h) \end{aligned}$$

Wir schreiben auch:

$$\begin{aligned} & f \in \Theta(n^2) \\ & f \in O(n^2) \\ & f \in o(n^3) \end{aligned}$$

b) $f(n) = 2^n,\ g(n) = n!$

$\Rightarrow f \in O(g)$,

nicht $g \in O(f)$, da $\lim\limits_{n\to\infty} \frac{2^n}{n!} = 0$

Einige nützliche Abschätzungen

Satz 1.1. [Stirling'sche Formel]

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Folgerung 1.1

a) $\log n! \in n \cdot \log n - \Theta(n)$

b) $\binom{2n}{n} \approx \frac{4^n}{\sqrt{\pi n}}$, $\quad n > 0$ Catalan-Zahlen

c) $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{e\cdot n}{k}\right)^k$, $\quad k > 0$

Die harmonische Reihe ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

Satz 1.2.

$$\ln n \leq H_n \leq 1 + \ln n$$

Für den natürlichen Logarithmus $\ln x = \log_e x$ gilt:

$$\ln x \leq x - 1$$

1.2 Das Maschinenmodell / Beschreibung und Komplexität von Algorithmen

Es gibt viele Modelle zur Beschreibung von Algorithmen, z. B.

- **while** -Programme
- RAM: random access machine
- RM: Register-Maschine
- TM: Turing-Maschine

Hier im Allgemeinen

Beschreibung von Algorithmen als **while** -Programme
mit - arithmetischen Operationen $+, -, *, /$
- Vergleichsoperationen $<, \leq, =, >, \geq, \ldots$
- Boole'schen Operationen

und Schleifen
- **while** -, **for** -, **repeat**-**until** -Schleifen

und Rekursion
und - elementaren Datenstrukturen
- Feldern, Bäumen, Listen, ...

Sie laufen auf einer RAM ab.

Random Access Machine

a) Hardware: Es gibt
α Akkumulator
$\gamma_1, \gamma_2, \gamma_3$ Register (Index-Register)
$0, 1, 2, \ldots$ Nummern der Speicherplätze

b) Befehlstypen
Sei $\varrho(i)$ Inhalt von Zelle i
reg $\in \{\alpha, \gamma_1, \gamma_2, \gamma_3\}$ Register
op $\in \{i, \varrho(i), \varrho(i+\gamma_j), \alpha, \gamma_j \mid i \in \mathbb{N}, 1 \leq j \leq 3\}$ Operand

Es gibt folgende Befehlstypen:

i) Transportbefehle
reg $\leftarrow$ op Laden
op $\leftarrow$ reg Speichern

ii) Sprünge
goto k
if reg $\circledast$ 0 **then goto** k $\circledast \in \{>, \geq, <, \leq, =, \neq\}$

iii) Arithmetische Befehle
$\alpha \leftarrow \alpha *$ op $* \in \{+, -, \cdot, \mathrm{div}\}$

iv) Indexbefehle
$\gamma_j \leftarrow \gamma_j \circ i$ $\circ \in \{+, -\}$

Einheitskostenmaß

Voraussetzung:
Jede Zahl passt in jeden Speicherplatz (Zelle).

$\Rightarrow$ Jede Operation kostet eine Zeiteinheit.
Jeder Zugriff auf ein Datum kostet eine Zeiteinheit
(Zugriff auf Feldelement, Zuweisung).

Zeitbedarf von $\mathcal{A}$ bei Eingabe $x \in X$:
Anzahl der benötigten Zeiteinheiten, falls die Rechnung hält.

Platzbedarf von $\mathcal{A}$ bei Eingabe $x \in X$:
Anzahl der benötigten Speicherplätze, falls die Rechnung hält.

Logarithmisches Kostenmaß

Zahldarstellung

als Dual- oder Dezimalzahl

(Unterscheidung in der Länge nur um konstanten Faktor)

$\Rightarrow$ Länge $|n|$ von $n \in \mathbb{N}$

$$|n| = \begin{cases} 1 & n = 0 \\ 1 + \lfloor \log n \rfloor & n > 0 \end{cases}$$

Logarithmisches Kostenmaß

Voraussetzung:
Es gibt beliebig lange Zahlen/Wörter.

$\Rightarrow$ Nicht jede Zahl passt in eine Zelle.
Jeder Zugriff auf ein Datum hängt von der Länge des Datums ab.

Zeitbedarf von $\mathcal{A}$ bei Eingabe $x \in X$:
Summe der Kosten der ausgeführten Befehle, falls die Rechnung hält.

Platzbedarf von $\mathcal{A}$ bei Eingabe $x \in X$:
Maximale Anzahl der Bits zum Speichern der Daten, falls die Rechnung hält.

Kostenmaße bei der RAM

Annahme Algorithmus $\mathcal{A}$ ist in ein RAM-Programm übersetzt.

Kosten für

1) Zugriffe auf die Operanden	Einh.-K.	logarithm. Kosten
auf i	0	0
auf reg	0	0
auf $\varrho(i)$	1	$\lvert i\rvert$
auf $\varrho(i+\gamma_j)$	1	$\lvert i\rvert + \lvert\gamma_j\rvert$

2) Ausführung des Befehls	Einh.-K.	logarithm. Kosten
Transportbefehl	1	$1+\lvert m\rvert$
Sprungbefehl	1	$1+\lvert k\rvert$
Arithm. Befehl $m_1 * m_2$	1	$1+\lvert m_1\rvert+\lvert m_2\rvert$
Index-Befehl	1	$1+\lvert i\rvert+\lvert\gamma_j\rvert$

$\lvert m\rvert$ = Länge des Wertes der rechten Seite m des Transportbefehls
$\lvert k\rvert$ = Länge der Sprungmarke k
$\lvert m_i\rvert$ = Länge des Wertes des Operanden m_i, $i \in \{1,2\}$
$\lvert i\rvert$ = Länge des Wertes i, $i \in \mathbb{N}$
$\lvert\gamma_j\rvert$ = Länge des Wertes im Register γ_j, $j \in \{1,2,3\}$

Äquivalenz von Maschinenmodellen

These
Alle »vernünftigen« Maschinenmodelle sind polynomial äquivalent: Zu je zwei Modellen $\mathcal{M}_1$ und $\mathcal{M}_2$ gibt es ein Polynom $p(n)$, so dass jede Rechnung auf $\mathcal{M}_1$ mit Eingabe $x \in X$ in Zeit $p(\lvert x\rvert)$ auf $\mathcal{M}_2$ simulierbar ist.

Satz 1.3.

Die obigen Maschinenmodelle **while** -Programme, RAM, RM und TM sind unter dem logarithmischen Kostenmaß polynomial äquivalent.

Zunächst
Beschränkung auf Einheitskostenmaß

1.3 Techniken für Zeitabschätzungen

Analyse von Schleifen

Zeitbedarf einer Schleife = Summe der Zeiten pro Durchlauf
$\Rightarrow$ Problem: Berechnung von Summen

Satz 1.4.

Sei $p \in \mathbb{N}$. Es gilt:

a) $$\sum_{i=1}^{n} i^p \in \Theta\left(n^{p+1}\right)$$

b) $$\sum_{i=1}^{n} i^p \cdot \log i \in \Theta\left(n^{p+1} \cdot \log n\right)$$

Beispiele

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j \in \sum_{i=1}^{n} \Theta\left(i^2\right) \subseteq \Theta\left(\sum_{i=1}^{n} i^2\right) \subseteq \Theta\left(n^3\right)$$

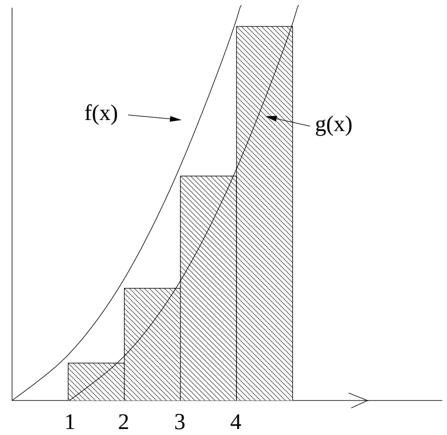
$$\sum_{i=1}^{n}\sum_{j=1}^{i} \log j \in \sum_{i=1}^{n} \Theta(i \cdot \log i) \subseteq \Theta\left(\sum_{i=1}^{n} i \cdot \log i\right) \subseteq \Theta(n^2 \log n)$$

Beweis zu Satz 1.4

Lemma 1.1
Sei $a_i \geq 0$, $s_n = a_1 + a_2 + \cdots + a_n$.
Gilt $g(x) \leq a_i \leq f(x)$ für $i \leq x \leq i+1$, so ist:

$$\int_1^{n+1} g(x)dx \leq s_n \leq \int_1^{n+1} f(x)dx$$

Beweis:



Hier ist $n = 4$ und s_4 die gestreifte Fläche.
Sei a die Treppenfunktion mit $a(x) = a_i$ für $i \leq x < i+1$.
Die Behauptung folgt durch Integration der Funktionen f, g und a.

Beweis zu Satz 1.4 (Forts.)

a) Wähle $f(x) = x^p$, $g(x) = f(x-1) = (x-1)^p$
Die Behauptung folgt aus

- $\displaystyle\int_1^{n+1} x^p dx = \frac{1}{p+1} x^{p+1}\Big|_1^{n+1} \in \Theta(n^{p+1})$
- $\displaystyle\int_1^{n+1} (x-1)^p dx = \int_0^n x^p dx \in \Theta(n^{p+1})$

Also:

$$\sum_{i=1}^{n} i^p \in \Theta(n^{p+1})$$

b) Wähle $f(x) = x^p \log x$ und $g(x) = f(x-1)$ für $x \geq 2$ und $g(x) = 0$ für $1 \leq x < 2$.
Die Behauptung folgt wieder durch Integration:

$$\begin{aligned} \int_1^n x^p \log x dx &= \frac{x^{p+1}}{p+1}\left(\log x - \frac{1}{p+1}\right)\Bigg|_1^n \\ &\in \Theta\left(n^{p+1} \cdot \log n\right) \end{aligned}$$

Analyse von Schleifen: Beispiele (1)

a) Problem P: Eingabe: $n, z \in \mathbb{N}$
Ausgabe: z^n

Algorithmus $\mathcal{A}$ zu P:

$\underline{\textbf{begin}}\ x \leftarrow 1$
$\quad \underline{\textbf{for}}\ i = 1\ \underline{\textbf{to}}\ n\ \underline{\textbf{do}}$
$\qquad x \leftarrow z \cdot x$
$\quad \underline{\textbf{od}}$
$\underline{\textbf{end}}$

Kosten von $\mathcal{A}$ im logarithmischen Kostenmaß

(i) Schleifeninvariante: $x = z^{i-1}$

$$\Rightarrow |x| \in \begin{cases} \Theta(1) & i = 1 \\ \Theta(|z|(i-1)) & i \geq 2 \end{cases}$$

denn $|z \cdot x| \in \Theta(|z| + |x|)$

Induktion nach i liefert die Behauptung.

(ii) Zeit für die Berechnung von $z \cdot x$ ist $\Theta(|z| + |x|)$.

$\Rightarrow$ Aus (i) und (ii) folgt für die Rechenzeit der Schleife:

$$\sum_{i=1}^{n} \Theta(|z| \cdot i) \subseteq \Theta\left(|z| \cdot \sum_{i=1}^{n} i\right) \subseteq \Theta(|z| \cdot n^2)$$

$\Rightarrow \mathcal{A}$ hat Zeitbedarf $\Theta(|z| \cdot n^2)$.

Analyse von Schleifen: Beispiele (2)

b) Problem P: Eingabe: Matrix $A = (a_{ij})_{i,j=1,\ldots,n}$
Ausgabe: Vektor $b = (b_i)_{i=1,\ldots,n}$
mit $b_i = \sum_{j=1}^{i} a_{ij}$

Algorithmus $\mathcal{A}$ zu P:

begin
  for $i = 1$ **to** n **do**
    $x \leftarrow 0$
    for $j = 1$ **to** i **do**
      $x \leftarrow x + a_{ij}$
    od
    $b_i \leftarrow x$
  od
end

Kosten von $\mathcal{A}$ im Einheitskostenmaß

- für die innere Schleife:

$$\sum_{j=1}^{i} \Theta(1) \subseteq \Theta\left(\sum_{j=1}^{i} 1\right) \subseteq \Theta(i)$$

- für die äußere Schleife:

$$\sum_{i=1}^{n} \Theta(i) \subseteq \Theta\left(\sum_{i=1}^{n} i\right) \subseteq \Theta(n^2)$$

$\Rightarrow \mathcal{A}$ hat Zeitbedarf $\Theta(n^2)$.

Analyse von Rekursionen

Beispiel Fibonacci-Zahlen FIB(n)

Problem P: Eingabe: $n \in \mathbb{N}$
Ausgabe: FIB $=$ FIB(n)

Algorithmus $\mathcal{A}$ zu P:

begin
  if $n \leq 1$
    then FIB $\leftarrow 1$
    else FIB $\leftarrow$ FIB$(n-1)$ + FIB$(n-2)$
  fi
end

Zeitbedarf $T(n)$ von $\mathcal{A}$ für FIB(n):

1	für $n = 0, 1$
$c +$ Zeit für FIB$(n-1)$ und FIB$(n-2)$	für $n \geq 2$

Also:

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= T(n-1) + T(n-2) + c \quad n \geq 2 \end{aligned}$$

Dies ist eine Rekurrenzgleichung.
Sie ist explizit lösbar.

Beispiel FIB (Forts.)

Die Lösung der Rekurrenzgleichung ist

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n - c$$

(c_1, c_2 aus $T(0) = T(1) = 1$ bestimmbar).
Also:

$$T(n) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

denn aus $\left|\frac{1-\sqrt{5}}{2}\right| < 1$ folgt $\lim\limits_{n\to\infty} \left(\frac{1-\sqrt{5}}{2}\right)^n = 0$

Problem: Lösung von Rekurrenzgleichungen

Das Master-Theorem (1)

Motivation Zeitanalyse für rek. Programme
Zeitabschätzung für mehrere Aufrufe von $\mathcal{A}$ in $\mathcal{A}$.
Siehe später: Strategie »Teile + Beherrsche«

Satz 1.5.

Seien $a, b \in \mathbb{N}_+$, $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ und
$f(1) = g(1), \quad f(n) = a \cdot f\left(\frac{n}{b}\right) + g(n)$ für $n > 1$
Dann gilt für $n = b^k, k \in \mathbb{N}$

$$f(n) = \sum_{i=0}^{k} a^i g\left(\frac{n}{b^i}\right) = \sum_{i=0}^{k} a^i g\left(b^{k-i}\right)$$

Beweis: Induktion nach k

$$k = 0 : \qquad f(1) = g(1) = \sum_{i=0}^{0} a^i g\left(\frac{n}{b^i}\right)$$

$$\begin{aligned} k \to k+1 : f\left(b^{k+1}\right) &= a \cdot f\left(b^k\right) + g\left(b^{k+1}\right) \\ &\overset{\text{I.V.}}{=} a \cdot \sum_{i=0}^{k} a^i g\left(b^{k-i}\right) + g\left(b^{k+1}\right) \\ &= \sum_{i=1}^{k+1} a^i g\left(b^{(k+1)-i}\right) + g\left(b^{k+1}\right) \\ &= \sum_{i=0}^{k+1} a^i g\left(b^{(k+1)-i}\right) \end{aligned}$$

Das Master-Theorem (2)

Wir setzen $g(n) = c \cdot n^p$ in Satz 1.5.

Satz 1.6.

Seien $a, b, c, p \in \mathbb{N}$, $a, p > 0$, $b > 1$, und

$$\begin{aligned} f(1) &= c, \\ f(n) &= a \cdot f\left(\tfrac{n}{b}\right) + c \cdot n^p \text{ für } n > 1 \end{aligned}$$

Dann gilt für $n = b^k$:

im Fall $a = b^p$:

$$f(n) = c \cdot n^p (k+1)$$

im Fall $a \neq b^p$:

$$f(n) = c \cdot n^p \cdot \frac{\alpha^{k+1} - 1}{\alpha - 1} \text{ mit } \alpha = \frac{a}{b^p}$$

Beweis: Folgerung aus Satz 1.5. Es gilt nämlich:

$$\begin{aligned} f(n) &= c \cdot \sum_{i=0}^{k} a^i \cdot b^{(k-i)p} \\ &= c \cdot b^{kp} \cdot \sum_{i=0}^{k} a^i \cdot b^{-pi} = c \cdot n^p \cdot \sum_{i=0}^{k} \alpha^i \end{aligned}$$

Das Master-Theorem (3)

Satz 1.7.

Seien $a, b \in \mathbb{N}$, $a \geq 1$, $b > 1$ und

$$T(n) \in a \cdot T\left(\tfrac{n}{b}\right) + \Theta(n^p).$$

Es gilt:

$$T(n) \in \begin{cases} \Theta(n^p) & \text{falls} \quad a < b^p \\ \Theta(n^p \cdot \log_2 n) & \qquad\quad a = b^p \\ \Theta(n^{\log_b a}) & \qquad\quad a > b^p \end{cases}$$

Beweis: Satz 1.6 liefert:

a) für $a < b^p$, also $\frac{a}{b^p} < 1$: $\quad T(n) \in \Theta(n^p)$

b) für $a = b^p$:

$$T(n) \in \Theta(n^p \cdot k) = \Theta(n^p \cdot \log_b n) = \Theta(n^p \cdot \log_2 n)$$

c) für $a > b^p$, also $\frac{a}{b^p} > 1$:

$$T(n) \in \Theta\left(n^p \cdot \frac{a^k}{b^{p \cdot k}}\right)$$

Es gilt:

$$\begin{aligned} n^p \cdot \frac{a^k}{b^{p \cdot k}} &= b^{k \cdot p} \cdot \frac{a^k}{b^{p \cdot k}} = a^k = a^{\log_b n} \\ &= a^{\log_a n \cdot \log_b a} = n^{\log_b a} \end{aligned}$$

Man beachte zu b), c): Für alle a, b, x gilt:

$$\frac{\log_b x}{\log_a x} = \log_b a$$

Zeitabschätzung für rekursives $\mathcal{A}$ am Beispiel

$\mathcal{A}$ zerlege ein Problem der Größe n in a Teilprobleme der Größe $\frac{n}{b}$ und benötige Zeit $\Theta(n^p)$ für die Erstellung der Lösung aus den Teillösungen.
Dann hat $\mathcal{A}$ den Zeitbedarf:

$$T(n) \in a \cdot T\left(\frac{n}{b}\right) + \Theta(n^p)$$

Beispiele

a) $T(n) \in 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n^p)$

$$\rightsquigarrow T(n) \in \begin{cases} \Theta(n \cdot \log n) & \text{falls } p = 1 \\ \Theta(n^p) & \text{falls } p > 1 \end{cases}$$

b) $T(n) \in T\left(\frac{n}{2}\right) + \Theta(n^p)$

$$\rightsquigarrow T(n) \in \Theta(n^p)$$

c) $T(n) = a \cdot T\left(\frac{n}{a}\right) + c \qquad c > 0 \qquad a > 1$

$$\rightsquigarrow T(n) \in \Theta(n)$$

Rekursiv definierte Zahlenfolgen

Motivation

Die Analyse von $\mathcal{A}$ liefere eine Zeitabschätzung der Form

$$T(n+1) = f(n) \cdot T(n) + g(n)$$

($f, g : \mathbb{N} \to \mathbb{N}$ fest)
Wie schnell wächst $T(n)$?

Satz 1.8.

Seien $b_1, b_2, b_3, \ldots$ und $c_1, c_2, c_3, \ldots$ feste Folgen über $\mathbb{N}$

a) Gilt $a_1 = b_1, a_{n+1} = c_{n+1} a_n + b_{n+1}$
so folgt

$$a_n = \sum_{i=1}^{n} \left(\prod_{j=i+1}^{n} c_j \right) \cdot b_i$$

b) Gilt $a_1 = b_1, a_{n+1} = c \cdot a_n + b \qquad c \neq 1$
so folgt

$$a_n = b_1 \cdot c^{n-1} + b \cdot \frac{c^{n-1} - 1}{c - 1}$$

Beweis: Induktion nach n.

Beispiel

$T(1) = b, T(n+1) = c \cdot T(n) + b$

$$\begin{array}{lllll} \rightsquigarrow & T(n) & = & b \cdot n & \text{falls } c = 1 \\ & T(n) & = & bc^{n-1} + b\frac{c^{n-1}-1}{c-1} & \\ & & \in & \Theta(c^n) & \text{falls } c \neq 1 \end{array}$$

1.4 Effizienz-Steigerung: Ein Beispiel

Aufgabe: Berechnung der Fibonacci-Zahlen, d.h.

Problem P: Eingabe: $n \in \mathbb{N}$
Ausgabe: $\mathsf{FIB}(n)$ mit

$$\mathsf{FIB}(0) = \mathsf{FIB}(1) = 1$$
$$\mathsf{FIB}(n+2) = \mathsf{FIB}(n+1) + \mathsf{FIB}(n)$$

Algorithmen zu P:

a) Direkte Umsetzung der Definition von FIB

$\mathcal{A}_1$: **if** $n \leq 1$ **then** FIB $\leftarrow 1$
else FIB $\leftarrow$ FIB$(n-1)$ + FIB$(n-2)$
fi

$\rightsquigarrow$ exponentieller Zeitbedarf für $\mathcal{A}_1$:

$$T_1(n) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Beachte: $\frac{1+\sqrt{5}}{2} > 1$

Vorsicht: $\mathcal{A}_1$ berechnet viele FIB(i) mehrfach!

Effizienz-Steigerung (1. Forts.)

b) Parallele Berechnung von FIB(n) und FIB$(n-1)$

b1) Rekursive Version: mit Hilfsfunktion PFIB

Invariante: PFIB$(n) = ($FIB(n), FIB$(n-1))$

PFIB : **if** $n = 0$
then PFIB $\leftarrow (1, 0)$ % FIB$(-1) = 0$
else $(v, w) \leftarrow$ PFIB$(n-1)$
PFIB $\leftarrow (v+w, v)$
fi

$\mathcal{A}_2^R$: $(v, w) \leftarrow$ PFIB(n)
FIB $\leftarrow v$

Zeitbedarf von $\mathcal{A}_2^R$:

$$T_2^R(n) = T_{\mathsf{PFIB}}(n) + c_1$$

$$T_{\mathsf{PFIB}}(n) = \begin{cases} c_2 & n = 0 \\ T_{\mathsf{PFIB}}(n-1) + c_3 & \text{sonst} \end{cases}$$

$\rightsquigarrow T_{\mathsf{PFIB}}(n) = n \cdot c_3 + c_2$

Also:

$$T_2^R(n) = n \cdot c_3 + c_2 + c_1 \in \Theta(n)$$

Effizienz-Steigerung (2. Forts.)

b2) Iterative Version

$\mathcal{A}_2^I$: **begin**

$(v, w) \leftarrow (1, 0)$
for $i = 1$ **to** n **do**
  $(v, w) \leftarrow (v + w, v)$
od
FIB $\leftarrow v$
end

$$\rightsquigarrow \quad \begin{array}{rcll} T_2^I(0) & = & c, & c > 0 \\ T_2^I(n+1) & = & T_2^I(n) + b, & b > 0 \end{array}$$

$\rightsquigarrow T_2^I(n) = b \cdot n + c \in \Theta(n)$

$\rightsquigarrow \mathcal{A}_2^I$ hat wie $\mathcal{A}_2^R$ linearen Zeitbedarf.

Effizienz-Steigerung (3. Forts.)

c) Kreative Lösung

(i) Die Berechnung von PFIB kann mit Techniken der linearen Algebra beschrieben werden.

Sei $(v, w) = \text{PFIB}(n - 1)$.

Dann ist $\text{PFIB}(n) = (v + w, v)$, also

$$\text{PFIB}(n) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} v \\ w \end{pmatrix}$$

Induktion nach n liefert:

$$\text{PFIB}(n) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

(ii) Strategie für schnelles Potenzieren »Teilen+Beherrschen«

pot(A, n) :

if $n = 1$ **then** $B \leftarrow A$
  else $m \leftarrow n$ div 2
    $B \leftarrow$ pot(A, m)
    $B \leftarrow B * B$
    if $n \neq 2m$ **then** $B \leftarrow B * A$ **fi**
fi
pot $\leftarrow B$

Effizienz-Steigerung (4. Forts.)

(iii) Berechnung von FIB(n) mit pot

$$
\begin{aligned}
\mathcal{A}_3 : A &\leftarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\
B &\leftarrow \mathsf{pot}(A, n) \\
\mathsf{FIB} &\leftarrow B_{1,1} \qquad \text{\% Element oben links von B}
\end{aligned}
$$

logarithmischer Zeitbedarf für $\mathcal{A}_3$:

$$T_3(n) \in \Theta(\log n)$$

Ergebnis der Effizienz-Steigerung hinsichtlich Zeitbedarf

$T_1(n) \in \Theta(c^n)$	exponentiell
$T_2^R(n) \in \Theta(n)$	linear
$T_2^I(n) \in \Theta(n)$	linear
$T_3(n) \in \Theta(\log n)$	logarithmisch

Bemerkung zur Rekursion

1. Rekursion ist wichtig für den **Entwurf** von Algorithmen.
   $\Rightarrow$ angemessene Entwurfsstrategie für viele Probleme
   gut lesbare Darstellung
2. Die direkte Implementierung der Rekursion führt häufig zu Ineffizienzen.
   $\Rightarrow$ größerer Zeitbedarf (konstanter Faktor)
   größerer Speicherbedarf (für Keller)
3. Rekursive Algorithmen kann man häufig »iterativ« programmieren.
   $\Rightarrow$ automatische Elimination der »Tail Recursion« möglich
4. Hinweis zur Lösung von Problemen
   (a) Gute Lösungsidee entwickeln
   (b) Angemessene Strategie für den Algorithmen-Entwurf verwenden
   (c) Korrektheit prüfen
   (d) Effizient implementieren
   (Dies benötigt i. Allg. die wenigste Kreativität!)

$\Rightarrow$
- Rekursion ist von zentraler Bedeutung für die Erstellung guter Algorithmen für (nicht-triviale) Probleme.
- Dies gilt ebenso für die Auswahl der Entwurfsstrategie. Welche Strategien gibt es?

1.5 Probabilistische Algorithmen

Idee

Zufall in Algorithmen einbauen
$\Rightarrow$ Verwendung von Zufallszahlen

Realisierung

Ideal:	Orakel für Zufallszahlen
Praktisch:	Berechnung von Pseudo-Zufallszahlen
Beachte:	Zufallszahlen immer bezüglich einer

- vorgegebenen Verteilung
  (z. B. Gleichverteilung, Poisson-Verteilung)
- über einer festen Menge M

Berechnung von Pseudo-Zufallszahlen $z_0, z_1, z_2, \ldots$

$z_0 := a$ $\quad$ $a \in M$ »seed«
$z_{i+1} := f(z_i)$ $\quad$ f »Zufallsgenerator«

Zufallszahlen (Forts.)

Beispiele für Gleichverteilung

1. Kongruenz-Methode für $M = [0, \ldots, c-1] \subseteq \mathbb{N}$:
   Seien $a, b, c \in \mathbb{N}$ »geeignet« (groß).
   Setze $\quad z_{i+1} := (a \cdot z_i + b) \bmod c$
2. Normierung auf $M = [\alpha, \ldots, \beta] \subseteq \mathbb{N}$:
   Sei $\gamma = \beta - \alpha$,
   $z_i' \in [0, \ldots, c-1]$ erzeugt nach Kongruenz-Methode.
   Setze $\quad z_i := \left\lfloor \frac{z_i'}{c} \cdot (\gamma + 1) \right\rfloor + \alpha$

Zufalls-Befehl

Syntax:
random x **in** M

Semantik:
Zufalls-Befehl liefert $x \in M$ »zufällig«.

Typen von probabilistischen Algorithmen

Bei probabilistischen Algorithmen hängt i. Allg. sowohl

- die Laufzeit als auch
- das Ergebnis vom Zufall ab.

Laufzeit

Sei $\mathcal{A}$ ein fester Algorithmus, der mit einer festen Zufallsverteilung (einem festen Zufallsgenerator) arbeitet.

$T'_{\mathcal{A}}(x)$ ist der Erwartungswert der Laufzeit von $\mathcal{A}$ bei fester Eingabe x.

$$T_{\mathcal{A}}(n) = \max_{|x|=n} T'_{\mathcal{A}}(x)$$

ist der Zeitbedarf von $\mathcal{A}$ im schlechtesten Fall.

$$T^*_{\mathcal{A}}(n) = \frac{1}{\|X_n\|} \sum_{x \in X_n} T'_{\mathcal{A}}(x) \quad \text{mit } X_n = \{x \mid |x| = n\}$$

ist der Zeitbedarf von $\mathcal{A}$ im Mittel.

Typen von probabilistischen Algorithmen (Forts.)

Bezeichnungen

$\mathcal{A}$ heißt **Las-Vegas-Algorithmus** für das Problem P, falls er entweder die richtige Antwort zur Eingabe x oder »Misserfolg« als Ausgabe liefert.

$\Rightarrow$ Jede berechnete Antwort ist korrekt.

$\mathcal{A}$ heißt **Monte-Carlo-Algorithmus** zu P, falls er zur Eingabe x mit hoher Wahrscheinlichkeit (?) die richtige Antwort liefert.

$\Rightarrow$ Eine berechnete Antwort kann falsch sein.

Beispiel: Bubble-Sort probabilistisch

Problem P: Eingabe: Vektor $A = (a_i)_{i=1,\dots,n}$
Ausgabe: Vektor A sortiert

Algorithmus $\mathcal{A}$ zu P:

repeat
  random i **in** $[1, \dots, n-1]$
  if $a_i > a_{i+1}$
    then vertausche(a_i, a_{i+1}) **fi**
until sorted(A, n)

Analyse:

- **Zeitbedarf** ist schwer zu bestimmen.
- Es sind sogar unendliche Rechnungen möglich.
- Jedes berechnete **Ergebnis** ist korrekt.

  $\Rightarrow$ Las-Vegas-Algorithmus

Beispiel: Primzahl-Test

Problem P: Eingabe: $n \in \mathbb{N}, n > 2$
Ausgabe: Ist n eine Primzahl?

Algorithmus $\mathcal{A}$ zu P »mit k Versuchen«:

$\mathcal{A}$: $\mathrm{Ausg} \leftarrow$ **true**
  $i \leftarrow 1$
  $m \leftarrow [\frac{n}{2}]$
  while $\mathrm{Ausg} \wedge i \leq k$ **do**
    $i \leftarrow i + 1;$
    random x **in** $[2, \dots, m]$
    $z \leftarrow$ transform(x)
    if $z | n$ **then** $\mathrm{Ausg} \leftarrow$ **false** **fi**
  od

Analyse:

a) Sei transform$(x) = x$
Die Antwort Ausg=**false** ist immer richtig, die Antwort **true** nur mit Wahrscheinlichkeit $p(k)$. Gilt $p(k) \to 1$ für $k \to \infty$?

b) Es gibt ein Transformations-Verfahren transform(x), so dass $p(k)$ sehr schnell gegen 1 strebt (M. Rabin).

$\Rightarrow$ $\mathcal{A}$ ist Monte-Carlo-Algorithmus

Beachte: Das Problem ist schwer (nicht polynomial (?), siehe Abschnitt 1.6). Das Rabin-Verfahren ist schnell, liefert mit hoher Wahrscheinlichkeit die richtige Antwort.

1.6 Nicht-deterministische Algorithmen

Vorbemerkungen:

- Einführung hier informell, später genauer.
- Beachte Analogie zu nicht-deterministischen Endlichen Automaten bzw. Kellerautomaten

Betrachtete Problemklasse:

zunächst nur Entscheidungsprobleme, also:
Ausgabe JA oder NEIN. Auch Ausgabe »Weiß nicht« zugelassen.

Definition:

Ein Algorithmen $\mathcal{A}$ heißt **nicht-deterministischer Entscheidungsalgorithmus** zu einem Problem P, falls er

- möglicherweise Random-Befehle enthält,
- bei jeder Eingabe x hält und
- eine Rechnung mit Antwort JA möglich ist, falls $P(x)$ gilt.

Beispiel für einen nicht-deterministischen Entscheidungsalgorithmus

Das Erfüllbarkeitsproblem der Aussagenlogik:

Eingabe: aussagenlogische Formel A in den Variablen
$V = \{x_1, \dots, x_n\}$
Ausgabe: 1 falls es eine Belegung
$\varphi : V \to \mathbb{B} = \{0, 1\}$
gibt mit $\varphi(A) = 1$ und 0 sonst.

Algorithmus:

$\mathcal{A}$: **random** φ **in** Menge der Belegungen
$\mathrm{Ausg} \leftarrow \varphi(A)$

$\Rightarrow$ $\mathcal{A}$ ist ein nicht-deterministischer Entscheidungsalgorithmus für das Erfüllbarkeitsproblem.

Die Klassen P und NP

Definition:

Ein Algorithmus $\mathcal{A}$ heißt **polynomial** (hat polynomialen Zeitbedarf), falls es ein Polynom $p : \mathbb{N} \to \mathbb{N}$ gibt, so dass für alle $n \in \mathbb{N}$ gilt:

$$T_{\mathcal{A}}(n) \leq p(n)$$

Frage

Was hilft der Nichtdeterminismus (das »Raten«) bei den polynomialen Entscheidungsalgorithmen, d.h.:
Gilt für jedes Entscheidungsproblem Q:

$$Q \text{ hat einen deterministischen polynomialen Algorithmus}$$
$$\text{gdw}$$
$$Q \text{ hat einen nicht-deterministischen polynomialen Algorithmus?}$$

Die Antwort auf diese Frage ist nicht bekannt!!!
$\Rightarrow$ berühmtes **P-NP-Problem**

Definition:

P = Klasse der Probleme mit polynomialem deterministischen Entscheidungsalgorithmus
NP = Klasse der Probleme mit polynomialem nicht-deterministischen Entscheidungsalgorithmus

$\Rightarrow$ P-NP-Problem lautet: Gilt P = NP?
Die Vermutung ist NEIN

NP-vollständige Probleme

Beachte P $\subseteq$ NP.
Gibt es Test-Probleme P für die Frage, ob NP $\subseteq$ P gilt? JA!

Definition

a) Ein Problem Q_1 ist **polynomial reduzierbar** auf ein Problem Q_2, falls es eine (Transformations-)Funktion f gibt mit
   - $Q_1(x) \text{ gdw } Q_2(f(x))$, und
   - es gibt einen deterministischen polynomialen Algorithmus $\mathcal{A}$ zur Berechnung von f.

   Bezeichnung: $Q_1 \leq_{\mathsf{pol}} Q_2$

b) $Q \in$ NP heißt **NP-vollständig**, falls für alle $Q' \in$ NP gilt: $Q' \leq_{\mathsf{pol}} Q$.

Die NP-vollständigen Probleme sind also die $\leq_{\mathsf{pol}}$-schwersten Probleme in NP.

Lemma 1.2

$Q_1 \leq_{\mathsf{pol}} Q_2$, $Q_2 \in$ P $\Rightarrow Q_1 \in$ P

Beweis: Übung (siehe auch später)

Satz 1.9.

Sei Q NP-vollständig, dann gilt:

$$\mathsf{P} = \mathsf{NP} \text{ gdw } Q \in \mathsf{P}$$

Beispiele NP-vollständiger Probleme (1)

Es gibt viele NP-vollständige Probleme von praktischer und theoretischer Bedeutung (auch wirtschaftlich). Für sie ist kein polynomialer deterministischer Algorithmus bekannt.

1. Das Erfüllbarkeitsproblem SAT

Sei $V = \{x_1, \ldots, x_n\}$ eine Menge aussagenlogischer Variablen.

- Ein Literal ist x_i oder $\bar{x}_i$.
- Eine Klausel ist $l_1 \vee \cdots \vee l_p$, l_i Literale.
- Eine Formel in konjunktiver Normalform (KNF) ist
  $$A \equiv c_1 \wedge \cdots \wedge c_r \;, \quad c_j \text{ Klauseln.}$$
  $\Rightarrow$ A ist also in KNF, falls A eine Konjunktion von Disjunktionen von Literalen ist.

  Z. B.: $A \equiv (x_1 \vee \bar{x}_2 \vee x_5) \wedge x_3 \wedge (\bar{x}_1 \vee x_4 \vee \bar{x}_5)$

Definition

Das Problem SAT ist gegeben durch:

Eingabe:	A in KNF
Ausgabe:	JA, falls A erfüllbar ist; NEIN sonst.

Beispiele

$A_1 \equiv x_1 \wedge (\bar{x}_1 \vee x_2)$ ist erfüllbar.

$A_2 \equiv x_1 \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2)$ ist nicht erfüllbar.

Beispiele NP-vollständiger Probleme (2)

2. Das Rucksackproblem

Gegeben sind:

- n Objekte mit
- Profiten $p_1, \ldots, p_n$, $p_i \in \mathbb{N}_+$, und
- Gewichten $w_1, \ldots, w_n$, $w_i \in \mathbb{N}_+$, sowie
- ein Rucksack mit Kapazität K, $K \in \mathbb{N}_+$.

- $x = (x_1, \ldots, x_n)$ mit $x_i \in \{0, 1\}$ heißt Mitnahmevektor.
- x heißt zulässig, falls $\sum_{i=1}^{n} w_i x_i \leq K$

Frage: Welchen Profit kann man maximal erzielen?

Definition

Das Problem RSP ist gegeben durch:

Eingabe:	$p_1, \ldots, p_n, w_1, \ldots, w_n, K, P$
Ausgabe:	JA, falls es einen zulässigen Mitnahmevektor x gibt mit $\sum_{i=1}^{n} p_i x_i \geq P$; NEIN sonst.

Beispiele NP-vollständiger Probleme (3)

3. Probleme der kürzesten Rundreise

- Sei $D = (d_{i,j})$ eine $n \times n$-Matrix (Kantengewichte) mit $d_{i,j} \in \mathbb{N}$.
- Eine Rundreise $\pi = (\pi_1, \dots, \pi_n)$ ist eine Permutation von $(1, 2, \dots, n)$.
- Ihre Länge ist:

$$l(\pi) = d_{\pi_1,\pi_2} + \cdots + d_{\pi_{n-1},\pi_n} + d_{\pi_n,\pi_1}$$

Frage: Wie lang ist die kürzeste Rundreise?

Definition

Das Problem TSP (travelling salesman problem) ist gegeben durch:

Eingabe: $D = (d_{i,j}), L \in \mathbb{N}_+$

Ausgabe: JA, falls es eine Rundreise π gibt mit $l(\pi) \leq L$;
NEIN sonst.

Ergebnisse zu NP-vollständigen Problemen

Satz 1.10.

Die Probleme SAT, RSP, TSP sind NP-vollständig.
o. Beweis (siehe später)

Beachte Die Probleme SAT, RSP und TSP sind als Entscheidungsprobleme definiert.

Viele Entscheidungsprobleme haben ein natürliches Optimierungsproblem, z. B.

RSP Welcher Profit ist erreichbar?
TSP Wie lang ist die kürzeste Rundreise?
oder
RSP Bestimme einen optimalen zulässigen Mitnahmevektor.
TSP Bestimme eine kürzeste Rundreise.

Satz 1.11.

Für $Q \in \{\text{RSP, TSP}\}$ gilt:

$Q \in \mathrm{P}$ gdw Das Optimierungsproblem zu Q ist deterministisch in polynomialer Zeit lösbar.

Folgerung: Gilt $\mathrm{P} \neq \mathrm{NP}$ dann gilt auch:
Es gibt keinen deterministischen polynomialen Algorithmen für das Optimierungsproblem zu RSP bzw. TSP.

2 Strategie Teilen und Beherrschen
Probleme: Suchen und Sortieren

Voraussetzung

- Die Berechnung von $f : X \to Y$ sei das Problem und
- $|x|$ die Größe der Eingabe $x \in X$.

Strategie

1. Zerlege das Problem zur Eingabe x in b Teilprobleme zur Eingabe $x_1, \ldots, x_b$.
2. Löse die Teilprobleme (häufig rekursiv nach der gleichen Methode).
3. Baue die Gesamtlösung aus den Teillösungen auf.

Kosten entstehen für:

- Erstellung der Teilprobleme
- rekursive Aufrufe
- Erstellung der Lösung aus den Teillösungen

Teilen und Beherrschen: Ein Beispiel

Aufgabe: Berechnung der Potenz pot(a, n) für $n > 0$

Idee: Zerlege pot(a, n) in ein Teilproblem pot$(a, [n/2])$.

Rekursiver Algorithmus:

```
pot^R(a, n) : if n = 1
                then pot ← a
                else if 2|n
                       then pot ← pot^R(a, n/2)^2
                       else pot ← a · pot^R(a, [n/2])^2
                     fi
              fi
```

Iterativer Algorithmus:

```
pot^I(a, n) : x ← a;  z ← 1;  p ← n
              while p > 1 do              % a^n = z · x^p
                    if not 2|p
                      then z ← z · x
                    fi
                    x ← x · x
                    p ← p div 2
              od
              pot ← x · z
```

Zeitbedarf

$$T_{\mathrm{pot}^R}(n) \in \Theta(\log n) \qquad T_{\mathrm{pot}^I}(n) \in \Theta(\log n)$$

2.1 Binäre Suche

Problem

Eingabe:
- $A = (a_i)_{i=1,\ldots,n}$ mit $a_1 < a_2 < \cdots < a_n$
- x

Ausgabe: $\text{erg} = \begin{cases} j & \text{falls } x = a_j \\ 0 & \text{falls } x \neq a_i \text{ für alle } i \end{cases}$

Zerlegung

1. Zerlege das Suchfeld in zwei etwa gleich große Teilfelder.
2. Suche im ersten Teilfeld, falls x nicht im zweiten Teilfeld ist, sonst suche im zweiten Teilfeld.

Binäre Suche (Forts.)

Algorithmus

$\mathcal{A}$: erg $\leftarrow$ binsuche$(1, n)$

mit

```
binsuche (links, rechts) :
        if rechts < links
          then binsuche ← 0
          else mitte ← (rechts + links) div 2
               if x = A[mitte]
                 then binsuche ← mitte
                 else if x < A[mitte]
                        then binsuche (links, mitte − 1)
                        else binsuche (mitte +1, rechts)
                      fi
               fi
        fi
```

Zeitbedarf (schlechtester Fall)

$T(n) = T(n/2) + c$

$\rightsquigarrow T(n) \in \Theta(\log n)$

$\rightsquigarrow \mathcal{A}$ hat logarithmischen Zeitbedarf.

2.2 Sortieren

Problem

Eingabe:
- Feld $A = (a_i)_{i=1,\dots,n}$ mit
- totaler Ordnung $\leq$ auf den Feldelementen

Ausgabe: Permutation $\pi = (a_{i_1}, \dots, a_{i_n})$ der a_i mit

$$a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$$

Beachte

Typ der Feld-Elemente ist nicht festgelegt.
$\Rightarrow$ polymorphes Sortieren

Zerlegungsmöglichkeiten: 2 Teilprobleme

a) Länge $n \to n-1 : 1$
b) Länge $n \to n/2 : n/2$
c) Länge $n \to n-k : k$ $\quad$ k abhängig von Eingabe

$\Rightarrow$ a) Sortieren durch Auswahl
b) Sortieren durch Mischen
c) Sortieren à la Quicksort

2.2.1 Sortieren durch Auswahl: SelectionSort

Verfahren

- Teilen im Verhältnis $n-1 : 1$
- Erstellung der Teilprobleme
  - bestimme maximales a_i
  - vertausche es mit a_n
- Rekursiver Aufruf
  - sortiere Anfangsfeld $A[1,\dots,n-1]$
- Zusammenbau der Lösung
  - trivial

Rekursiver Algorithmus

```
SelectionSort(n):   if n > 1
                      then k ← 1; b ← A[1]
                           for i = 2 to n do
                              if b < A[i]
                                then b ← A[i]
                                     k ← i
                              fi
                           od
                           vertausche(A[k], A[n])
                           SelectionSort(n − 1)
                    fi
```

SelectionSort iterativ programmiert

Iterativer Algorithmus

SelectionSort(n):

```
for m = n step −1 to 2 do
    k ← 1
    b ← A[1]
    for i = 2 to m do
        if b < A[i]
          then b ← A[i]
               k ← i
        fi
    od
    vertausche(A[k], A[m])
od
```


Analyse von SelectionSort

Vorbemerkung

- Beschränkung auf die iterative Version
- Analyse der rekursiven Version: Übung

Zur Korrektheit

Es gilt die Schleifeninvariante:

- $A[m,\ldots,n]$ ist aufsteigend sortiert und
- $A[i] \leq A[m]$ für alle $i \leq m$

Zeitbedarf (im besten und schlechtesten Fall)

$$T(n) \in \sum_{m=2}^{n} \sum_{i=2}^{m} \Theta(1) \subseteq \Theta(n^2)$$

Platzbedarf

außer Platz für Eingabe nur wenig Speicherplätze, unabhängig von n
$\Rightarrow$ „sorting in place“

Ergebnis der Analyse

- leicht zu programmieren
- wenig zusätzlicher Platz
- Zeitbedarf quadratisch
- nur maximal $2 \cdot n$ Umspeicherungen von Feldelementen
  Dies ist wichtig, falls die a_i selbst große Daten sind.

2.2.2 Sortieren durch Mischen: MergeSort

Verfahren

- Teilen im Verhältnis $n/2 : n/2$
- Erstellung der Teilprobleme
  - trivial: 1. Hälfte und 2. Hälfte
- Rekursiver Aufruf
  - $B_1 \leftarrow$ 1. Hälfte sortiert
  - $B_2 \leftarrow$ 2. Hälfte sortiert
- Zusammenbau der Lösung
  - $A \leftarrow$ mischen (B_1, B_2)

Rekursiver Algorithmus

MergeSort(A,n): mergesort(A,1,n)

mit

mergesort(A, links, rechts):

```
if rechts > links
  then mitte ← (rechts+links) div 2
       mergesort(A, links, mitte)
       mergesort(A, mitte+1, rechts)
       Mische(A, links, mitte, rechts)
fi
```


Analyse von MergeSort

Beachte zu »Mische«

- Realisierung mit Hilfe des Algorithmus »mische« (siehe später) möglich.
- Mische(A, links, mitte, rechts) benötigt maximal $m = \mathit{rechts} - \mathit{links}$ Vergleiche.

Es gilt im besten und schlechtesten Fall:

Anzahl der Vergleiche

$$C(n) \leq n + 2 \cdot C\left(\tfrac{n}{2}\right) \qquad C(1) = 0$$

Also (Induktion nach n):

$$C(n) \leq n \cdot \log_2 n$$

Zeitbedarf

$$T(n) \in 2 \cdot T\left(\tfrac{n}{2}\right) + \Theta(n)$$

Also:

$$T(n) \in \Theta(n \cdot \log n)$$

Platzbedarf

außer der Eingabe bei geschickter iterativer Programmierung (siehe folgende Folien):

$$O(n)$$

MergeSort iterativ

Idee

- Rekursion „von innen nach außen“ programmieren.
- Nur ein Hilfsfeld benutzen.

Beispiel

- $n = 11$
- l=Länge der Teilprobleme

A	5	3	1	6	7	7	3	9	8	2	1	l=1	liefert
B	3	5	1	6	7	7	3	9	2	8	1	l=2	liefert
A	1	3	5	6	3	7	7	9	1	2	8	l=4	liefert
B	1	3	3	5	6	7	7	9	1	2	8	l=8	liefert
A	1	1	2	3	3	5	6	7	7	8	9		

MergeSort iterativ (1. Forts.)

Mergesort(A, B, n) sortiert das Feld A der Länge n und speichert das Ergebnis in Feld B:

Algorithmus Mergesort

Mergesort(A, B, n): $l \leftarrow 1$
    flag $\leftarrow$ **true**;
    while $l < n$ **do**
        if flag
            then msort(A, B, l, n)
            else msort(B, A, l, n)
        fi
        $l \leftarrow 2 \cdot l$
        flag $\leftarrow$ **not** flag
    od
    if flag
        then $B \leftarrow A$
    fi

MergeSort iterativ (2. Forts.): msort

msort(A, B, l, n) mischt von A nach B mit Blocklänge l:

Algorithmus msort

msort(A, B, l, n):

$k \leftarrow 1$
$i \leftarrow 1;\ ii \leftarrow l$
$j \leftarrow ii + 1;\ jj \leftarrow \min(n, j + l - 1)$
while $k \leq n$ **do**
  mische(A, i, ii, j, jj, B, k)
  $i \leftarrow jj + 1;\ ii \leftarrow \min(n, i + l - 1)$
  $j \leftarrow ii + 1;\ jj \leftarrow \min(n, j + l - 1)$
  $k \leftarrow i$
od

Beachte

Hier sind alle Feld-Parameter Referenz-Parameter!

MergeSort iterativ (3. Forts.): Mischen

mische(A, i, ii, j, jj, B, k) mischt die beiden Teilfelder $A[i, \ldots, ii]$ und $A[j, \ldots, jj]$ nach $B[k, \ldots, kk]$:

Algorithmus mische

mische(A, i, ii, j, jj, B, k):

while $i \leq ii$ **and** $j \leq jj$ **do**
  if $a_i \leq a_j$
    then $b_k \leftarrow a_i;\ i \leftarrow i + 1;\ k \leftarrow k + 1$
    else $b_k \leftarrow a_j;\ j \leftarrow j + 1;\ k \leftarrow k + 1$
  fi
od
while $i \leq ii$ **do**
  $b_k \leftarrow a_i;\ i \leftarrow i + 1;\ k \leftarrow k + 1$
od
while $j \leq jj$ **do**
  $b_k \leftarrow a_j;\ j \leftarrow j + 1;\ k \leftarrow k + 1$
od

2.2.3 Sortieren durch Zerlegen: Quicksort

Verfahren

- Teilen im Verhältnis $n - k : k$
  - k hängt von Eingabe ab
- Erstellung der Teilprobleme
  - wähle x in A beliebig und zerlege A in
    - A_1= Feld mit $a_i \leq x$ für alle a_i in A_1
    - A_2= Feld mit $a_i \geq x$ für alle a_i in A_2
- Rekursive Aufrufe
  - liefern A_1, A_2 sortiert
- Zusammenbau der Teillösungen
  - trivial: $A = A_1 * A_2$

Algorithmus (Grobversion)

Quicksort(A) = wähle x aus A
Quicksort(A_1) $*$ Quicksort(A_2)
mit (A_1, A_2) = Partition(A, x)

Vorsicht beim Platzbedarf

- Rekursionstiefe nicht zu groß werden lassen.
  $\Rightarrow$ Kleineres Teilproblem erst lösen.
  Realisierbar bei iterativer Lösung.
- Nicht mehrfach Platz für das Feld benutzen.
  $\Rightarrow$ A_1, A_2 auf dem Platz von A speichern.
  Realisierbar schon bei rekursiver Lösung.

Realisierung von Partition

Partition$(A, x, links, rechts, l, r)$ ordnet das Teilfeld $A[links, \ldots, rechts]$ so an, dass gilt:

$$\left.\begin{array}{ll} A[i] \leq x & \text{für } links \leq i \leq r \\ A[i] \geq x & \text{für } l \leq i \leq rechts \end{array}\right\} (\ddagger)$$

Algorithmus

Partition$(A, x, links, rechts, l, r)$ =
  Ref: A, l, r
  $l \leftarrow links;\ r \leftarrow rechts$
  repeat
    while $A[l] < x$ **do** $l \leftarrow l + 1$ **od**
    while $A[r] > x$ **do** $r \leftarrow r - 1$ **od**
    if $l \leq r$ **then** vertausche$(A[l], A[r])$
      $l \leftarrow l + 1;\ r \leftarrow r - 1$ **fi**
  until $l > r$

Zur Korrektheit

- Schleifeninvariante ist:

$$\begin{array}{ll} A[i] \leq x \text{ für alle } i < l & A[i] \geq x \text{ für ein } i \geq l \\ A[i] \geq x \text{ für alle } i > r & A[i] \leq x \text{ für ein } i \leq r \end{array}$$

  Also: Die inneren while-Schleifen halten immer.
- Beim Stop gilt $l > r$ und $(\ddagger)$.

Realisierung von Quicksort

Algorithmus

Quicksort(A, n) $=$ Ref: A
qsort$(A, 1, n)$

mit

qsort$(A, \mathit{links}, \mathit{rechts})$=
Ref: A
<u>if</u> $\mathit{rechts} > \mathit{links}$
<u>then</u>
wähle$(x, A, \mathit{links}, \mathit{rechts})$
Partition$(A, x, \mathit{links}, \mathit{rechts}, l, r)$
qsort(A, links, r)
qsort(A, l, rechts)
<u>fi</u>

Bemerkungen zu Quicksort

Beachte

- Wahl von x beeinflusst das Teilungsverhältnis.
- Es ist $k = 1$ und $k = n - 1$ möglich.
- Im Wesentlichen gilt:
  $x = k$-größtes $A[i] \rightsquigarrow$ Teilung $k : n - k$
- Also: Sei $m = (\mathit{links} + \mathit{rechts})$ div 2.
  Bestimme m-größtes $A[i]$, nimm es als x.
  $\Rightarrow$ zu teuer: Zeitbedarf ist $O(\mathit{rechts} - \mathit{links})$.
  $\Rightarrow$ x in konstanter Zeit wählen.

Zur Wahl von x für Partition$(A, x, \mathit{links}, \mathit{rechts}, l, r)$

a) $x = A$[links] oder $x = A$[rechts]
$\Rightarrow$ Schlechte Teilung $1 : n - 1$ für vorsortierte Felder

b) $x = A[$(links+rechts) div $2]$, d.h. „die Mitte"
$\Rightarrow$ Gute Teilung $n/2 : n/2$ für vorsortierte Felder.

c) $x =$ Mittelwert aus 3 bzw. 5 A-Werten
$\Rightarrow$ In der Praxis gut bewährt.

Analyse von Quicksort (1)

Sei $C(n)$=Anzahl der Vergleiche.
Es gilt für den Zeitbedarf:

$$T(n) \leq c \cdot C(n) \qquad c \text{ feste Konstante}$$

Lemma 2.1

a) Es gilt $C(n) \leq n^2$.
b) Es gibt $A[1, \ldots, n]$ mit $C(n) \in \Theta(n^2)$.

Beweis:

a) Induktion nach n:

$$\begin{array}{lcl} C(n) & \leq & n + 1 + \max\limits_{1<k<n} \{C(k) + C(n-k)\} \\ & \underset{(\text{I.V.})}{\leq} & n + 1 + \max\limits_{1<k<n} \{k^2 + (n-k)^2\} \\ & \leq & n + 2 + (n-1)^2 \leq n^2 \end{array}$$

b) Übung (leicht, falls stets $x = A[\text{links}]$)

Analyse von Quicksort (2)

Satz 2.1.

Quicksort hat Zeitbedarf im schlechtesten Fall:

$$T_{\text{max}}(n) \in \Theta(n^2)$$

Satz 2.2.

Man kann Quicksort iterativ so realisieren, dass für den Platzbedarf $S(n)$ (neben der Eingabe) gilt:

$$S(n) \in O(\log n)$$

Beweis: Übung!

Analyse von Quicksort (3)

Analyse im besten Fall

Lemma 2.2

Quicksort benötigt minimal

$$C_{\mathsf{min}}(n) \geq n \cdot \log_2 n$$

Vergleiche.

Beweis: Induktion nach n:

$$\begin{aligned} C_{\mathsf{min}}(n) &\geq n + \min_{1<k<n} \{C_{\mathsf{min}}(k) + C_{\mathsf{min}}(n-k)\} \\ C_{\mathsf{min}}(n) &\geq n + 2 \cdot C_{\mathsf{min}}(n/2) \quad \leadsto \text{Beh.} \end{aligned}$$

Analyse im Mittel

Voraussetzung:

Gleichverteilung der $A[i]$

$\Rightarrow$ Wahrscheinlichkeit der Teilung $k : n-k$ ist $1/n$.

Lemma 2.3

Quicksort benötigt im Mittel

$$C_{\mathsf{mit}}(n) \leq 2 \ln 2 \cdot n \cdot \log_2 n$$

Vergleiche.

Beweis zu Lemma 2.3

Beweis: Sei $C(n) = C_{\mathsf{mit}}(n)$. Es gilt $C(1) = 0$.

$$\begin{aligned} C(n) &= (n+1) + \frac{1}{n}\sum_{k=1}^{n-1}(C(k) + C(n-k)) \\ &= (n+1) + \frac{2}{n}\sum_{k=1}^{n-1} C(k) \\ n \cdot C(n) &= n(n+1) + 2 \cdot \sum_{k=1}^{n-1} C(k) \\ n \cdot C(n) &- (n-1) \cdot C(n-1) = 2n + 2C(n-1) \\ \frac{C(n)}{n+1} &= \frac{2}{n+1} + \frac{C(n-1)}{n} = \cdots \\ &= \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{3} + \frac{C(1)}{2} \\ \frac{C(n)}{n+1} &\leq 2 \cdot \sum_{k=3}^{n+1} \frac{1}{k} = 2(\ln n - \alpha) \\ C(n) &\leq 2(n+1) \cdot (\ln n - \alpha) \\ &\quad \in 2n \cdot \ln n + \Theta(n) \\ &\quad = 2 \ln 2 \cdot n \cdot \log_2(n) + \Theta(n) \end{aligned}$$

Analyse von Quicksort (4)

Satz 2.3.

Quicksort hat Zeitbedarf

- im besten Fall: $T_{\text{min}}(n) \in \Theta(n \cdot \log n)$
- im Mittel: $T_{\text{mit}}(n) \in \Theta(n \cdot \log n)$

Beweis:
Folgerung aus Lemmata 2.2 und 2.3.

Beachte

Quicksort benötigt im Mittel nur 39% mehr Vergleiche als im besten Fall.

Beweis:
Folgerung aus Lemmata 2.2 und 2.3 mit $2 \cdot \ln 2 \leq 1.39$.

2.2.4 Sortieren mit Bäumen: HeapSort

Vorbemerkungen

- Dieses Verfahren benutzt **nicht** die Strategie „Teilen + Beherrschen“.
- Zentrale Idee: Datenstruktur Heap

Definition 2.1
Sei $(M, \leq)$ eine geordnete Menge.
Ein $\max$-**Heap** ist ein binärer Baum mit

- jede Ebene - außer der tiefsten - ist voll besetzt,
- die tiefste Ebene ist ab links voll besetzt,
- die Knoten sind mit Elementen aus M markiert, so dass gilt:
  Ist m' Sohn von m, so ist $m \geq m'$.

Realisierung eines Heaps als Feld A mit $A[i] = a_i$

- Der Baum habe n Knoten $\leadsto$ $A = A[1, \ldots, n]$.
- a_i hat a_{2i} und a_{2i+1} als Söhne.
- a_k hat $a[k/2]$ als Vater.
- Das Teilfeld $A[li, \ldots, re]$ erfüllt die **Heap-Bedingung**, falls für alle $li \leq i \leq re$ gilt:
  - $* \; 2i \leq re \quad \leadsto \quad a_i \geq a_{2i}$
  - $* \; 2i+1 \leq re \quad \leadsto \quad a_i \geq a_{2i+1}$

Beachte

- Ist A ein $\max$-Heap, so ist a_1 das größte Element in A.
- Analog kann man einen $\min$-Heap definieren.

Entwurf von HeapSort (1)

Eingabe $A[1, \ldots, n]$
1. Phase: A in $\max$-Heap transformieren (bottom-up).
2. Phase: A so sortieren, dass $A[i] \leq A[i+1]$ gilt.

1. Phase: Heap aufbauen

- Sei $m = n$ div 2.
  $\rightsquigarrow A[m{+}1, \ldots, n]$ erfüllt die Heap-Bedingung,
  da für $m + 1 \leq i$ gilt: $2i > n$.
- Prozedur einfügen(A, i, n) erstellen, so dass gilt:
  Erfüllt $A[i+1, \ldots, n]$ die Heap-Bedingung vor dem Aufruf von einfügen(A, i, n), so erfüllt $A[i, \ldots, n]$ die Heap-Bedingung nach dem Aufruf.

Aufbau des Heaps

Algorithmus einfügen

einfügen$(A, li, re) =$ Ref: A
  $k \leftarrow 2 \cdot li$;
  if $k > re$ **then** **return**; **fi**
  $j \leftarrow k$;
  if $k < re$ **and** $A[k] < A[k+1]$
    then $j \leftarrow k+1$; % $A[j]$ ist der größere
  fi % Sohn von $A[li]$.
  if $A[li] < A[j]$
    then vertausche$(A[li], A[j])$;
      einfügen(A, j, re)
    else **return** **fi**

Zur Korrektheit
einfügen(A, li, re) lässt $A[li]$ so lange im Baum einsinken, bis $A[li, \ldots, re]$ die Heap-Bedingung erfüllt.

Zeitbedarf für BuildHeap(n, A)

Algorithmus zum Aufbau eines Heaps aus $A[1, \ldots, n]$

BuildHeap(n, A) = **for** $li = n$ div 2 **step** -1 **to** 1 **do**
    einfügen(A, li, n)
  od

Lemma 2.4

BuildHeap(n, A) hat Zeitbedarf im schlechtesten Fall:

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

Beweis:

- Zeit für Einfügen in Baum der Höhe h ist: $\Theta(h)$.
- Anzahl Teilbäume der Höhe h ist: $n/2^h$.

Also:

$$T(n) \in \Theta\left(\sum_{h=1}^{\log n} h \cdot n \cdot 2^{-h}\right) = \Theta\left(n \cdot \sum_{h=1}^{\log n} h \cdot 2^{-h}\right) = \Theta(n)$$

Beachte: $\sum_{h=1}^{\infty} h \cdot 2^{-h} < \infty$

Entwurf von HeapSort (2)

2. Phase: sortiertes Feld erstellen

$$\text{sortieren}(A, n) = \left[\begin{array}{ll} \text{vertausche}(A[1], A[n]) & (1) \\ \text{einfügen}(A, 1, n-1) & (2) \\ \text{sortieren}(A, n-1) & (3) \end{array}\right.$$

Es gilt:

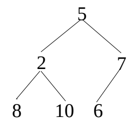
- nach (1): Das größte A-Element steht richtig auf Platz n.
- nach (2): $A[1, \ldots, n-1]$ erfüllt die Heap-Bedingung.
- nach (3): $A[1, \ldots, n]$ ist aufsteigend sortiert.

Algorithmus HeapSort

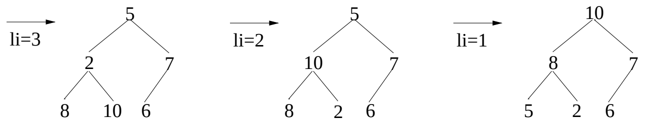
HeapSort(A, n) = Ref: A
  for $li = n$ div 2 **step** -1 **to** 1 **do**
    einfügen(A, li, n)
  od
  for $i = n$ **step** -1 **to** 2 **do**
    vertausche$(A[1], A[i])$
    einfügen$(A, 1, i-1)$
  od

HeapSort: Ein Beispiel

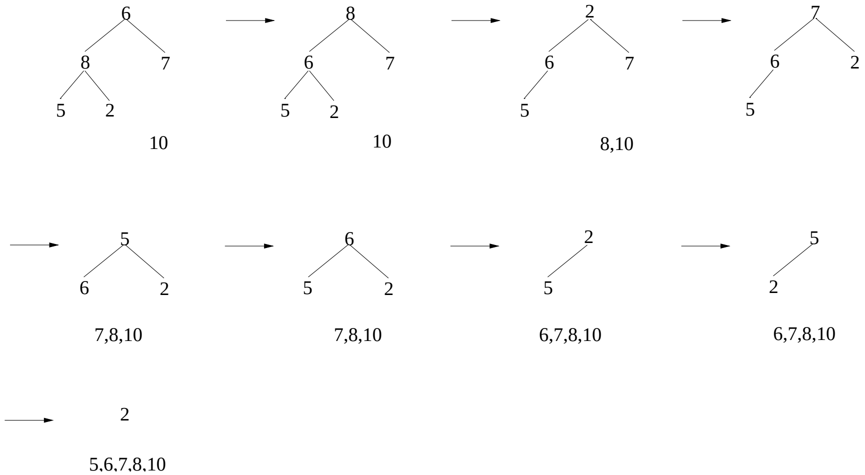
Eingabe: $A = (5, 2, 7, 8, 10, 6)$
also $n = 6$



1. Phase



2. Phase



Ausgabe: $A = (2, 5, 6, 7, 8, 10)$

Analyse von HeapSort (1)

Zeitbedarf

Im schlechtesten Fall gilt für den Zeitbedarf:

- in 1. Phase: $\Theta(n)$
- in 2. Phase: $\sum_{i=2}^{n} \Theta(\log_2 i) = \Theta(n \log n)$

Satz 2.4.

HeapSort hat Zeitbedarf im schlechtesten Fall:

$$T_{\max}(n) \in \Theta(n \log n)$$

Analyse von HeapSort (2)

Anzahl der Vergleiche $\mathbf{C(n)}$

- einfügen(A, li, re) benötigt $\leq 2t$ Vergleiche, wenn t die Tiefe von $A[li]$ im Baum ist.
- 1. Phase: Die Summe $V(t)$ der Tiefen im Binärbaum der Tiefe t ist:

  $$V(t) = t + 2V(t-1) \qquad V(0) = 0$$

  also: $V(t) \leq 2^{t+1}$

  Hat der Baum n Knoten, so gilt:

  $$2^{t+1} \leq n + 1$$

  $\rightsquigarrow$ 1. Phase benötigt $\leq 2n$ Vergleiche.
- 2. Phase: Maximal

  $$\sum_{i=2}^{n} 2 \cdot \log_2 i \leq 2 \cdot n \cdot \log_2 n \quad \text{Vergleiche}$$

Lemma 2.5

HeapSort benötigt im schlechtesten Fall

$$2 \cdot n \cdot \log n + \Theta(n)$$

Vergleiche.

2.2.5 Sortieren mit Hashing: BucketSort

Bisher: Sortieren nur durch Vergleichen der a_i
Jetzt: Die a_i vorsortieren mit Hashing

Eingabe: $A = (a_1, \ldots, a_n) \quad a_i \in \mathbb{R} \quad 0 \leq a_i < 1$
(durch Normieren stets erreichbar)
Ausgabe: A aufsteigend sortiert

Algorithmus

BucketSort$(A, n) =$
for $i = 0$ **to** $n - 1$ **do**
  Liste $B[i] \leftarrow$ *leer* **od**
for $i = 1$ **to** n **do**
  hänge $A[i]$ an $B[\lfloor n \cdot A[i] \rfloor]$ an **od**
for $i = 0$ **to** $n - 1$ **do**
  sortiere $B[i]$ nach Standard-Verfahren **od**
$A \leftarrow$ Listen-Inhalte von $B[0], \ldots, B[n-1]$

Platzbedarf zusätzlich: n Zeiger und n Plätze für die a_i

Zeitbedarf

Sei $b_i =$ Länge von $B[i]$.
Sortierung der $B[i]$ erfolge mit quadratischem Verfahren.
$\Rightarrow$ Zeitbedarf im schlechtesten Fall ($b_i = n$ für ein i):

$$T_{\mathsf{max}}(n) \in \Theta(n^2)$$

$\Rightarrow$ Zeitbedarf im Mittel (ohne Beweis)

$$T_{\mathsf{mit}}(n) \in O(n)$$

2.3 Bestimmung des $\mathbf{k}$-kleinsten Elements

Problem

Eingabe: • $A[1,\ldots,n]$
• k mit $1 \leq k \leq n$

Ausgabe: k-kleinstes Element in A

Beachte

- $k = 1 \rightsquigarrow$ Bestimmung des Minimums
- $k = n \rightsquigarrow$ Bestimmung des Maximums

$\Rightarrow$ Beides ist in Zeit $\Theta(n)$ mit $n-1$ Vergleichen machbar.

Idee für QuickSelect

Verwende Quicksort, aber betrachte jeweils nur das Teilfeld, das Platz k enthält.

QuickSelect

Algorithmus

```
QuickSelect(A, n, k) =
   Ref: A
   links ← 1;  rechts ← n;
   while links < rechts do
         wähle x ∈ A[links, ..., rechts]
         Partition(A, x, links, rechts, l, r)
         if k ≤ r then rechts ← r
                  else if k ≥ l then links ← l
                                else links ← rechts
                     fi fi
   od
   QuickSelect ← A[k]
```

Analyse von QuickSelect (1)

Platzbedarf zusätzlich zur Eingabe:

$$O(1)$$

Zeitbedarf

Beachte: $T(n) \in \Theta(C(n))$

a) Im schlechtesten Fall

$C_{\text{max}}(1) = 0$

$$C_{\text{max}}(n) = (n+1) + \max_{1 \leq i < n} C_{\text{max}}(i) = (n+1) + C_{\text{max}}(n-1)$$

also:

$$C_{\text{max}}(n) = \sum_{i=2}^{n+1} i \in \Theta(n^2)$$

$$T_{\text{max}}(n) \in \Theta(n^2)$$

Analyse von QuickSelect (2)

b) Im Mittel (Gleichverteilung)

Sei $C(n) = C_{\text{mit}}(n)$.

Für festes i gilt:

$\frac{1}{n}$ = Wahrscheinlichkeit für Zerlegung $i : n - i$

$k \leq i \rightsquigarrow$ neues Teilfeld hat Länge i.

$k > i \rightsquigarrow$ neues Teilfeld hat Länge $n - i$.

Also:

$$\begin{aligned} C(1) &= 0 \\ C(n) &= (n+1) + \frac{1}{n} \sum_{i=1}^{n-1} C(\max(i, n-i)) \\ &\leq (n+1) + \frac{2}{n} \sum_{i=n/2}^{n-1} C(i) \end{aligned}$$

Also (Induktion nach n):

$$C(n) \leq 4 \cdot n \quad \rightsquigarrow \quad T_{\text{mit}}(n) \in \Theta(n)$$

Satz 2.5.

QuickSelect hat Zeitbedarf

- $\Theta(n^2)$ im schlechtesten Fall und
- $\Theta(n)$ im Mittel.

2.4 Untere Schranken für Such- und Sortierprobleme

Probleme

- Suche in sortierten Feldern
- Sortieren von Feldern

Einschränkung für betrachtete Algorithmen
Feldelemente a_i dürfen nur in Vergleichen verwendet werden, z. B.:

- $a_i < a_j$ erlaubt
- $B[f(a_i)]$ nicht erlaubt

Also: BucketSort ist hier nicht erlaubt

Bezeichnung: Vergleichs-basierter Algorithmus

2.4.1 Technik der Entscheidungsbäume

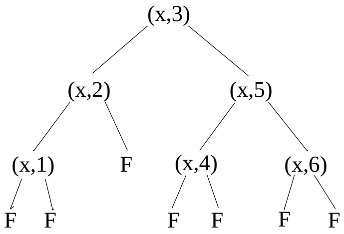
1. Problem
Suche nach x im aufsteigend sortierten Feld $A[1, \ldots, n]$

Entscheidungsbaum zum Vergleichs-basierten Algorithmus $\mathcal{A}$
Binärbaum mit

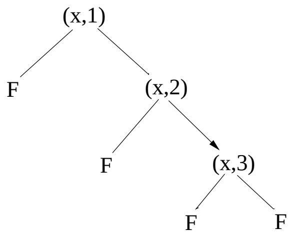
- Knoten (x, i) steht für Vergleich x mit a_i
- rechter Teilbaum steht für $x > a_i$
- linker Teilbaum steht für $x < a_i$
- Blatt F steht für x nicht gefunden
- Wurzel (x, i_0) steht für 1. Vergleich x mit a_{i_0}

Beispiele

a) binäre Suche, $n = 6$



b) lineare Suche, $n = 3$



Anzahl der Vergleiche im schlechtesten Fall

$$C_{\mathcal{A},\mathsf{max}}(n) = \text{Tiefe des Entscheidungsbaumes}$$

Untere Schranke für das Suchen in sortierten Feldern

Beachte

Der Entscheidungsbaum hat mindestens n innere Knoten (x, i).

Denn sonst: Es seien alle a_i verschieden und (x, k) fehle.
Eingabe: $x = a_k$
$\rightsquigarrow$ x wird nicht gefunden.
$\rightsquigarrow$ $\mathcal{A}$ ist inkorrekt.

Satz 2.6.

Jeder Vergleichs-basierte Algorithmus $\mathcal{A}$ zum Suchen in sortierten Feldern benötigt im schlechtesten Fall mindestens $\lceil \log_2(n+1) \rceil$ Vergleiche, also

$$T_{\max}(n) \in \Omega(\log n)$$

Zeitschritte.

Beweis:

Der Entscheidungsbaum hat mindestens n Knoten, also

$$\text{Tiefe} \geq \lceil \log_2(n+1) \rceil$$

$\rightsquigarrow C_{\mathcal{A},\max}(n) \geq \lceil \log_2(n+1) \rceil$

Untere Schranke für das Sortieren (1)

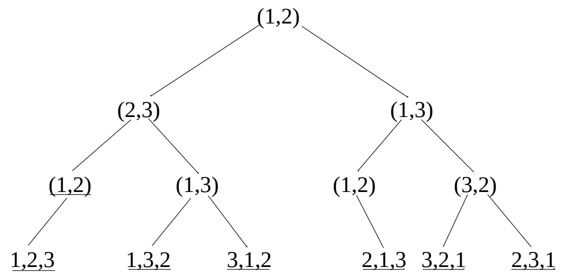
2. Problem
Sortieren eines Feldes $A[1, \ldots, n]$ mit $a_i = A[i]$

Entscheidungsbaum zum Vergleichs-basierten Algorithmus $\mathcal{A}$
Binärbaum mit

- Knoten (i, j) steht für Vergleich a_i mit a_j
- rechter Teilbaum steht für $a_i \geq a_j$
- linker Teilbaum steht für $a_i < a_j$
- Wurzel (i_0, j_0) steht für 1. Vergleich a_{i_0} mit a_{j_0}
- Blatt π steht für Lösungspermutation von $\{1, \ldots, n\}$

Beispiel

$\mathcal{A} =$ SelectionSort, $n = 3$



Pfad für Eingabe $a_3 < a_1 \leq a_2$

Untere Schranke für das Sortieren (2)

Beachte

Seien die a_i fest und paarweise verschieden.

- Es gibt $n!$ verschiedene Anordnungen der a_i.
- Jede dieser $n!$ Eingaben führt zu einem anderen Blatt.
- Der Entscheidungsbaum hat mindestens $n!$ Blätter, also:

$$\text{Tiefe} \geq \lceil \log_2(n!) \rceil$$

Satz 2.7.

Jeder Vergleichs-basierte Algorithmus $\mathcal{A}$ zum Sortieren von Feldern benötigt im schlechtesten Fall $C_{\text{max}}(n) \geq \lceil \log_2 n! \rceil$ Vergleiche, also

$$T_{\text{max}}(n) \in \Omega(n \cdot \log n)$$

Zeitschritte.

Beweis:

Da $\mathcal{A}$ korrekt ist, hat er einen Entscheidungsbaum mit einem Ast der Tiefe $\lceil \log_2 n! \rceil$, der im schlechtesten Fall durchlaufen wird. Also:

$$C_{\text{max}}(n) \geq \lceil \log_2 n! \rceil \quad \rightsquigarrow \quad T_{\text{max}}(n) \in \Omega(n \log n)$$

Beachte

Aus der Formel von Stirling folgt:

$$\lceil \log_2 n! \rceil \in n \cdot \log_2 n - n \cdot \log_2 e + \tfrac{1}{2} \log_2 n + O(1)$$

also: $$\log n! \in \Theta(n \cdot \log n)$$

2.4.2 Untere Schranken für das $\mathbf{k}$-kleinste Element

Lemma 2.6

Jeder Vergleichs-basierte Algorithmus benötigt zur Bestimmung des kleinsten Elements aus n paarweise verschiedenen Elementen im besten Fall mindestens $n-1$ Vergleiche.

Beweis:

Sei $\mathcal{A}$ ein Algorithmus zur Bestimmung des kleinsten Elements aus $a_1, a_2, \ldots, a_n$.

Vergleicht $\mathcal{A}$ a_i mit a_j und gilt $a_i < a_j$, so heißt a_i der Sieger des Vergleichs.

Gibt es weniger als $n-1$ Vergleiche in $\mathcal{A}$ bei Eingabe $(a_1, \ldots, a_n)$, so gibt es 2 Elemente a_p und a_q, die keinen Vergleich verloren haben.

Erklärt $\mathcal{A}$ a_p zum kleinsten Element, so betrachte die Eingabe $(a'_1, \ldots, a'_n)$ mit $a'_i = a_i$ für $i \neq q$ und $a'_q < a_p$. Dann verlaufen alle Vergleiche genau so wie bei Eingabe $(a_1, \ldots, a_n)$, es wird also a'_p als Sieger erklärt, obwohl $a'_q < a'_p$ gilt.

$\Rightarrow$ Also ist $\mathcal{A}$ nicht korrekt.

Die Orakel-Technik

Ziel

Sei $\mathcal{A}$ ein Vergleichs-basierter Algorithmus und $(a_1, \ldots, a_n)$ eine Eingabe.
Ein Orakel O steuert den Ausgang der Vergleiche so, dass $\mathcal{A}$ (zur Bestimmung des kleinsten Elements a_p) möglichst viele Vergleiche (mit a_p) macht.

Definition 2.2

eines Orakels O zu $\mathcal{A}$.
O bestimmt den Ausgang eines Vergleichs von a_i mit a_j als

- $a_i < a_j$, falls bisher a_i und a_j ungeschlagen sind und a_i mehr Vergleiche gewonnen hat als a_j.
- $a_i < a_j$, falls bisher a_i ungeschlagen ist, a_j aber nicht.
- beliebig sonst, aber mit der bisherigen Partialordnung $<$ verträglich.
  (z. B. $a_i < a_j$, $a_k < a_i \Rightarrow a_j < a_k$ kann nicht gelten,
  O setzt $a_k < a_j$.)

Beachte

Haben a_i und a_j bisher nie verloren, so sind die Definitionen von $a_i < a_j$ und $a_j < a_i$ mit dem bisherigen $<$ verträglich.

Eigenschaft des Orakels O

Betrachte die Entwicklung des folgenden Waldes W.

- Zeit $t = 0$: W besteht aus $T_1, \ldots, T_n$, wobei T_i nur aus a_i besteht.
- Zeit $t \to t+1$: Es werde in $\mathcal{A}$ a_i mit a_j verglichen.
  Sind a_i und a_j bisher ungeschlagen und setzt O $a_i < a_j$, so wird a_j direkt unter a_i gehängt. Sonst bleibt W ungeändert.

Lemma 2.7

Zu jedem Zeitpunkt t gilt:

- Die Wurzeln der Bäume sind genau die a_i, die bisher ungeschlagen sind.
- Hat die Wurzel a_i bisher p Vergleiche gewonnen, so enthält ihr Baum $\Delta(i) \leq 2^p$ Knoten.

Beweis: Induktion nach t.

$t = 0$: Die Behauptung gilt.

$t \to t+1$: a_j werde unter a_i gehängt.
Hat a_i p und a_j q Vergleiche gewonnen, so gilt $q \leq p$. Unter a_i hängen dann
$$\Delta(i) \leq 2^p + 2^q \leq 2^{p+1}$$
Knoten.

Bestimmung der beiden kleinsten Elemente (1)

Problem

Eingabe: Feld $A[1, \ldots, n]$

Ausgabe: Die beiden kleinsten Elemente in A

Satz 2.8.

Jeder Vergleichs-basierte Algorithmus $\mathcal{A}$, der die beiden kleinsten Elemente in $A[1, \ldots, n]$ bestimmt, benötigt im schlechtesten Fall mindestens

$$n + \lceil \log_2 n \rceil - 2$$

Vergleiche.

Beweis: Betrachte das Orakel O zu $\mathcal{A}$.

Sei a_k das kleinste Element und p die Anzahl der Vergleiche, an denen a_k beteiligt ist. Dann gilt beim Stopp

$$\Delta(k) = n \leq 2^p, \text{ also } p \geq \lceil \log_2 n \rceil$$

Für die Bestimmung des kleinsten unter den restlichen $n-1$ Elementen benötigt man mindestens $(n-1)-1$ Vergleiche zwischen diesen Elementen.

$\Rightarrow$ Also: $\mathcal{A}$ braucht mindestens

$$\lceil \log_2 n \rceil + (n-1) - 1 = n + \lceil \log_2 n \rceil - 2$$

Vergleiche.

Bestimmung der beiden kleinsten Elemente (2)

Lemma 2.8

Es gibt einen Vergleichs-basierten Algorithmus $\mathcal{A}$, der zur Bestimmung der beiden kleinsten Elemente in $A[1, \ldots, n]$ genau

$$n + \lceil \log_2 n \rceil - 2$$

Vergleiche benötigt.

Beweis: Konstruktion von $\mathcal{A}$

1. Phase: Starte k.o.-Turnier.
   $\Rightarrow$ Der Sieger ist das kleinste Element.
   $\Rightarrow$ Dies kostet $n-1$ Vergleiche.
   $\Rightarrow$ Es gibt $\lceil \log_2 n \rceil$ Verlierer gegen den Sieger.
2. Phase: Starte k.o.-Turnier unter den $\lceil \log_2 n \rceil$ Verlierern.
   $\Rightarrow$ Der Sieger ist das zweitkleinstes Element.
   $\Rightarrow$ Dies kostet $\lceil \log_2 n \rceil - 1$ Vergleiche.

$\rightsquigarrow C_{\text{max}}(n) = n - 2 + \lceil \log_2 n \rceil$

Bestimmung des $\mathbf{k}$-kleinsten Elements

Ziel

Untere Zeitschranken für die Bestimmung des k-kleinsten Elements in $A[1,\ldots,n]$.

Satz 2.9. [Hyafil]

Jeder Vergleichs-basierte Algorithmus zur Bestimmung des k-kleinsten Elements in $A[1,\ldots,n]$ benötigt mindestens

$$C(k,n) = n - k + (k-1) \cdot \left\lceil \log_2 \frac{n}{k-1} \right\rceil$$

Vergleiche.
ohne Beweis

3 Strategie Dynamisches Programmieren

Die zentrale Idee: Rekursion

- Reduktion des Problems der Größe n auf (i. Allg. mehrere) Probleme der Größe $< n$

Aber für die hier zu behandelnden Probleme gilt:

- Teilprobleme sind „überlappend“, nicht disjunkt wie bei „Teile und Beherrsche“.
- Rekursive Programmierung ist ineffizient/Mehrfachberechnungen
- bottom-up-Verfahren statt top-down wie bei „Teile und Beherrsche“

Anwendungsgebiet

Optimierungsprobleme

Voraussetzung

Bellman'sches Optimalitätsprinzip:

Die optimale Lösung eines Problems setzt sich zusammen aus optimalen Lösungen von (a priori nicht bekannten) Teilproblemen.

Verfahren

- Speichere Lösung für alle kleineren Probleme.
- Berechne daraus Lösung für die nächst größeren Probleme.

3.1 Multiplikation mehrerer Matrizen

Problem

Eingabe: n Matrizen M_i der Dimension $p_{i-1} \times p_i$.

Ausgabe: Klammerung, so dass das Produkt $M_1 \cdot M_2 \cdot \ldots \cdot M_n$ mit minimaler Anzahl von Element-Multiplikationen berechnet wird.

Beachte

Die Multiplikation auf Matrizen ist
- assoziativ, aber
- nicht kommutativ.

Beispiel

Vier Matrizen mit Dimension
$M_1(13 \times 5)$, $M_2(5 \times 89)$, $M_3(89 \times 3)$, $M_4(3 \times 34)$

Gesucht: optimale Klammerung für $M_1 \cdot M_2 \cdot M_3 \cdot M_4$

Klammerung	Kosten
$((M_1 \cdot M_2) \cdot M_3) \cdot M_4$	$13 \cdot 5 \cdot 89 + 13 \cdot 89 \cdot 3 + 13 \cdot 3 \cdot 34 = 10582$
$(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$	$13 \cdot 5 \cdot 89 + 89 \cdot 3 \cdot 34 + 13 \cdot 89 \cdot 34 = 54201$
$(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$	$5 \cdot 89 \cdot 3 + 13 \cdot 5 \cdot 3 + 13 \cdot 3 \cdot 34 = 2856$
$M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$	$5 \cdot 89 \cdot 3 + 5 \cdot 3 \cdot 34 + 13 \cdot 5 \cdot 34 = 4055$
$M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$	$89 \cdot 3 \cdot 34 + 5 \cdot 89 \cdot 34 + 13 \cdot 5 \cdot 34 = 26418$

Also: Die 3. Klammerung ist die beste.
Die 2. Klammerung ist die schlechteste.

Naiver Algorithmus für optimale Klammerung

Verfahren

- Berechne Kosten für jede Klammerung.
- Bestimme die beste unter ihnen.

Anzahl der möglichen Klammerungen $K(n)$ ist, falls Top-Klammerung nach j Matrizen:

$$K(n) = K(j) \cdot K(n-j)$$

Also:

$$K(1) = 1 \qquad K(n) = \sum_{j=1}^{n-1} K(j) \cdot K(n-j)$$

$$\begin{aligned} \rightsquigarrow K(n) &= \frac{1}{n}\binom{2(n-1)}{n-1} \approx \frac{4^n}{n^{3/2}} \\ K(n) &= C_{n-1} = (n-1)\text{-te Catalan-Zahl} \end{aligned}$$

Zeitbedarf des naiven Algorithmus

$$T(n) \in O\left(\frac{4^n}{n^{3/2}}\right) \qquad \text{exponentiell}$$

Beachte

- Die Matrizen M_i selbst sind für die Lösung uninteressant.
- Wichtig sind nur n und die p_i.

Entwurf von MatMultKlamm

Eingabe: n und $P[0,\ldots,n]$ mit $P[i] = p_i$.

Idee

Sei m_{ij} = Anzahl der Element-Multiplikationen für $M^{ij} = M_i \cdot \cdots \cdot M_j$ bei optimaler Klammerung, $1 \le i \le j \le n$.

Es gilt:

- M^{ij} hat Dimension $p_{i-1} \times p_j$.
- $m_{ij} = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{m_{ik} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j\} & i < j \end{cases}$

Sei k_{ij} das k, bei dem Minimum angenommen wird.

Ergebnis

$\Rightarrow$ m_{1n} gibt die Kosten der optimalen Klammerung an.
Optimale Klammerung ist aus den k-Werten berechenbar.

Berechnung der m- und k-Werte für
$l = j - i = 1, 2, \ldots, n-1$
also Diagnonale für Diagonale, von unten nach oben.
Das Ergebnis steht ganz oben rechts.

Beispiel für MatMultKlamm

Sei $n = 6$ und $P = (6, 12, 20, 3, 10, 5, 18)$.

Tabelle der m/k-Werte

i \ j	1	2	3	4	5	6
1	0	1440/1	936/1	1116/3	1176/3	1680/3
2		0	720/2	1080/3	1050/3	1788/3
3			0	600/3	450/3	1500/3
4				0	150/4	420/5
5					0	900/5
6						0

Optimale Klammerung ist:

$$\begin{aligned} M^{1,n} &= M^{1,3} \cdot M^{4,6} = (M_1 \cdot M^{2,3}) \cdot (M^{4,5} \cdot M_6) \\ &= (M_1 \cdot (M_2 \cdot M_3)) \cdot ((M_4 \cdot M_5) \cdot M_6) \end{aligned}$$

Kosten der optimalen Klammerung: 1680

Beachte

Bellman's Optimalitätsprinzip gilt:

> Jedes Teilprodukt im optimal geklammerten Produkt ist optimal geklammert.

Algorithmus für MatMultKlamm

Algorithmus

Eingabe: $n, P[0, \ldots, n]$

Ausgabe: $n \times n$-Matrizen M und K
(Diese Matrizen realisieren die Tabellen der m- und der k-Werte.)

MatMultKlamm(n, P, M, K)=

for $i := 1$ **to** n **do**
    $M[i, i] \leftarrow 0$
od
for $l := 1$ **to** $n - 1$ **do**
    for $i = 1$ **to** $n - l$ **do**
        $j \leftarrow i + l;$
        $p \leftarrow \infty;$
        for $r := i$ **to** j **do**
            $q \leftarrow M[i, r] + M[r + 1, j] + P[i - 1] \cdot P[r] \cdot P[j]$
            if $q < p$
              then
                    $p \leftarrow q;\ \ K[i, j] \leftarrow r$
            fi
        od
        $M[i, j] \leftarrow p$
    od
od

Algorithmus für MatMultKlamm (Forts.)

Zeitbedarf

im besten und schlechtesten Fall

$$T(n) \in O\left(\sum_{l=1}^{n-1} \sum_{i=1}^{n-l} \sum_{r=i}^{i+l} 1\right) = O(n^3)$$

Berechnung von $O(n^2)$ Einträgen in M, K mit $O(n)$ Zeitbedarf pro Eintrag.

Extraktion des Ergebnisses aus $\mathbf{M}, \mathbf{K}$

Seien M und K in MatMultKlamm berechnet:

$$m_{ij} = M[i,j] \quad \text{und} \quad k_{ij} = K[i,j].$$

Es gilt:

- m_{1n} sind die Kosten der optimalen Klammerung.
- k_{1n} liefert die optimale Top-Klammerung:

$$M^{1n} = M^{1,k} \cdot M^{k+1,n} \text{ mit } k = k_{1n}$$

allgemein:

- k_{ij} liefert die optimale Top-Klammerung von M^{ij}.

Also:

Die Kosten der optimalen Klammerung und die optimale Klammerung selbst sind aus M und K in Zeit $O(n)$ berechenbar. M und K sind in Zeit $O(n^3)$ aus P berechenbar.

3.2 Kürzeste Wege in Graphen

Problem

Eingabe: Gewichteter Graph $G = (V, E, \gamma)$ mit $V = \{1.\dots,n\}$, $E \subseteq V \times V$ und $\gamma : E \to \mathbb{R}_+$

Ausgabe: $D = (d_{ij})$ mit $d_{ij} =$ kürzester Weg von i nach j

Idee (Warshall-Algorithmus)

Sei $a_{ij}^k =$ kürzester Weg von i nach j, der nur Zwischenknoten $\leq k$ benutzt.

Kostenmatrix $C = (c_{ij})$

$$c_{ij} = \begin{cases} \gamma((i,j)) & \text{falls } (i,j) \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Es gilt:

$$\begin{aligned} a_{ij}^0 &= c_{ij} \\ a_{ij}^k &= \min\{a_{ij}^{k-1}, a_{ik}^{k-1} + a_{kj}^{k-1}\} \text{ für } k \geq 1 \end{aligned}$$

Beachte zur Korrektheit:

- Jeder Weg hat positive Länge, also besucht ein kürzester Weg keinen Knoten mehrfach.
- Es gilt Bellmans Optimalitätsprinzip.

Algorithmus für kürzeste Wege

Algorithmus (informell)

Man berechne analog zum letzten Abschnitt 2 Tabellen.
Für $k = 1, \ldots, n$ berechne für alle i, j

a_{ij}^k wie oben

$$b_{ij}^k = \begin{cases} 1, & a_{ij}^k = a_{ij}^{k-1} \\ 0, & \text{sonst} \end{cases}$$

Daraus lässt sich wieder berechnen:

- die Länge des kürzesten Weges
- der kürzeste (bzw. ein kürzester) Weg selbst

Zeitbedarf

- $O(n^3)$ für die Tabellen A und B
- $O(n)$ für den kürzesten Weg

Transitive Hülle einer Relation

Definition 3.1

Sei $\rightarrow \subseteq V \times V$ eine Relation auf V.

Dann ist die **transitive Hülle** $\xrightarrow{+}$ von $\rightarrow$ definiert durch

$u \xrightarrow{+} v$ gdw es gibt einen $\rightarrow$-Weg $u \rightarrow \circ \cdots \circ \rightarrow v$ von u nach v.

Problem

Eingabe: Relation $\rightarrow$ auf V

Ausgabe: Die transitive Hülle $\xrightarrow{+}$ zu $\rightarrow$

Idee

Setze $E = \rightarrow$ und betrachte $G = (V, E)$.
Es gilt: $u \xrightarrow{+} v$ gdw es gibt in G einen Weg von u nach v.

Übung

Man formuliere einen Algorithmus mit Zeitbedarf $\Theta(\|V\|^3)$ zur Berechnung von $\xrightarrow{+}$ aus $\rightarrow$.

3.3 Optimale binäre Suchbäume

Definition 3.2

Ein **binärer Suchbaum** für M ist ein binärer Baum mit Knoten(-Beschriftungen) aus M, so dass für jeden Knoten m gilt:
Alle Knoten im linken (rechten) Teilbaum sind $< m (> m)$.

$\Rightarrow$ Durchlauf in in-order liefert gespeicherte Werte aufsteigend sortiert.

Seien
- $a_1 < a_2 < \cdots < a_n$ die zu speichernden Werte
- p_i = Zugriffshäufigkeit auf a_i
- t_i = Tiefe von a_i im Suchbaum

$\Rightarrow (1 + t_i)$ = Anzahl der benötigten Vergleiche beim Zugriff auf a_i

Problem

Eingabe: $P = (p_i), A = (a_i), i = 1, \ldots, n$

Ausgabe: Optimaler Suchbaum für $a_1, \ldots, a_n$, d. h.

$$\sum_{i=1}^{n} (1 + t_i) p_i \quad \text{ist minimal.}$$

Gesucht sind also die t_i.

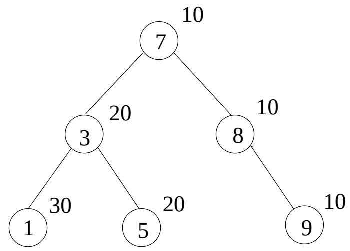
Beachte

Bellman's Optimalitätsprinzip gilt:
Jeder Teilbaum eines optimalen Suchbaumes ist optimaler Suchbaum.

Beispiel

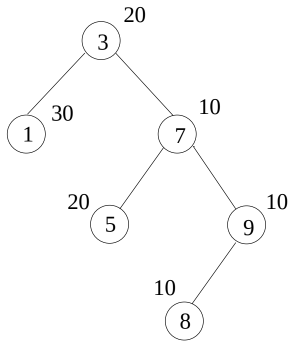
Sei $n = 6$, $P = (30, 20, 20, 10, 10, 10)$,
$A = (1, 3, 5, 7, 8, 9)$.

T_1:



$\Rightarrow 3 \cdot 30 + 2 \cdot 20 + 3 \cdot 20 + 1 \cdot 10 + 2 \cdot 10 + 3 \cdot 10 = 250$

T_2:



$\Rightarrow 2 \cdot 30 + 1 \cdot 20 + 3 \cdot 20 + 2 \cdot 10 + 4 \cdot 10 + 3 \cdot 10 = 230$

Also: T_2 ist besser als T_1.
$\Rightarrow$ Was ist ein optimaler Suchbaum zu A und P?

Entwurf für OptSuchbaum

Idee

Sei für $\{a_i, \ldots, a_j\}, i \leq j$,

$$w_{i,j} = \sum_{r=i}^{j} p_r \qquad g_{i,j} = \sum_{r=i}^{j} (1 + t_r) p_r$$

$\Rightarrow$ $g_{i,j}$ = Gewicht des optimalen Suchbaums für $a_i, \ldots, a_j$

Es gilt:

$$g_{i,j} = \begin{cases} 0 & i > j \\ w_{i,j} + \min\limits_{i \leq k \leq j} (g_{i,k-1} + g_{k+1,j}) & i \leq j \end{cases}$$

Beachte zur Korrektheit

Sei a_k Wurzel des optimalen Suchbaums für $\{a_i, \ldots, a_j\}$.

Es gilt: $g_{i,j} = \underbrace{p_k}_{\text{Wurzel}} + \underbrace{(w_{i,k-1} + g_{i,k-1})}_{\text{linken Teilbaum}} + \underbrace{(w_{k+1,j} + g_{k+1,j})}_{\text{rechten Teilbaum}}$

Gewicht für

Es gilt: $\quad p_k + w_{i,k-1} + w_{k+1,j} = w_{i,j}$

$\Rightarrow \quad g_{i,j} = w_{i,j} + g_{i,k-1} + g_{k+1,j}$

Entwurf für OptSuchbaum (Forts.)

Beachte

Analogie zu MatMultklamm

Berechnung der Tabelle:

$$G = (g_{i,j}) \qquad K = (k_{i,j})$$

für $l = j - i = 1, 2, \ldots, n - 1$.

Also Diagonale für Diagonale von unten nach rechts oben.

Zeitbedarf

$$T(n) \in O(n^3)$$

Beachte die Effizienzverbesserung in Abschnitt 3.6.

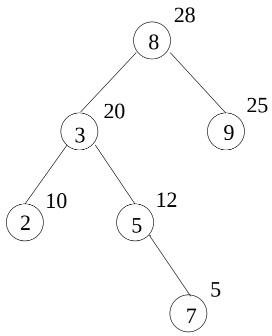
Beispiel für OptSuchbaum

Sei $n = 6, \quad P = (10, 20, 12, 5, 28, 25)$
$\quad A = (2, 3, 5, 7, 8, 9)$

Tabelle der g/k-Werte

i \ j	1	2	3	4	5	6
1	$10/1$	$40/2$	$64/2$	$79/2$	$152/2$	$204/5$
2		$20/2$	$44/2$	$59/2$	$123/3$	$174/5$
3			$12/3$	$22/3$	$67/5$	$117/5$
4				$5/4$	$38/5$	$88/5$
5					$28/5$	$78/5$
6						$25/6$

Optimaler Baum (Teilbäume mit Gewicht)



Beachte

Die p_i geben die relativen Häufigkeiten an.
Normierung auf $\sum p_i = 1$ liefert Wahrscheinlichkeiten.

3.4 Das Problem des Handlungsreisenden

Problem TSP

Eingabe: $n \times n$ Matrix $\quad D = (d_{ij})$ mit $d_{ij} \in \mathbb{R}_+ \cup \{\infty\}$
Entfernungsmatrix zu n Städten

Ausgabe: Kürzeste Rundreise mit
Rundreise = Permutation π
Gesucht also π mit

$$\sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \qquad \text{ist minimal.}$$

Beachte

- D beschreibt einen Graph $G = (V, E)$ mit
  - $V = \{1, \ldots, n\}$,
  - $E = \{(i, j) \mid d_{ij} \in \mathbb{R}_+\}$

  Betrachtet werden Rundreisen in G.
- Wegen $d_{ij} > 0$ gilt:
  - Jeder Weg hat positive Länge (Gewicht).
  - Optimale Rundreise besucht jeden Knoten genau einmal.

Das Problem des Handlungsreisenden (Forts.)

Beachte

Es gibt viele Anwendungen des Problems, z. B.:

- Roboterarm plant Rundreise auf Platine.
- Umrüstzeiten für Maschinen/Anlagen.
- Fahrzeuge/Schiffe planen Rundreisen.

Beachte

Außer $d_{ij} > 0$ gibt es keine Voraussetzungen an D;
z. B. wird **nicht** gefordert:

- Symmetrie: $d_{ij} = d_{ji}$
- Dreiechsungleichung: $d_{ij} + d_{jk} \geq d_{ik}$

Entwurf eines Algorithmus zum TSP

Beachte

- Es gibt $n! \approx \left(\frac{n}{e}\right)^n = 2^{n \cdot (\log n - \log e)}$ Rundreisen.
  $\Rightarrow$ Alle auszuprobieren ist zu teuer.
- TSP ist NP-vollständig.
  $\Rightarrow$ Jeder Algorithmus ist exponentiell (vermutlich !).
- Es geht aber schneller als $O(2^{n \cdot \log n})$.

Idee

Für $i \notin S \subseteq S_0 = \{2, \ldots, n\}$ sei

$g(i, S)$ = Länge des kürzesten Weges ab i durch S nach 1

Berechne alle $g(i, S)$.

Es gilt: $g(1, S_0)$ = Länge der kürzesten Rundreise.

Beachte

Bellman's Optimalitätsprinzip:
Sei $w \in V^*$ ein kürzester Weg ab i durch S nach 1 und $w = iju$.
Dann ist ju ein kürzester Weg ab j durch $S \setminus \{j\}$ nach 1.

Entwurf eines Algorithmus zum TSP (Forts.)

Berechnung der $g(i,S)$:
Für $\|S\| = 0, 1, \ldots, n-2$ berechne alle $g(i,S)$
und daraus $g(1, S_0)$:

$$g(i,S) \quad = \quad \begin{cases} d_{i1} & \text{für } S = \varnothing \\ \min\limits_{j \in S}\{d_{ij} + g(j, S \setminus \{j\})\} & \text{für } S \neq \varnothing \end{cases}$$

$$g(1, S_0) \quad = \quad \min_{j \in S_0}\{d_{1j} + g(j, S_0 \setminus \{j\})\}$$

Ein Beispiel für das TSP

Sei $n = 4$ und

$$D = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

$\|S\| = 0$ $\quad g(2,\varnothing) = 5, \quad g(3,\varnothing) = 6, \quad g(4,\varnothing) = 8$

$\|S\| = 1$
$$\begin{aligned} g(2,\{3\}) &= d_{23} + g(3,\varnothing) = 9 + 6 = 15 \\ g(2,\{4\}) &= d_{24} + g(4,\varnothing) = 10 + 8 = 18 \\ g(3,\{2\}) &= d_{32} + g(2,\varnothing) = 13 + 5 = 18 \\ g(3,\{4\}) &= d_{34} + g(4,\varnothing) = 12 + 8 = 20 \\ g(4,\{2\}) &= d_{42} + g(2,\varnothing) = 8 + 5 = 13 \\ g(4,\{3\}) &= d_{43} + g(3,\varnothing) = 9 + 6 = 15 \end{aligned}$$

$\|S\| = 2$
$$\begin{aligned} & g(2,\{3,4\}) = \min\{9 + 20, 10 + 15\} = 25 \\ & \Rightarrow j = 4 \\ & g(3,\{2,4\}) = \min\{13 + 18, 12 + 13\} = 25 \\ & \Rightarrow j = 4 \\ & g(4,\{2,3\}) = \min\{8 + 15, 9 + 18\} = 23 \\ & \Rightarrow j = 2 \end{aligned}$$

$\|S\| = 3$
$$\begin{aligned} g(1,\{2,3,4\}) &= \min\{10 + 25, 15 + 25, 20 + 23\} \\ &= 35 \\ \Rightarrow j = 2 & \end{aligned}$$

Ein Beispiel für das TSP (Forts.)

Also

Eine optimale Rundreise ist

$$\pi = (1, 2, 4, 3).$$

Die Länge von π ist 35.

Beachte zur Berechnung der Rundreise

- Bei jeder Berechnung von $g(i, S)$ kann man das j speichern, bei dem das Minimum angenommen wird.
- Dies liefert wie folgt die Konstruktion der Rundreise π:
  - Start bei 1,
  - dann j aus $g(1, S_0)$ berechnen,
  - dann π rekursiv weiter berechnen.

Ein Algorithmus für das TSP

Algorithmus für g

(für Länge einer kürzesten Rundreise)
Erweiterung zur Konstruktion der Rundreise ist einfach

g-Werte(n, D)=

$S_0 := \{2, \ldots, n\}$
for $i := 2$ **to** n **do**
  $g(i, \varnothing) \leftarrow d_{i1}$
od
for $k = 1$ **to** $n - 2$ **do**
  for alle $S \subseteq S_0, \|S\| = k$ **do**
    for alle $i \in S_0 - S$ **do**
      $g(i, S) \leftarrow \min_{j \in S}\{d_{ij} + g(j, S \setminus \{j\})\}$
    od
  od
od
$g(1, S_0) \leftarrow \min_{j \in S_0}\{d_{1j} + g(j, S_0 \setminus \{j\})\}$

Analyse von g-Werte

Zeitbedarf für

- (3), (12): $O(n)$
- (8): $O(k)$ pro Durchlauf

$$
\begin{aligned}
T(n) &\in O(n) + O\left(\sum_{k=1}^{n-2} \binom{n-1}{k}(n-1-k)\cdot k\right)\\
&= O(n) + O\left(\sum_{k=1}^{n-2}(n-1)(n-2)\binom{n-3}{k-1}\right)\\
&= O(n) + O\left(n^2 \cdot \sum_{k=0}^{n-3}\binom{n-3}{k-1}\right)\\
&= O(n) + O(n^2 \cdot 2^{n-3})\\
&= O(n^2 \cdot 2^n)
\end{aligned}
$$

$$\Rightarrow T(n) \in O(n^2 \cdot 2^n)$$

Beachte

Dieser Algorithmus ist sehr viel effizienter als die Berechnung aller $n!$ Rundreisen. Es gilt nämlich

$$n^2 \cdot 2^n = 2^{n+2\log n} \ll 2^{n\log n - n\log e} \approx n!$$

3.5 Das 0/1-Rucksackproblem

Problem RSP

verbal:

Eingabe: n Objekte mit Profiten p_i und Gewichten w_i, Rucksack der Kapazität k

Ausgabe: Teilmenge der n Objekte mit maximalem Profit, die in den Rucksack passt

formal:

Eingabe: $P = (p_i),\ W = (w_i)$ mit $p_i, w_i \in \mathbb{N}_+$, $i = 1, \ldots, n$, und $k \in \mathbb{N}_+$

Ausgabe: Vektor $x = (x_i)$ mit
- $x_i \in \{0,1\}, i = 1, \ldots, n$,
- $\sum_{i=1}^{n} p_i x_i$ ist maximal,
- $\sum_{i=1}^{n} w_i x_i \le k$

Anwendungen

Es gibt viele Anwendungen dieses Problems, z. B.:

- n Aufträge mit Profiten p_i, Zeitbedarf w_i, Gesamtzeit k
- Verschnittproblem: Rohr der Länge k, n kurze Rohre der Länge w_i mit Profiten p_i
- ...

Gesucht ist stets ein zulässiger Vektor x, der den Profit maximiert.

Naive Ansätze für das Rucksackproblem (RSP)

Ansätze Greedy-Strategie

1) Sortiere die Objekte nach „Güte“ und
2) nimm die „besten“, bis der Rucksack voll ist.

„Gütekriterien“

a) Nimm die mit großem p_i zuerst.
b) Nimm die mit kleinem w_i zuerst.
c) Nimm die mit großem p_i/w_i zuerst.

$\Rightarrow$ Zeitbedarf bei diesen Ansätzen:

$$O(n \cdot \log n)$$

Aber: Die Algorithmen sind nicht korrekt!!

Beispiel $n = 3$

1. $P = (8, 5, 4), \quad W = (3, 2, 2), \quad k = 4$
   Optimal ist: Nimm 2. und 3. Objekt. $\Rightarrow$ Dies liefert Profit 9.
   a) und c) nehmen nur 1. Objekt. $\Rightarrow$ Dies liefert Profit 8.

2. $P = (10, 5, 4), \quad W = (3, 2, 1), \quad k = 3$
   Optimal ist: Nimm 1. Objekt. $\Rightarrow$ Dies liefert Profit 10.
   b) nimmt 2. und 3. Objekt. $\Rightarrow$ Dies liefert Profit 9.

Naive Ansätze für das Rucksackproblem (RSP) (Forts.)

Beachte

Das Problem ist NP-vollständig.
$\Rightarrow$ Jeder korrekte Algorithmus ist (vermutlich) exponentiell.

Beachte

Es gibt 2^n 0/1-Vektoren x.
Alle zu probieren, kostet Zeit $O(n \cdot 2^n)$.
Geht es auch schneller?

Entwurf eines Algorithmus zum RSP

Idee

Berechne

$w(i,g)=$ maximaler Profit beim Packen des Rucksacks mit Objekten aus $\{1,...,i\}$ bei Kapazitätsgrenze g

Dann ist $w(n,k)$ der gesuchte Wert.

Beachte

Bellman's Optimalitätsprinzip:

- Ist $O_i \subseteq \{1,...,i\}$ optimale Packung bei Kapazitätsgrenze g, und
- ist $i \in O_i$,

so ist $O_i \setminus \{i\}$ optimale Packung aus $\{1,...,i-1\}$ bei Kapazitätsgrenze $g - w_i$.

Entwurf eines Algorithmus zum RSP (Forts.)

Berechnung von $w(i,g)$

$$w(i,g) = \begin{cases} 0 & i = 0 \\ w(i-1,g) & i > 0, \quad g < w_i \\ \max\{w(i-1,g), \\ \qquad w(i-1,g-w_i) + p_i\} & \text{sonst} \end{cases}$$

Berechnung der $w(i,g)$ für $i = 0,1,...,n$ für alle $g \leq k$.

Speichere parallel zu $w(i,g)$ den Wert $b(i,g)$ mit

$$b(i,g) = \begin{cases} 0, & \text{falls Element } i \text{ nicht genommen wird,} \\ & \text{d. h. } w(i,g) = w(i-1,g) \\ 1, & \text{sonst} \end{cases}$$

Dies erlaubt die Berechnung der optimalen Packung.

Algorithmus für das RSP

Beispiel

Sei $n = 4,\ P = (1,7,2,5),\ W = (2,5,3,4),\ k = 10.$

Tabelle der $w(i, g)$ - Werte

i \ g	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1
2	0	0	1	1	1	7	7	8	8	8	8
3	0	0	1	2	2	7	7	8	9	9	10
4	0	0	1	2	5	7	7	8	9	12	12

$\Rightarrow$ Der optimale Profit ist $w(4, 10) = 12$.
Die optimale Packung ist $O_4 = \{2, 4\}$, also $x = (0, 1, 0, 1)$.

Der Algorithmus OptRSP

Beachte zum Speicherplatz

- Für $w(i, g)$ benötigt man die $w(j, g')$ nur für $j = i - 1$.
  $\Rightarrow$ Speicherplatz: $O(2 \cdot k) = O(k)$
- Für die Konstruktion der optimalen Packung benötigt man die ganze Tabelle.
  $\Rightarrow$ Speicherplatz: $O(n \cdot k)$

Algorithmus

```
OptRSP(n, P, W)=
    for i = 0 to n do
        for g = 0 to k do
            w(i, g) ← ···
            b(i, g) ← ···
        od
    od
```

Zeitbedarf

$O(n \cdot k)$

Der Algorithmus OptRSP (Forts.)

Beachte

Größe $|e|$ der Eingabe e ist:

$$|e| = \log k + \sum_{i=1}^{n} \log p_i + \sum_{i=1}^{n} \log w_i$$

$\Rightarrow$ Zeitbedarf ist **exponentiell** in $|e|$.

Verbesserung des Algorithmus (Idee)

Die $w_i(g) = w(i, g)$ sind Treppenfunktionen in g.
$\Rightarrow$ Berechne die w_i nur an den Sprungstellen.

3.6 Effizienz-Verbesserungen

Beachte

Bei einigen Algorithmen wird in der innersten Schleife ein $\min$- bzw. $\max$-Wert bestimmt, z. B.:

- MatMultKlamm (Multiplikation vieler Matrizen)
  $m_{i,j} = \min_{i \leq k < j} \{m_{i,k} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j\}$
  $k_{i,j} =$ das k, bei dem das Minimum angenommen wird
- OptSuchbaum (optimaler binärer Suchbaum)
  $t_{i,j} = w_{i,j} + \min_{i \leq k \leq j} \{t_{i,k-1} + t_{k+1,j}\}$
  $k_{i,j} =$ das k, bei dem das Minimum angenommen wird

Kann man das Suchintervall für k verkleinern?
Beachte: In beiden Fällen sind die $k_{i,j}$ in i und j wachsend.

Vermutung

Es gilt stets:

$$k_{i,j-1} \leq k_{i,j} \leq k_{i+1,j}$$

$\Rightarrow$ Dies ist im Allgemeinen falsch!
Dies gilt aber manchmal! Es ist für jede Anwendung zu testen.
Es liefert dann eine Effizienzverbesserung.

Einschränkung des Suchraums

Für SuchOpt ist die Einschränkung $k_{i,j-1} \leq k_{i,j} \leq k_{i+1,j}$ korrekt:

Lemma 3.1 Sei (n, P, A) eine Eingabe für SuchOpt.
Ist T_1 ein optimaler Suchbaum für $\{a_i, \ldots, a_{j-1}\}$ mit Wurzel a_l und T_2 ein optimaler Suchbaum für $\{a_{i+1}, \ldots, a_j\}$ mit Wurzel a_r, dann gibt es einen optimalen Suchbaum T für $\{a_i, \ldots, a_j\}$ mit Wurzel a_k und $l \leq k \leq r$.

Beweis: Übung

Satz 3.1.

Man kann zur Eingabe (n, P, A) in Zeit $O(n^2)$ einen optimalen Suchbaum bestimmen.

Beweis:
SuchOpt ist von gleicher Struktur wie MatMultKlamm.
Einschränkung der innersten Schleife auf $r = k_{i,j-1}$ bis $r = k_{i+1,j}$ liefert den folgenden Zeitbedarf:

$$\begin{aligned} T(n) &\in O(n) + O\left(\sum_{l=1}^{n-1}\sum_{i=1}^{n-l}(1 + k_{i+1,l+i} - k_{i,i+l-1})\right) \\ &= O(n) + O\left(\sum_{l=1}^{n-1}(n - l + k_{n-l+1,n} - k_{1,l})\right) \\ &= O(n) + O\left(\sum_{l=1}^{n-1} 2n\right) = O(n) + O(n^2) = O(n^2) \end{aligned}$$

4 Die gierige Methode (The greedy method)

Die Methode

Eingabe:
- Menge M
- Folgen $y \in M^*$ sind zulässig/unzulässig
- $\varphi : M^* \to \mathbb{N}$ Bewertung der Folgen

Ausgabe: zulässiges $y \in M^*$ mit $\varphi(y)$ optimal (minimal, maximal)

Voraussetzung: Jede Anfangsfolge einer optimalen Folge ist optimal. (*)

Strategie: Berechne y schrittweise:
$y \rightsquigarrow y \cdot m$, so dass
- $y \cdot m$ zulässig ist,
- $y \cdot m$ beste Verlängerung von y um ein Glied ist.

Beachte

a) Vergleich mit Dynamischer Programmierung:
Voraussetzung hier stärker als Bellman's Optimalitätskriterium
$\Rightarrow$ kleinerer Suchraum
schnellere Algorithmen zu erwarten

b) Voraussetzung impliziert:
„lokale Optimierung führt zu globalem Optimum“

4.1 Optimale Speicherung auf Bändern

Problem

Eingabe: n Datensätze der Länge $l_1, \ldots, l_n$
Wahrscheinlichkeit auf Zugriff gleichverteilt

Ausgabe: Reihenfolge $\pi = (i_1, \ldots, i_n)$, so dass die mittlere Zugriffszeit minimal ist.

Mittlere Zugriffszeit (mean retrieval time)

$$\text{Zugriffszeit auf } j: \qquad t_j = t_j(\pi) = \sum_{k=1}^{j} l_{i_k}$$

$$\mathrm{MTR}(\pi) = \frac{1}{n} \sum_{j=1}^{n} t_j$$

Optimale Speicherung auf Bändern (Forts.)

Verfahren

(i_1): Wähle i_1 so, dass l_{i_1} minimal ist.

$(i_1, \ldots, i_k) \rightsquigarrow (i_1, \ldots, i_k, i_{k+1})$:
Wähle i_{k+1} so, dass i_{k+1} minimal unter den restlichen i_j ist.

Algorithmus

- Sortiere $(l_1, \ldots, l_n)$ aufsteigend zu $(l_{i_1}, \ldots, l_{i_n})$.
- Setze $\pi = (i_1, \ldots, i_n)$.

Zeitbedarf

$$T(n) \in O(n \cdot \log n)$$

Beachte zur Korrektheit

Voraussetzung (*) ist erfüllt.

4.2 Das Bruchteil-Rucksackproblem

Problem

Eingabe:
- n Objekte mit Profiten p_i und Gewichten w_i
- Rucksack der Kapazität K

Es gilt $K, p_i, w_i > 0$.

Ausgabe: $x = (x_1, \ldots, x_n)$, $\quad 0 \leq x_i \leq 1$, mit

- $p(x) = \sum_{i=1}^{n} p_i x_i \quad$ maximal
- $w(x) = \sum_{i=1}^{n} w_i x_i \leq K$

Vergleich mit 0/1-Rucksackproblem

Hier $0 \leq x_i \leq 1$ statt $x_i \in \{0, 1\}$ erlaubt.

Lösungsidee

Nimm die Objekte mit größtem Wirkungsgrad p_i/w_i zuerst.

Voraussetzung ab jetzt

$$p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n \qquad (+)$$

Sei $q_i = p_i/w_i$.

Definition 4.1

$(x_1, \ldots, x_m)$, $m \leq n$, heißt **zulässig**, falls

$$\sum_{i=1}^{m} w_i x_i \leq K$$

Das Bruchteil-Rucksackproblem (Forts.)

Lemma 4.1

Es gibt eine optimale Lösung $x = (x_1, \ldots, x_n)$ des Bruchteil-Rucksackproblems der Form

a) $x_i = 1 \quad$ für $\quad i = 1, \ldots, n$, falls $\sum_{i=1}^{n} w_i \leq K \qquad$ oder

b) $x_i = 1 \quad$ für $\quad i = 1, \ldots, j-1$
$x_i = 0 \quad$ für $\quad i = j+1, \ldots, n$
$x_j = \alpha$, $\quad$ falls $\sum_{i=1}^{j-1} w_i + \alpha \cdot w_j = K, \quad 0 \leq \alpha < 1$

Beachte

- Ist $\sum_{i=1}^{n} w_i > K$, so gibt es j und α wie in b).
- Für die optimale Lösung x aus dem Lemma gilt Bellman's Optimalitätsprinzip: „Jede Anfangsfolge von x ist optimal."

Das Bruchteil-Rucksackproblem (Forts.)

Beweis von Lemma 4.1

(i) Ist $\sum_{i=1}^{n} w_i \leq K$, so ist das x aus a) eine optimale Lösung.

(ii) Sonst, falls also $\sum_{i=1}^{n} w_i > K$, betrachte das x aus b).

Dann gilt für x: $w(x) \leq K$
Sei $OPT = p(y)$ der p-Wert einer optimalen Lösung y.
Zu zeigen ist: $p(x) = OPT$.

Annahme: $p(x) < OPT$, also $x \neq y$
Sei $y = (y_1, \ldots, y_n)$ eine optimale Lösung, die mit x maximal lang übereinstimmt, etwa in den ersten $k-1$ Komponenten, also:

- $p(y) = OPT$,
- $y_i = x_i$ für $1 \leq i < k$, $1 \leq k \leq n$,
- $y_k \neq x_k$
- $p(z) = OPT$, $w(z) \leq K \Rightarrow z_i \neq x_i$ für ein $i \leq k$ $(++)$

Es gilt $y_k < x_k$, denn
$k < j$: Es ist $x_k = 1$, also $y_k < x_k$.
$k = j$: Nach Konstruktion von x gilt: $w(x) = K$
$\Rightarrow y_k < \alpha = x_k$, da sonst $w(y) > K$
$k > j$: Es wäre $y_k > 0 = x_k$.
Dies kann nicht eintreten, da sonst $w(y) > K$.

Das Bruchteil-Rucksackproblem (Forts.)

Beweis von Lemma 4.1 (Forts.)

Es gibt $z = (z_1, \ldots, z_n) \in [0,1]^n$ mit

- $w(z) = K$,
- $x_i = y_i = z_i$ für $1 \leq i < k$ und
- $y_k < x_k = z_k$.

d.h.:
- erhöhe y_k zu x_k und
- verkleinere dafür $y_{k+1}, \ldots, y_n$ so, dass $w(z) = K$.

Dann gilt:
- z stimmt mit x in k Komponenten überein und
- z ist optimal, denn:

$$
\begin{aligned}
p(z) &= p(y) + (z_k - y_k) \cdot w_k \cdot q_k - \\
&\quad \sum_{i=k+1}^{n} (y_i - z_i) \cdot w_i \cdot q_i \\
&\underset{(+)}{\geq} p(y) + \\
&\quad \left((z_k - y_k) \cdot w_k - \sum_{i=k+1}^{n} (y_i - z_i) \cdot w_i\right) \cdot q_k \\
&= p(y)
\end{aligned}
$$

$\Rightarrow \lightning$ zu $(++)$

Algorithmus

$\mathcal{A}$: BT-RSP$(n, p, w, K) =$

Sortiere die p_i/w_i so, dass $p_1/w_1 \geq \cdots \geq p_n/w_n$.
$g \leftarrow K$
for $i = 1$ **to** n **do**
  if $g \geq w_i$
    then
      $x_i \leftarrow 1;\ g \leftarrow g - w_i$
    else
      if $g > 0$
        then
          $x_i \leftarrow g/w_i$;
          $g \leftarrow 0$
        else
          $x_i \leftarrow 0$
      fi
  fi
od
output (x)

Korrektheit von BT-RSP

BT-RSP berechnet das x aus Lemma 4.1.

Zeitbedarf

- $O(n \cdot \log n)$ für das Sortieren
- $O(n)$ für den Rest

also:

$$T(n) \in O(n \cdot \log n)$$

4.3 Bäume mit minimaler gewichteter Weglänge

Definition 4.2

Sei T ein Binärbaum mit n Blättern der Tiefe $b_1, \ldots, b_n$ und Gewichten $q_1, \ldots, q_n$.

Dann ist

$$P_T = \sum_{i=1}^{n} b_i q_i$$

die **gewichtete Weglänge** von T, und

$$\sum_{i=1}^{n} q_i$$

ist das **Gewicht** von T.

Problem

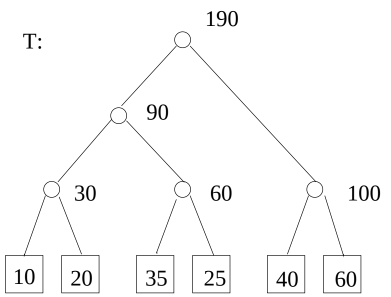
Eingabe: $n, q_1, \ldots, q_n, \quad n \in \mathbb{N}_+, \quad q_i \in \mathbb{R}_+$

Ausgabe: Binärbaum T mit minimalem P_T

Bäume mit minimaler gewichteter Weglänge (Forts.)

Beispiel

$n = 6, \quad Q = (10, 20, 25, 35, 40, 60)$



T mit induziertem Gewicht für jeden Teilbaum

$\Rightarrow P_T = 3 \cdot 10 + 3 \cdot 20 + 3 \cdot 35 + 3 \cdot 25 + 2 \cdot 40 + 2 \cdot 60 = 470$

Ist T optimal?

Beachte

Ähnlichkeit zu optimalen Suchbäumen, aber hier können Blätter beliebig angeordnet sein.

Anwendungen: Huffman-Code

Definition 4.3

Ein **Präfixcode** zu einer Menge M ist

$$c : M \to \{0,1\}^*$$

mit: $m, m' \in M \;\rightsquigarrow\; c(m)$ ist kein Präfix von $c(m')$.
(Dies garantiert leichte Dekodierbarkeit, d.h. den Übergang
$w \in \{0,1\}^* \;\rightsquigarrow\; m \in M$ mit $c(m) = w$.)

Beachte

T Binärbaum, $M =$ Menge der Blätter von T
$\Rightarrow T$ definiert Präfixcode für M.
Im Beispiel: $M = Q = \{10, 20, 25, 35, 40, 60\}$ und
T von letzter Folie.

$$c(10) = 000 \quad c(20) = 001 \quad c(35) = 010$$
$$c(25) = 011 \quad c(40) = 10 \quad c(60) = 11$$

Problem

Eingabe: $n, M = (m_1, \ldots, m_n), Q = (q_1, \ldots, q_n)$
n Objekte mit Namen m_i, Zugriffshäufigkeit q_i
Ausgabe: Präfixcode mit minimaler mittlerer Codelänge

Lösung

Bestimme Baum T zu n und Q mit minimaler gewichteter Weglänge und den m_i in den Blättern zu q_i.
Dies liefert optimalen Code c für M.

Anwendungen: Mischproblem

Problem

Eingabe: n Files der Länge $q_1, \ldots, q_n$ (einzeln sortiert).
Diese sollen auf einem Prozessor zu einem File der Länge $q_1 + \cdots + q_n$ gemischt werden, wobei jeweils 2 Files pro Lauf gemischt werden.
Ausgabe: Optimale Misch-Strategie, also Mischen mit minimaler Anzahl von Vergleichen

Beachte

Ein Binärbaum T mit n Blättern der Tiefe $b_1, \ldots, b_n$ und Gewichten $q_1, \ldots, q_n$ definiert eine Misch-Strategie mit P_T Vergleichen: Abarbeitung des Baum bottom-up.

Lösung

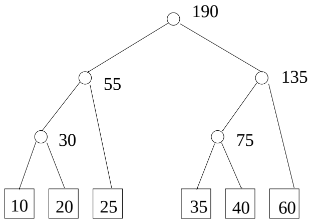
Bestimme T zu $Q = (q_1, \ldots, q_n)$ mit minimaler gewichteter Weglänge.
Dies definiert eine optimale Misch-Strategie.

Ansätze zur Lösung von B-MGW (1)

1. Ansatz: Teilen + Beherrschen

- Sortiere so, dass $q_1 \leq q_2 \leq \cdots \leq q_n$ gilt.
- Teile $n/2 : n/2$.
- Löse Teilprobleme.
- Baue zusammen.

Das Beispiel liefert:



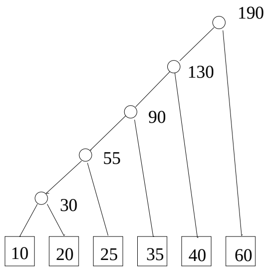
$\Rightarrow P_T = 485$
$\Rightarrow$ nicht optimal

Ansätze zur Lösung von B-MGW (2)

2. Ansatz: Greedy-Strategie

- Starte mit leerer Folge.
- Nimm jeweils kleinstes q_i hinzu.

Das Beispiel liefert:



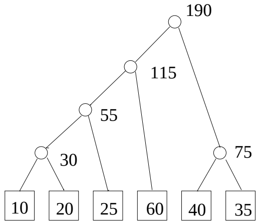
$\Rightarrow P_T = 495$
$\Rightarrow$ nicht optimal, Verfahren ist nicht gierig genug

Ansätze zur Lösung von B-MGW (3)

3. Ansatz:

- Starte mit n Bäumen $\boxed{q_i}$, $i = 1, \ldots, n$.
- Verschmelze jeweils die 2 Bäume mit kleinstem Gewicht.

Das Beispiel liefert:



$\Rightarrow P_T = 4 \cdot 10 + 4 \cdot 20 + 3 \cdot 25 + 2 \cdot 60 + 2 \cdot 40 + 2 \cdot 35 = 465$
$\Rightarrow$ Dies ist optimal.

Ansätze zur Lösung von B-MGW (4)

Wahl der Datenstruktur

- Für T:
  Baum (mit Gewicht in den Knoten)
  $\Rightarrow$ Verwalte Menge von Bäumen.
- Für Verschmelzung:
  Welche Operationen müssen unterstützt werden?
  a) 2 kleinste Gewichte in der Menge M finden und entfernen.
  b) Neues Gewicht (Summe der gewählten Gewichte) in M einfügen.

  $\Rightarrow$ Verwalte M als Min-Heap.
  - Kleinste $m \in M$ stehen ganz oben. $\rightsquigarrow$ Zeit: $O(1)$
  - Entfernen und Einfügen ist leicht. $\rightsquigarrow$ Zeit: $O(\log n)$

Algorithmus für B-MGW

Algorithmus

B-MGW(n, Q)=

Starte mit n Bäumen $\boxed{q_i}$, $i = 1, \ldots, n$.
Bilde Min-Heap M aus $q_1, q_2, \ldots, q_n$.
for $i = 1$ **to** $n - 1$ **do**
    $(p_1, p_2) \leftarrow$ die 2 kleinsten Elemente in M
    verschmelze die beiden Bäume zu einem Baum mit
        Gewicht $p_1 + p_2$
    entferne p_1, p_2 aus M und füge $p_1 + p_2$ ein
od

Lemma 4.2

Der von B-MGW(n, Q) berechnete Baum hat minimale gewichtete Weglänge bezüglich $q_1, \ldots, q_n$.

Zeitbedarf

- $O(n)$ für die Bäume $\boxed{q_i}$
- $O(n)$ für den Heap
- $\sum_{i=1}^{n-1} \log(n - i)$ für die Schleife

Also:

$$T(n) \in O(n \cdot \log n)$$

Beweis zum Lemma

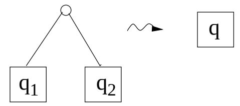
Sei $\mathcal{A}$ =B-MGW(n, Q).

Beweis: Induktion nach n

$n = 1$: Dies ist trivial

$n > 1$: Seien q_1, q_2 kleinste Elemente in Q und $q = q_1 + q_2$.
$\mathcal{A}$ verschmelze $\boxed{q_1}$, $\boxed{q_2}$ im 1. Schritt.
$\mathcal{A}$ liefere Baum T zur Eingabe $(q_1, q_2, \ldots, q_n)$.
$\mathcal{A}$ liefere Baum T' zur Eingabe $(q, q_3, \ldots, q_n)$.

Also: T' ist optimal für $(q, q_3, \ldots, q_n)$ nach IV.
T' entsteht aus T durch



(+)

$$P_T = P_{T'} + q$$

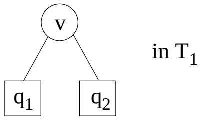
Die Eigenschaft (+) ist zentral für Korrektheit;
sie gilt für 1. und 2. Ansatz nicht !

Beweis zum Lemma (Forts.)

Sei T_0 ein optimaler Baum zu $(q_1, q_2, \ldots, q_n)$.

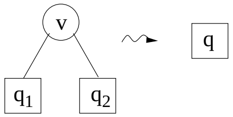
ⓥ innerer Knoten in T_0 maximaler Tiefe
mit Söhnen $\boxed{q_i}$, $\boxed{q_j}$

T_1 entsteht aus T_0 durch Vertauschen von
q_1, q_2 mit q_i, q_j, also



Also $P_{T_1} \leq P_{T_0}$, da q_1, q_2 minimal, v maximaler Tiefe.

Es entstehe T_1' aus T_1 durch



Also: T_1' ist Baum zu $(q, q_3, \ldots, q_n)$.
$P_{T'} \leq P_{T_1'}$, da T' optimal zu $(q, q_3, \ldots, q_n)$

Also: $P_T = P_{T'} + q \leq P_{T_1'} + q = P_{T_1} \leq P_{T_0}$
$P_T = P_{T_0}$, da T_0 optimal zu $(q_1, q_2, \ldots, q_n)$

Also: T ist optimal zu $(q_1, q_2, \ldots, q_n)$.

4.4 Matroide

Ziel

Abstraktion der Greedy-Strategie, so dass ein Korrektheitsbeweis für Instanzen einfach ist.

Zentrale Idee der Strategie

Eingabe: Menge $E = \{e_1, \ldots, e_n\}$

Ausgabe: Optimale zulässige Teilmenge $T \subseteq E$

Vorgehen: T sukzessiv aufbauen

```
Sortiere E : e_1 ≥ ··· ≥ e_n
T ← ∅
for k = 1 to n do
    if T ∪ {e_k} zulässig
       then T ← T ∪ {e_k}
    fi
od
```

Beispiel 0/1-Rucksackproblem

e_i Objekt mit Profit p_i und Gewicht w_i

T zulässig: $\sum_{e_i \in T} w_i \leq K$

T optimal: $\sum_{e_i \in T} p_i$ maximal

Ordnung: $e_i \geq e_j \ \text{gdw} \ p_i/w_i \geq p_j/w_j$

Hier liefert der Algorithmus **nicht** das optimale T.

Formale Abstraktion

Definition 4.4

a) Ein **Matroid** ist ein Paar $(E, \mathcal{U})$, wobei E eine endliche Menge und $\mathcal{U}$ ein System von Teilmengen von E ist mit:
   (1) $\varnothing \in \mathcal{U}$
   (2) $A \subseteq B, B \in \mathcal{U} \Rightarrow A \in \mathcal{U}$
   (3) $A, B \in \mathcal{U}, \|A\| < \|B\|$
   $\Rightarrow \exists x \in B \setminus A : A \cup \{x\} \in \mathcal{U}$

b) Sei $w : E \to \mathbb{R}$ eine Gewichtsfunktion. Dann ist für alle $A \in \mathcal{U}$:

$$w(A) = \sum_{e \in A} w(e)$$

c) Das **Optimierungsproblem zu** $(E, \mathcal{U};\ w)$ ist:

   Bestimme ein $\subseteq$-maximales $A \in \mathcal{U}$, so dass $w(A)$ maximal (bzw. minimal) ist.

Bemerkungen

- $A \in U$ ist **$\subseteq$-maximal**, wenn keine echte Obermenge von A in U liegt.
- Die $A \in \mathcal{U}$ sind die **zulässigen** Teilmengen von E.
  Wir suchen $\subseteq$-maximale $A \in \mathcal{U}$, die optimal bezüglich w sind.
- Wegen (2) heißt $\mathcal{U}$ **Teilmengen–abgeschlossen**.
- (3) heißt die **Austauscheigenschaft**.
  Sie wird benötigt, damit der Algorithmus ein optimales A liefert.

Algorithmus für $(E, \mathcal{U}; w)$

Algorithmus

$\mathcal{A}$: opt$(E, \mathcal{U}, w, T)$ =

```
Ref: T
sortiere die e_i, so dass
        w(e_1) ≥ w(e_2) ≥ ··· ≥ w(e_n)
for k = 1 to n do
    if T ∪ {e_k} ∈ U
      then
          T ← T ∪ {e_k}
    fi
od
```

Satz 4.1.

Sei $(E, \mathcal{U}; w)$ ein Matroid mit Gewichtsfunktion $w : E \to \mathbb{R}$.

Der Algorithmus opt$(E, \mathcal{U}, w)$ liefert ein $\subseteq$-maximales $T \in \mathcal{U}$, so dass $w(T) \geq w(A)$ für alle $\subseteq$-maximalen $A \in \mathcal{U}$ gilt.

Sortiert man die e_i so, dass $w(e_i) \leq w(e_{i+1})$ gilt, so liefert opt$(E, \mathcal{U}, w)$ ein $\subseteq$-maximales $T \in \mathcal{U}$, so dass $w(T) \leq w(A)$ für alle $\subseteq$-maximalen $A \in \mathcal{U}$ gilt.

Beweis zu Satz 4.1

Offenbar ist T $\subseteq$-maximal.

Sei $E = \{e_1, \dots, e_n\}$ und $w(e_i) \geq w(e_{i+1})$.

Sei $T = \{e_{i_1}, \dots, e_{i_k}\}$ die von $\mathcal{A}$ gelieferte Menge und
$T_0 = \{e_{j_1}, \dots, e_{j_l}\}$ eine optimale Lösung mit
$w(e_{i_1}) \geq \dots \geq w(e_{i_k})$ und
$w(e_{j_1}) \geq \dots \geq w(e_{j_l})$. $\quad (+)$

Zu zeigen: $w(T) \geq w(T_0)$

a) Sei $w(e_{i_\mu}) \geq w(e_{j_\mu})$ für $\mu = 1, \dots, \min\{k, l\}$.
Ist $k = l$, so gilt $w(T) \geq w(T_0)$.
Ist $k \neq l$, so ist entweder T oder T_0 nicht $\subseteq$-maximal.
$\leadsto$ ↯

b) Sei $w(e_{i_\mu}) \geq w(e_{j_\mu})$ für $\mu = 1, \dots, p-1$ und
$w(e_{i_p}) < w(e_{j_p})$.
Setze $A = \{e_{i_1}, \dots, e_{i_{p-1}}\}$ und $B = \{e_{j_1}, \dots, e_{j_p}\}$.
$\underset{(2)}{\leadsto}$ $A, B \in \mathcal{U}$, wegen
- $A \subseteq T$, $B \subseteq T_0$ und
- $T, T_0 \in \mathcal{U}$

$\underset{(3)}{\leadsto}$ $\exists e \in B \setminus A : A \cup \{e\} \in \mathcal{U}$ wegen $\|A\| < \|B\|$

$\leadsto w(e) \overset{(+)}{\geq} w(e_{j_p}) > w(e_{i_p})$

$\leadsto$ ↯, da $\mathcal{A}$ e hätte aufnehmen müssen.

a) + b) $\leadsto$ Beh.

4.5 Auftragsplanung mit Schlußterminen

Problem

Eingabe: n Aufträge mit Profiten p_i und Terminen d_i.
Jeder Auftrag benötigt 1 Zeiteinheit.

Ausgabe: Optimaler Auftragsplan $\vec{A} = (i_1, \dots, i_k)$, d. h.,
- $d_{i_j} \geq j$ (alle Termine eingehalten)
- $\sum_{j=1}^{k} p_{i_j} = \max$ (maximaler Profit)

Definition 4.5

$A \subseteq \{1, \dots, n\}$ heißt **zulässig**, falls sich die $i \in A$ so zu $\vec{A} = (i_1, \dots, i_k)$ anordnen lassen, dass $d_{i_j} \geq j$ für alle $j = 1, \dots, k$ gilt.

Beachte

Sei $A \subseteq \{1, \dots, n\}$.
Sortiert man die $i \in A$ zu $\vec{A} = (i_1, \dots, i_k)$, so dass $d_{i_j} \leq d_{i_{j+1}}$ gilt, so ist A genau dann zulässig, wenn gilt

$$d_{i_j} \geq j \quad \text{für alle} \quad j = 1, \dots, k$$

Planungsstrategie: "Die dringendsten Aufträge zuerst."

Idee

Entwickle Greedy-Algorithmus:
Nimm die Aufträge mit größtem Profit zuerst.

Algorithmus APlanST

Algorithmus

$\mathcal{A}$: APlanST$(n, P, D) =$

```
Sortiere so, dass  p_1 ≥ p_2 ≥ ··· ≥ p_n gilt.
A ← ∅
for i = 1 to n do
    if A ∪ {i} zulässig
      then A ← A ∪ {i}
    fi
od
Gib A aus.
```

Beispiel

Seien $n = 4, P = (100, 27, 15, 10), D = (2, 1, 2, 1)$.

Es gibt folgende zulässige $\subseteq$-maximale A mit Profit $P(A)$:

A :	$\{2,1\}$	$\{2,3\}$	$\{4,1\}$	$\{4,3\}$	$\{1,3\}$
$P(A)$:	127	42	110	25	115

Optimal ist der Plan $\vec{A} = (2, 1)$ mit Profit 127.

Der Algorithmus liefert im i-ten Durchlauf A_i:

$A_1 = \{1\}$
$A_2 = \{2, 1\} = A_3 = A_4$,
also: $\vec{A} = (2, 1)$

Korrektheit von APlanST

Lemma 4.3

APlanST(n, P, D) liefert einen optimalen Auftragsplan.

Beweis: Wir zeigen: Dies ist Optimierungsproblem für Matroide.
Sei $E = \{1, \ldots, n\}$
$A \in \mathcal{U}$ gdw A ist zulässig
$w(i) = p_i$

Wir zeigen: $(E, \mathcal{U})$ erfüllt Matroid-Bedingungen (1)-(3).
Dann gilt das Lemma nach Satz 4.1.

Zu (1),(2): Dies gilt.
Zu (3): Seien A, B zulässig mit $\|A\| < \|B\|$.
Zu zeigen: $\exists i \in B \setminus A : A \cup \{i\}$ ist zulässig.
Sei $j \in B \setminus A$, d_j maximal und
$\vec{B} = (i_1, \ldots, j, \ldots, i_k)$.
In A gibt es
- nicht weniger i mit $d_i > d_j$
- weniger i mit $d_i \leq d_j$

als in B.
Also ist $A \cup \{j\}$ zulässig, da A und B zulässig sind
(j kann in $\vec{A}$ eingeschoben werden).

Zeitbedarf von APlanST

Fragen

- Speicherung von A?
- Realisierung von „$A \cup \{i\}$ zulässig?“

Lösung:

A als Folge $\vec{A}$ speichern
$\rightsquigarrow$ Zeitbedarf, falls $\vec{A}$ die Länge l hat: $O(l)$

Zeitbedarf

$O(n \cdot \log n)$	für (1)
$O(i)$	für (4) und (5)
$O(n^2)$	für Schleife (3-7)

$\Rightarrow$ Zeitbedarf von APlanST(n, P, D):

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

4.6 Aufspannende Bäume in Graphen

Definition 4.6
Sei $G = (V, E)$ ein ungerichteter Graph.
$T = (V, E')$ heißt **aufspannender Baum** für G, falls

- T ein Baum ist,
- $E' \subseteq E$ gilt, und
- jeder Knoten $v \in V$ auf einer Kante $e \in E'$ liegt.

Ist $G = (V, E;\ w)$, $w : E \to \mathbb{N}$, ein gewichteter Graph, so ist das **Gewicht** von T:

$$w(T) = \sum_{e \in E'} w(e)$$

Problem

Eingabe:	$G = (V, E;\ w)$ gewichteter zusammenhängender Graph
Ausgabe:	Ein aufspannender Baum T mit minimalem Gewicht

Idee Kruskal-Algorithmus

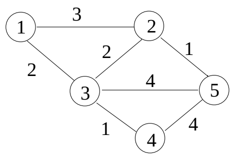
- Verwalte Wald von Bäumen.
- Nimm jeweils eine leichteste Kante hinzu, so dass wieder ein Wald (also kein Kreis) entsteht.

Beachte

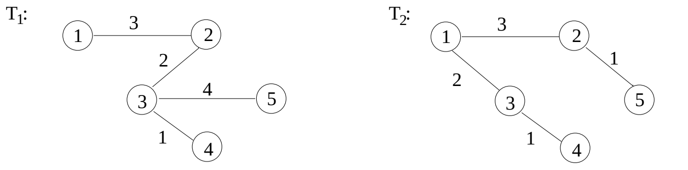
Hat G n Knoten, so hat jeder Wald $\leq n - 1$ Kanten.
Also hat jeder aufspannende Baum genau $n - 1$ Kanten.

Beispiel

Sei G

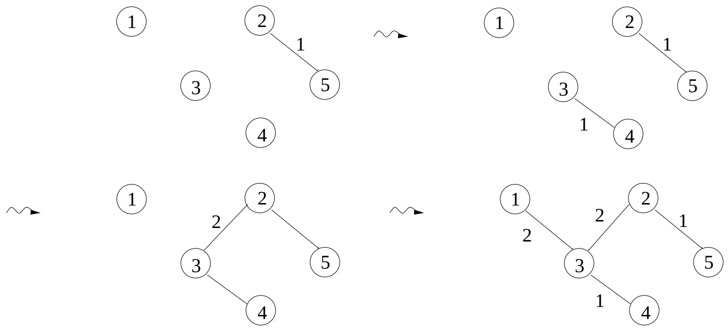


Aufspannende Bäume sind zum Beispiel:



$w(T_1)=10$ $w(T_2)=7$

Der Algorithmus liefert:



$w(T)=6$

Formulierung als Matroid-Problem

Sei
- E= Menge der Kanten des Universums
- $A \in \mathcal{U}$ gdw (V, A) ist kreis-frei
- $w : E \to \mathbb{N}$ die gegebene Gewichtsfunktion

Lemma 4.4

$(E, \mathcal{U})$ ist ein Matroid

Beweis: Übung

Algorithmus Kruskal

$\mathcal{A}$: Kruskal(n, V, E, w) =

Sortiere E so, dass $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_k)$ gilt.
$A \leftarrow \varnothing;\ m \leftarrow 0$
for $i = 1$ **to** k **do**
  if $A \cup \{e_i\} \in \mathcal{U}$
   then $A \leftarrow A \cup \{e_i\}$
      $m \leftarrow m + 1$
      if $m = n - 1$
       then **goto** Stopp **fi** **fi**
od
Stopp: Gib A aus.

Zeitbedarf

Hängt vom Test „$A \cup \{e_i\}$ zulässig?“ ab.
Siehe später.

4.7 Kürzeste Wege in Graphen

Problem

Eingabe: $G = (V, E; w), w : E \to \mathbb{N}$ gewichteter Graph, $u \in V$

Ausgabe: für alle $v \in V$:
- $\alpha(v) =$ kürzester Weg $u \rightsquigarrow v$
- $l(v) =$ Länge von $\alpha(v)$

Beachte Optimalitätsprinzip

$$\alpha(v) = \beta(v', v) \quad \rightsquigarrow \quad \beta = \text{kürzester Weg } u \rightsquigarrow v'$$

Idee: Greedy-Algorithmus

Verlängere jeweils das kürzeste $\alpha(v)$ minimal um (v, v') zu $\alpha(v')$.

Sei $k(v) =$ letzte Kante (v', v) auf $\alpha(v)$ für alle $v \in V$.
Dann lässt sich zu jedem v der Weg $\alpha(v)$ rückwärts von v nach u berechnen.

Name: Dijkstra-Algorithmus

Algorithmus Dijkstra

Schleifeninvarianten

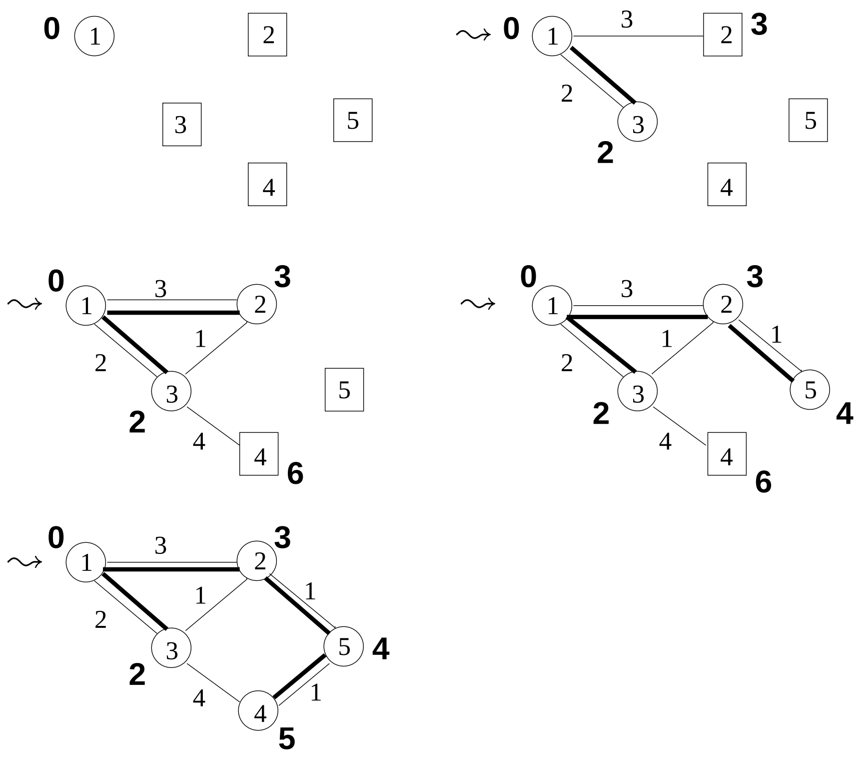
- $W = \{v \in V \mid l(v)$ noch nicht bekannt$\}$
- $l(v) =$ Länge des bisher besten Weges $u \rightsquigarrow v$
- $k(v) = (v', v) =$ letzte Kante des bisher besten Weges $u \rightsquigarrow v$

Algorithmus

Dijkstra(u, V, E, w)=

 for $v \in V$ **do**
  if $(u, v) \in E$
   then $l(v) \leftarrow w(u, v), k(v) \leftarrow (u, v)$
   else $l(v) \leftarrow \infty$ **fi** **od**
 $W \leftarrow V \setminus \{u\}$
 $l(u) \leftarrow 0$
 for $i = 1$ **to** $n - 1$ **do**
  Finde $v \in W$ mit $l(v)$ minimal.
  $W \leftarrow W \setminus \{v\}$
  for $v' \in W$ mit $(v, v') \in E$ **do**
   if $l(v) + w(v, v') < l(v')$
    then $l(v') \leftarrow l(v) + w(v, v')$
     $k(v') \leftarrow (v, v')$ **fi** **od**
 od
 Gib Vektoren l, k aus.

Beispiel

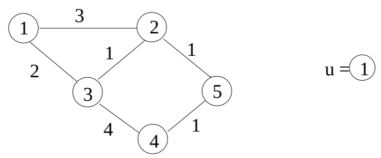


Die Elemente von $i \in W$ sind als $\boxed{i}$ dargestellt.

Angegeben sind der bisher besuchte Teilgraph und die bisher besten $l(v)$ und die $\alpha(v)$.

Beispiel (Forts.)

Sei G:



Ablauf des Algorithmus (for-Schleife (7)-(14)):

i	0	1	2	3	4
aus W	①	③	②	⑤	④
$l(1)$	0	0	0	0	0
$k(1)$	–	–	–	–	–
$l(2)$	3	$\underline{3}$	3	3	3
$k(2)$	$(1,2)$	$(1,2)$	$(1,2)$	$(1,2)$	$(1,2)$
$l(3)$	$\underline{2}$	2	2	2	2
$k(3)$	$(1,3)$	$(1,3)$	$(1,3)$	$(1,3)$	$(1,3)$
$l(4)$	∞	6	6	$\underline{5}$	5
$k(4)$	–	$(3,4)$	$(3,4)$	$(5,4)$	$(5,4)$
$l(5)$	∞	∞	$\underline{4}$	4	4
$k(5)$	–	–	$(2,5)$	$(2,5)$	$(2,5)$

Das minimale $l(v)$ mit $v \in W$ ist in jeder Spalte unterstrichen.

Ergebnis:

Optimaler Weg ① $\rightsquigarrow$ ④
Länge $l(4) = 5$
Konstruktion des Weges:

① $\rightsquigarrow$ ④ ① $\rightsquigarrow$ ⑤-④ ① $\rightsquigarrow$ ②-⑤-④ ①-②-⑤-④

Korrektheit des Dijkstra-Algorithmus

1. Beweis:

Direkt mit dem Optimalitätsprinzip wie in Abschnitt 4.3.

2. Beweis: Reduktion auf Matroid-Problem

Sei $(E, U; w)$ erklärt durch:

- $E = \{\alpha \mid \alpha = u \rightsquigarrow v$ kreis-frei, $v \in V\}$
- $A \in \mathcal{U}$ gdw $u \rightsquigarrow v_1$, $u \rightsquigarrow v_2$ in $A \Rightarrow v_1 \neq v_2$
- $w(\alpha) = \sum_{e \in \alpha} w(e)$

Es gilt:

a) $(E, \mathcal{U})$ ist ein Matroid.

b) Dijkstra ist effiziente Implementierung des Optimierungs-Algorithmus für $(E, \mathcal{U}; w)$:
Man muß nicht erst alle $w(\alpha)$, $\alpha \in E$, berechnen und sortieren.
Dies geschieht implizit „on demand".

Zeitbedarf des Dijkstra-Algorithmus

Sei $n = \|V\|$.
Sei (E, w) als $n \times n$-Matrix, k und l als Vektoren gespeichert.

Zeitbedarf

a) Zeilen (1-6):

$$T_{(1-6)} \in O(n)$$

b) Zeilen (7-14):
Ein Schleifendurchlauf kostet zweimal $O(n)$ Zeiteinheiten.

$$T_{(7-14)} \in O(n^2)$$

c) insgesamt:

$$T_{\text{Dijkstra}} \in O(n^2)$$

Also: Zeitbedarf für Dijkstra(u, V, E, w), $\|V\| = n$:

$$T_{\text{Dijkstra}} \in O(n^2)$$

4.8 Der Union-Find-Algorithmus

Motivation

Problem

Sei M eine endliche Menge.

Man verwalte ein System disjunkter Teilmengen $M_k \subseteq M$ und unterstütze dabei die Operationen:

- Find(i): Bestimme k mit $i \in M_k$ (die Menge von i).
- Union(i, j): Vereinige die Mengen von i und j.

Beachte

Das Problem tritt z. B. beim Kruskal-Algorithmus auf.

Idee

Verwalte die Menge M als Wald. Jedes M_k ist als Baum T_k dargestellt mit Zeigern vom Sohn zum Vater. Die Wurzel des Baums T_k ist der Name von M_k.

Find(i): Steige von i hinauf zur Wurzel und hänge dabei jeden besuchten Knoten direkt unter die Wurzel. Find(i) ist die Wurzel und damit der Name der Menge von i.

Union(i, j): Berechne ii = Find(i), jj = Find(j) und hänge entweder ii direkt unter jj oder jj direkt unter ii.

Der Union-Find-Algorithmus: Realisierung (1)

Sei $M = \{1, \ldots, n\}$.

Realisierung der (gerichteten) Kanten als Feld $A[1, \ldots, n]$.
Dabei steht $A[i] = j$ für die Kante $i \to j$.

Die Algorithmen

```
Init(A) = Ref: A
          for i = 1 to n do
              A[i] ← i
          od

Find(A, i) = Ref: A
             if A[i] = i
               then
                   Find ← i
               else
                   j ← Find(A, A[i])
                   A[i] ← j
                   Find ← j
             fi
```

Der Union-Find-Algorithmus: Realisierung (2)

Union(A, i, j) = Ref: A
  $ii \leftarrow$ Find(A, i)
  $jj \leftarrow$ Find(A, j)
  random $z \in \{0, 1\}$
  if $z = 0$ **then** $A[ii] \leftarrow jj$
    else $A[jj] \leftarrow ii$
  fi

Bemerkung

Die Frage, ob der Baum T_{ii} unter den Baum T_{jj} gehängt wird, hängt hier von Zufall ab. Man will das Entstehen tiefer Bäume verhindern. Dazu könnte man auch den weniger tiefen in den tieferen Baum hängen.

Sei $$\log^*(n) = \mu_k[\underbrace{\log \circ \ldots \circ \log}_{k \text{ mal}}(n) \leq 1].$$

Die Funktion $\log^*$ wächst extrem langsam:

Für $n \leq 10^{19728}$ ist $\log^*(n) \leq 5$.

Satz 4.2.

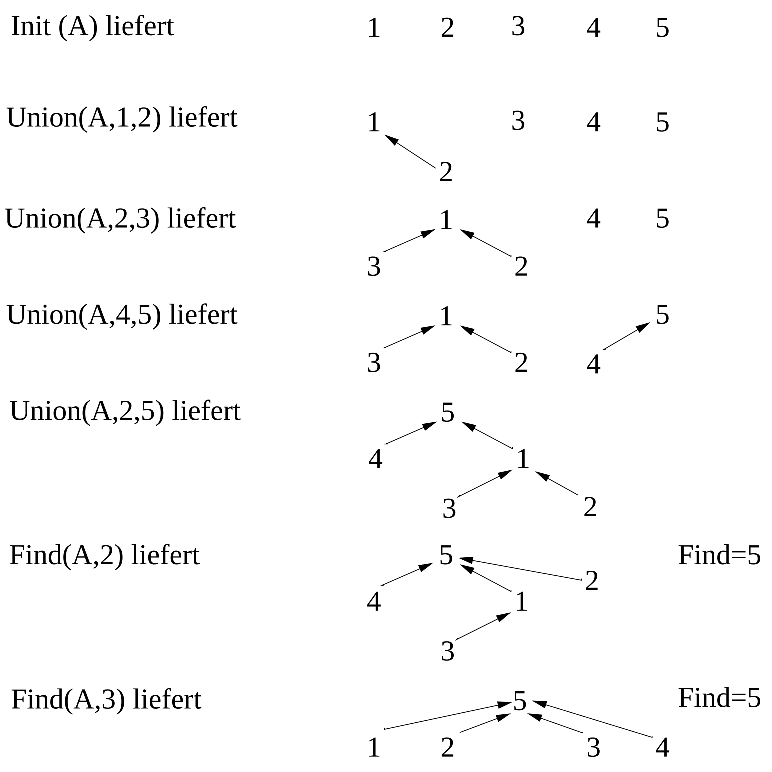
Der Union-Find-Algorithmus benötigt für die Bearbeitung von $O(n)$ Union- und Find-Operationen im Mittel die Zeit

$$T_{mittel} \in O(n \cdot \log^*(n)).$$

Ohne Beweis.

Der Union-Find-Algorithmus: Ein Beispiel

Sei $M = \{1, \ldots, 5\}$. Wir stellen M als Wald dar, lassen aber die Pfeile $i \to i$, d. h. $A[i] = i$, weg.



Der Union-Find-Algorithmus: Bemerkungen

Es gilt zu jedem Zeitpunkt

- i, j liegen in der gleichen Menge gdw Find$(i) =$ Find(j).
- Sei t die maximale Tiefe der Bäume.
  Find(A, i) und Union(A, i, j) haben Zeitbedarf in $O(t)$.
- Die maximale Tiefe t bleibt **sehr** klein (in der Praxis durch kleines $c \in \mathbb{N}$ beschränkt).

Beachte den engen Zusammenhang zwischen Äquivalenz-Relation und System disjunkter Mengen.

- Ist M_k, $k = 1, \ldots, p$, ein System disjunkter Teilmengen von M, so ist $\sim$ mit
  $i \sim j$ gdw i, j liegen in der gleichen Menge M_k.
  eine Äquivalenzrelation auf M.
- Ist $\sim$ eine Äquivalenzrelation auf M, so sind die Mengen M_k mit
  $$M^k = \{i \mid i \sim k\}$$
  disjunkt.
  D. h. es gilt entweder $M^k = M^{k'}$ oder $M^k \cap M^{k'} = \varnothing$.

Also

Der Union-Find-Algorithmus dient zur Verwaltung sich dynamisch ändernder Äquivalenz-Relationen.

Äquivalenz von endlichen Automaten

Problemstellung

Gegeben: Zwei endliche Automaten über dem Alphabet Σ.
Frage: Akzeptieren sie die gleiche Sprache?

Definition 4.7

Ein **endlicher Automat** ist ein 5-Tupel $M = (\Sigma, Q, q_0, \delta, F)$ mit

- $\Sigma = \{a_1, \ldots, a_r\}$ ein Alphabet
- $Q = \{q_0, \ldots, q_n\}$ die Zustandsmenge
- $q_0 \in Q$ der Startzustand
- $\delta : Q \times \Sigma \to Q$ die Übergangsfunktion
- $F \subseteq Q$ die Menge der akzeptierenden Zustände

Sei δ erweitert zu $\delta : Q \times \Sigma^* \to Q$ durch:
$$\delta(q, e) = q \qquad \delta(q, ax) = \delta(\delta(q, a), x)$$
Dann ist:
$$L(M) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$$
die von M akzeptierte Sprache.

Also lautet die **Problemstellung** präzise:

Gegeben: $M_1 = (\Sigma, Q_1, q_0^1, \delta_1, F_1)$, $M_2 = (\Sigma, Q_2, q_0^2, \delta_2, F_2)$
Frage: Gilt $L(M_1) = L(M_2)$?
Gilt $L(M_1) = L(M_2)$, so heißen M_1 und M_2 auch **äquivalent**.

Idee des Entscheidungsalgorithmus

Seien $M_i = (\Sigma, Q_i, q_0^i, \delta_i, F_i)$, $i = 1, 2$, endliche Automaten mit $Q_1 \cap Q_2 = \emptyset$.

Setze:
- $Q = Q_1 \cup Q_2$
- $F = F_1 \cup F_2$
- $\bar{F} = Q \setminus F$
- $\delta : Q \times \Sigma \to Q$
- $\delta(q, a) = \delta_i(q, a)$, falls $q \in Q_i$
- $\approx$ Äquivalenzrelation auf Q mit
  $p \approx q$ gdw $\forall x \in \Sigma^* : \delta(p, x) \in F \Leftrightarrow \delta(q, x) \in F$

Also: $L(M_1) = L(M_2)$ gdw $q_0^1 \approx q_0^2$

Berechne $\sim$ auf Q so, dass gilt:

(1) $\forall x \in \Sigma^* : \delta_1(q_0^1, x) \sim \delta_2(q_0^2, x)$

(2) $q_0^1 \approx q_0^2 \;\Rightarrow\; \sim \subseteq \approx$

Dann gilt:

$L(M_1) = L(M_2)$

gdw $q_0^1 \approx q_0^2$

gdw $\forall q \in Q : [q] \subseteq F \vee [q] \subseteq \bar{F}$

Algorithmus ÄquivEA

Seien
- Q, F, δ, Σ wie oben,
- $m = \|Q\|$, $\varphi : Q \to \{1, \ldots, m\}$ bijektiv,
- $A[1, \ldots, m]$ ein Feld,
- S eine Schlange über $Q \times Q$.

Algorithmus

$\mathcal{A}$: ÄquivEA(M_1, M_2) =

```
Init(A)
S ← (q_0^1, q_0^2);  erg ← true
while S nicht leer  and  erg = true  do
    (p, q) ← erstes Element von S
    S ← S ohne erstes Element
    i ← Find(A, φ(p));  j ← Find(A, φ(q))
    if i ≠ j then
        if p ∈ F ⇔ q ∈ F
        then Union(A, i, j)
             for all a ∈ Σ do
                 S ← S • (δ(p, a), δ(q, a))
             od
        else erg ← false
        fi
    fi
od
output(erg)
```

Zeitbedarf von ÄquivEA

Beachte

- Beim Start gibt es m Äquivalenzklassen.
  $\Rightarrow$ Die Union-Operation kann also höchstens m-mal ausgeführt werden.
- In S werden $O(m)$ Paare (p, q) aufgenommen.
- Es gibt $O(m)$ viele Durchläufe durch die while-Schleife.
  Zeitbedarf pro Durchlauf ist Zeitbedarf für die Union-/Find-Operationen $+ O(1)$.

Also

$$T_{\mathcal{A},mittel}(m) \in O(m \cdot \log^*(m))$$

$\mathcal{A}$ hat "im Wesentlichen" linearen Zeitbedarf.

Speziell

$\mathcal{A}$ hält mit jeder Eingabe.

Korrektheit von ÄquivEA

Sei $\sim_i$ $= \sim$ beim Start des i-ten Schleifendurchlaufs
$\sim$ = die Äquivalenzrelation beim Stopp
$[q]_i$ = die Äquivalenzklasse von q zu $\sim_i$
$[q]$ = die Äquivalenzklasse von q beim Stopp

Lemma 4.5

a) Für alle $i \geq 1$ gilt:
   i) $q_0^1 \approx q_0^2 \quad \Rightarrow \quad \sim_i \subseteq \approx$
   ii) $erg =$ **true** $\quad \Rightarrow \quad \forall q \in Q : [q]_i \subseteq F$ oder $[q]_i \subseteq \bar{F}$

b) Zu jedem (p, q) in S gibt es $x \in \Sigma^*$ mit $p = \delta_1(q_0^1, x)$, $q = \delta_2(q_0^2, x)$.

c) Sei $erg =$ **true** beim Stopp.
   Dann gilt: $\delta_1(q_0^1, x) \sim \delta_2(q_0^2, x)$ für alle $x \in \Sigma^*$.

Beweis: Übung

Korrektheit von ÄquivEA (Forts.)

Satz 4.3.

$\mathcal{A}$ liefert beim Stopp $erg =$ **true** genau dann, wenn $L(M_1) = L(M_2)$ gilt.

Beweis:

"$\Leftarrow$" Sei $L(M_1) = L(M_2)$.
also: $\forall x \in \Sigma^* : \delta_1(q_0^1, x) \in F \Leftrightarrow \delta_2(q_0^2, x) \in F$
also kann erg nie auf **false** gesetzt werden nach b).
also gilt $erg =$ **true** beim Stopp.

"$\Rightarrow$" Sei $erg =$ **true** beim Stopp.
also $\forall q : [q]_i \subseteq F$ oder $[q]_i \subseteq \bar{F}$ für alle i nach a)ii)
also $\forall q : [q] \subseteq F$ oder $[q] \subseteq \bar{F}$
also $\forall x \in \Sigma^* : \delta_1(q_0^1, x) \in F \Leftrightarrow \delta_2(q_0^2, x) \in F$
nach c)
also $L(M_1) = L(M_2)$

5 String Matching

Problemstellung

Sei Σ ein festes Alphabet.
$T = T_1 \dots T_n$ $\quad$ $T_i \in \Sigma$ ist ein **Text**.
$P = P_1 \dots P_m$ $\quad$ $P_j \in \Sigma$ ist ein **Muster** (pattern).
$T^i = T^{i,m} = T_i \dots T_{i+m-1}$
$m < n$

P kommt in T an der Stelle i vor, falls $P = T^i$.
Dann heißt i auch eine „matching position".

Beispiel

$T = a\ b\ a\ b\ a \qquad P = a\ b\ a$
P kommt zweimal in T vor, bei $i = 1$ und $i = 3$

Problem

Eingabe: $T = T_1 \dots T_n$
$P = P_1 \dots P_m$
Ausgabe: alle i mit $P = T^i$, d. h. alle Vorkommen von P in T

Bemerkungen

- Dieses Problem tritt bei der Volltextsuche nach Mustern in Dokumenten auf.
- n und m sind im Allgemeinen sehr groß.

Ein trivialer Algorithmus

Idee: Für $i = 1, \ldots, n-m$ teste, ob i matching position ist.

$\mathcal{A}$: TrivStringMatch(T, n, P, m)=

```
for i = 1 to n − m do
    for j = 0 to m − 1 do
        if T_{i+j} ≠ P_{j+1}
          then goto NEXT
        fi
    od
    output i
    NEXT :
od
```

Zeitbedarf

$T_{\mathcal{A}}(n, m) \in O(n \cdot m)$

Frage

Gibt es schnellere Algorithmen?

Etwa Zeit $O(n+m)$?

5.1 Fingerabdrücke: Das Rabin-Karp-Verfahren

Idee „fingerprinting“

Berechne „Fingerabdrücke“ $h(P)$ und $h(T^i)$ mit

1. $h(P) = h(T^i) \Rightarrow P = T^i$ mit hoher Wahrscheinlichkeit,
2. $h(P)$ und $h(T^1)$ sind in Zeit $O(m)$ berechenbar,
3. $h(T^{i+1})$ ist in Zeit $O(1)$ aus $h(T^i)$ berechenbar.

Dann für $i = 1, \ldots, n-m$:

Falls $h(P) = h(T^i)$, teste, ob $P = T^i$ und gib gegebenenfalls i aus.

Das Rabin-Karp-Verfahren (Forts.)

Wahl von $\mathbf{h}$

Sei $q \in \mathbb{N}$, $\Sigma = \{a_0, \ldots, a_{r-1}\}$ also $\|\Sigma\| = r$

$$\varphi(a_{i_{m-1}} \ldots a_{i_1} a_{i_0}) = i_{m-1} r^{m-1} + \cdots + i_1 r + i_0$$

Setze: $\quad h(w) = \varphi(w) \bmod q \qquad w \in \Sigma^*,\ |w| = m$

Dabei ist q eine große Primzahl, so dass $q \cdot r$ in eine Speicherzelle passt.

Es gilt 1.-3.
zu 1. und 2.: Dies gilt natürlich.
zu 3.: Sei $\quad T^i = a_k v$ und $T^{i+1} = v a_l$.
Dann ist („bei T^i vorne a_k abschneiden und hinten a_l anhängen“).

$$h(T^{i+1}) = ((h(T^i) - k \cdot r^{m-1}) \cdot r + l) \bmod q$$

Dies ist in Zeit $O(1)$ berechenbar, also unabhängig von $n = |T|$ und $m = |P|$.

Der Rabin-Karp-Algorithmus

Sei der Einfachheit halber $\Sigma = \{0, 1, \ldots, r-1\}$, also $a_i = i \in \mathbb{N}$.

Algorithmus

$\mathcal{A}$: RK-StringMatch(T, n, P, m) =

```
p ← 0;
t ← 0;
u ← r^(m-1) mod q
for i = 1 to m do
    p ← (r · p + P_i) mod q
    t ← (r · t + T_i) mod q
od
for i = 1 to n - m + 1 do
    if p = t
      then if P = T^i
               then  output i
           fi
    fi
    if i ≤ n - m
      then
           t ← ((t - T_i · u) · r + T_(i+m)) mod q
    fi
od
```

Zeitbedarf von RK-StringMatch

Zeitbedarf im schlechtesten Fall

$$T_{\mathcal{A}}^{\max}(n, m) \in O(n \cdot m)$$

Für $T = a^n$, $P = a^m$ werden $\Theta(n \cdot m)$ Schritte benötigt.

Zeitbedarf im Mittel

Sei $k_{n,m}$ die mittlere Anzahl der Stellen i mit $h(P) = h(T^i)$.
Dann ist der Zeitbedarf im Mittel

$$O(m) + O(k_{n,m} \cdot m) + O(n)$$

Realistische Annahme

Es gibt im Mittel nicht mehr als n/m Stellen i mit $h(P) = h(T^i)$.

Zeitbedarf im Mittel

unter dieser Annahme

$$T_{\mathcal{A}}^{\text{mittel}}(n, m) \in O(n + m)$$

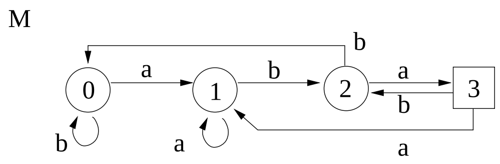
5.2 String Matching mit endlichen Automaten

Idee

Konstruktion eines endlichen Automaten M aus P mit:
Ist $T = uPv$, so ist M nach Einlesen von uP in einem Endzustand.

Beispiel

$\Sigma = \{a, b\}$, $P = a\ b\ a$
Dies liefert



Eingabe $T = a\ \ b\ \ a\ \ b\ \ a\ \ a\ \ b\ \ a$
Zustand $\quad 0\ \ 1\ \ 2\ \ \underline{3}\ \ 2\ \ \underline{3}\ \ 1\ \ 2\ \ \underline{3}$

also P kommt in T an den Stellen vor:

$$\begin{aligned} 3 - |P| + 1 &= 1, \\ 5 - |P| + 1 &= 3, \\ 8 - |P| + 1 &= 6 \end{aligned}$$

Definition von M aus P

Sei $|P| = m$.

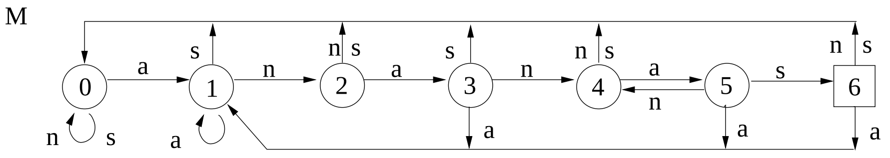
Setze $M = (Q, \Sigma, q_0, \delta, F)$ mit

- $Q = \{0, 1, \ldots, m\}$ — Zustände
- $q_0 = 0 \in Q$ — Startzustand
- $F = \{m\}$ — Endzustände
- $\delta : Q \times \Sigma \to Q$ — Übergangsfunktion

$$\delta(i, a) = \begin{cases} \max\{k \leq i+1 \mid P_1 \ldots P_k \text{ ist Endwort von } P_1 \ldots P_i a\} \\ \quad \text{falls solch ein Endwort existiert} \\ 0 \quad \text{sonst} \end{cases}$$

Beispiel

$\Sigma = \{a, n, s\}$, $P = a\ n\ a\ n\ a\ s$
Dies liefert



Analyse

Zeitbedarf

$T(n, m)$= Zeit für Konstruktion von $M + O(n)$.

Zeitbedarf für Berechnung von M (also δ) erscheint groß. Der nächste Abschnitt zeigt, dass M in Zeit $T(n) \in \Theta(n)$ berechnet werden kann (bei festem Σ).

Korrektheit

Siehe nächsten Abschnitt.

5.3 Das Knuth-Morris-Pratt-Verfahren

Beachte

Der naive Algorithmus testet „$P = T^i$?“ für $i = 1, \ldots, n$.

$\Rightarrow$ Also: Er schiebt das „Fenster P“ über T, und zwar immer um eine Stelle nach rechts.

Idee
des KMP-Verfahrens

1. Verschiebe das Fenster um möglichst viele Stellen.
2. Realisiere die Funktion δ des endlichen Automaten aus 5.2 (2-dimensionale Tabelle) als 1-dimensionale Verschiebetabelle Π.

Das Knuth-Morris-Pratt-Verfahren (Forts.)

Idee
der Verschiebung am Beispiel

P vor Verschiebung	$\underline{a\ a}\ a\ b\ \underline{a\ a}\ c \ldots$	
Text	$a\ a\ a\ b\ \underline{a\ a}\ d \ldots$	$c \neq d$
P nach Verschiebung	$\quad\quad\quad\ \underline{a\ a}\ a\ b\ a\ a\ c \ldots$	

Allgemeiner:

Sei

- $P = ucv$ $\qquad u, v \in \Sigma^*$, $c \in \Sigma$
- bei c trete ein „Mismatch“ auf
- x das max. Endwort von u, das gleichzeitig Anfangswort ist, d. h., $u = yx = xz$ für gewisse $y, z \in \Sigma^*$.

Dann ist eine Verschiebung von P um $|y|$ Stellen möglich. Wir speichern nicht $|y|$, sondern $|x|$.

Die Verschiebetabelle Π

Sei $P = P_1 P_2 \ldots P_m$, $P_i \in \Sigma$.

Setze für $q = 1, \ldots, m$

$$\Pi[q] = \max\{k < q \mid P_1 \ldots P_k = P_{q-k+1} \ldots P_q\}$$

Es ist also $\Pi(q)$ die Länge des längsten Endworts von $P_1 \ldots P_q$, das gleichzeitig Anfangswort von P ist.

Beispiel

$P = a\ n\ a\ n\ a\ s \qquad m = 6$

P	a	n	a	n	a	s
i	1	2	3	4	5	6
Π	0	0	1	2	3	0

$T:\ a\ n\ a\ n\ a\ n\ a\ s\ a\ s$
$P:\ a\ n\ a\ n\ a\ s$

Mismatch bei $q + 1 = 6$ in P.
Verschiebung um $q - \Pi(q) = 5 - \Pi(5) = 2$ liefert

$T:\ a\ n\ a\ n\ a\ n\ a\ s\ a\ s$
$P:\ a\ n\ a\ n\ a\ s$

Die Verschiebetabelle Π (Forts.)

Beachte

Π ist weniger eine Verschiebe-, als eine Wiederaufsetz-Tabelle:
Es trete ein Mismatch bei P_{q+1} auf

$\Rightarrow$ Verschiebung des Muster um $d = q - \Pi(q)$ Stellen
Wiederaufsetzpunkt für Vergleiche ist $q' + 1$, $q' = \Pi(q)$.
Nach Konstruktion von Π stimmt nämlich $P_1 \ldots P_{q'}$ mit dem Test darüber überein.

Der KMP-Algorithmus informell

Sei

- i aktueller Zeiger auf T
- q aktueller Zeiger auf P
- $T_i = P_{q+1}$? aktueller Vergleich
- Schleifeninvariante $P_1 \dots P_q = T_{i-q} \dots T_{i-1}$

Dann

a) $T_i = P_{q+1} \Rightarrow q \leftarrow q+1, \;\; i \leftarrow i+1$

b) $T_i \neq P_{q+1} \Rightarrow q \leftarrow \Pi[q]$;
verschiebe P um $q - \Pi[q]$-Stellen, teste erneut $T_i = P_{q+1}$.

Beispiel

$P = \; a\, n\, a\, n\, a\, s \qquad m = 6$

T	a	n	n	a	n	a	n	a	n	a	s
P	$\underline{a}$	$\underline{n}$	a	n	a	s					
			a	n	a	n	a	s			
				$\underline{a}$	$\underline{n}$	$\underline{a}$	$\underline{n}$	$\underline{a}$	s		
						$\underline{a}$	$\underline{n}$	$\underline{a}$	$\underline{n}$	$\underline{a}$	$\underline{s}$
i	1	2	3	4	5	6	7	8	9	10	11
q	0	1	2	0	1	2	3	4	5	4	5
			0						3		

Der Knuth-Morris-Pratt-Algorithmus

Sei
$\mathcal{A}_\Pi$ Algorithmus zur Berechnung von Π (siehe unten)

Algorithmus
$\mathcal{A}$: KMP-StringMatch(T, n, P, m)=

    $\mathcal{A}_\Pi$
    $q \leftarrow 0$;
    for $i = 1$ **to** n **do**
        while $(q > 0$ **and** $P_{q+1} \neq T_i)$ **do**
                $q \leftarrow \Pi[q]$;
        od
        if $P_{q+1} = T_i$
            then
                    $q \leftarrow q+1$; **fi**
        if $q = m$
            then
                    output $i - m + 1$;
                    $q \leftarrow \pi[q]$;
        fi
    od

Der Knuth-Morris-Pratt-Algorithmus (Forts.)

Zur Korrektheit

- i ist die aktuelle Stelle in T für Test „$T^{i-q} = P$?“
- q ist Länge des gemeinsamen Anfangswortes von P und T^{i-q}.
  In der while-Schleife wird P verschoben, bis $q = 0$ oder q um 1 erhöht werden kann.
- Gilt $q = m$, so $P = T^{i-q} = T^{i-m}$.
  Also wird die Stelle $i - m$ ausgegeben und P verschoben (durch $q \leftarrow \pi[q]$ realisiert).

Also

$\Rightarrow$ Es gilt die Schleifeninvariante:
Die ersten q Buchstaben von T^{i-q} und P stimmen überein.
$\Rightarrow$ $\mathcal{A}$ liefert alle j mit $P = T^j$.

Berechnung der Verschiebetabelle Π

Idee

Zur Berechnung von $\Pi[q]$ muss das maximale k bestimmt werden, so dass $P_1 \dots P_k = P_{q-k+1} \dots P_q$. Dies ist wieder ein (spezielles) String-Matching-Problem.
Also wird $\mathcal{A}_\Pi$ nach der Idee für $\mathcal{A}$ erstellt.

Algorithmus

$\mathcal{A}_\Pi : \Pi[1] \leftarrow 0;\ k \leftarrow 0;$

```
for q = 2 to m do
    while (k > 0 and P_{k+1} ≠ P_q) do
          k ← Π[k];
    od
    if P_{k+1} = P_q
      then k ← k + 1;
    fi
    Π[q] ← k
od
```

Berechnung der Verschiebetabelle Π (Forts.)

Zur Korrektheit

- q ist die aktuelle Position: Berechnung von $\Pi[q]$
- k ist die Länge des aktuellen Wortes x, das eventuell Anfangswort von P und Endwort von $P_1 \dots P_q$ ist.

Also

$\Rightarrow$ Es gilt die Schleifeninvariante:
$\Pi[1], \dots, \Pi[q-1]$ sind korrekt berechnet und die ersten k Buchstaben von P und $P_{q-k} \dots P_{q-1}$ stimmen überein.
$\Rightarrow$ $\mathcal{A}_\Pi$ berechnet Π korrekt.

Zeitbedarf des KMP-Algorithmus

Grobe Zeitabschätzung

while-Schleife: $O(q)$ Durchläufe zu Kosten $O(1)$
for-Schleife: $O(q) + O(1)$ Kosten pro Durchlauf
$\Rightarrow$ Also $T_\mathcal{A}(n,m) = O(n \cdot m)$

Feine Zeitabschätzung

Jede Veränderung von q in der while-Schleife bedeutet eine Verschiebung von P um mindestens eine Stelle. P wird maximal um $n - m$ Stellen verschoben.
Also: Die while-Schleife verursacht insgesamt nur $O(n)$ Kosten. (Abschätzung der **amortisierten Kosten**).
Also: Zeitbedarf des KMP-Algorithmus ohne Kosten für die Berechnung von Π ist $O(n)$.

Analog: Kosten für die Berechnung von Π ist $O(m)$.

Zeitbedarf

des KMP-Algorithmus insgesamt

$$T_\mathcal{A}(n,m) \in O(n+m)$$

Berechnung des endlichen Automaten aus 5.2

Problem

Gegeben: $P = P_1 \dots P_m$, $P_i \in \Sigma$
Gesucht: $M = (Q, \Sigma, q_0, \delta, F)$ mit
$\delta(q_0, x) \in F \text{ gdw } x = yP$ für ein $y \in \Sigma^*$

Idee

Setze Q, q_0, F wie in 5.2.
Berechne δ mit Verschiebetabelle Π zu P.

Setze

$Q = \{0, 1, \dots, m\}$, $q_0 = 0$, $F = \{m\}$

$\delta : Q \times \Sigma \to Q$ ist gegeben durch

$$\delta(0, a) = \begin{cases} 1 & a = P_1 \\ 0 & \text{sonst} \end{cases}$$

und für $q = 1, \dots, m$

$$\delta(q, a) = \begin{cases} q + 1 & a = P_{q+1} \wedge q < m \\ \delta(\Pi(q), a) & \text{sonst} \end{cases}$$

Berechnung des endlichen Automaten aus 5.2 (Forts.)

Zur Korrektheit

Wird $x \in \Sigma^*$ gelesen, so wird P wie beim KMP-Algorithmus immer so weit verschoben, dass der jeweils nächste Buchstabe gelesen werden kann. Dafür muss δ rekursiv berechnet werden:
Ist $\delta(q, a) \neq q+1$, so muss P verschoben werden, also $q \leftarrow \Pi(q)$ gesetzt werden.

Es gilt dann (als Schleifeninvariante) für alle $x \in \Sigma^*$:

$$\delta(q_0, x) = q \quad \text{gdw} \quad P_1 \dots P_q \text{ ist das längste Endwort von } x \text{ unter den } P_1 \dots P_i,\ 0 \leq i \leq m.$$

Also

$$\delta(q_0, x) \in F \;\; \text{gdw} \;\; x = uP, \;\; u \in \Sigma^*$$

Berechnung von M aus 5.2 (Forts.)

Beispiel

$\Sigma = \{a, n, s\} \qquad P = \ a\ n\ a\ n\ a\ s$
Es ergibt sich der aus 5.2 bekannte Automat M mit
$Q = \{0, \ldots, 6\}$ und $\delta : \Sigma \times Q \to Q$

δ	a	n	s
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	1	0	0

Wir verfolgen die Berechnung von δ bei $q = 4$.
Dazu seien die δ-Werte für $q = 0, \ldots, 3$ schon bekannt.
Beachte $\Pi(4) = 2$.
Es ergibt sich

$\delta(4, a) = 5$, da $a = P_{4+1}$
$\delta(4, n) = \delta(2, n) = 0 \qquad \delta(4, s) = \delta(2, s) = 0$

Berechnung von M aus 5.2 (Forts.)

Also

1. Der endliche Automat M kann in der Zeit $O(m)$ berechnet werden.
2. Zu $x, y \in \Sigma^*$ mit $|x| \geq |y|$ lässt sich in der Zeit $O(|x|)$ entscheiden, ob y ein Teilwort von x ist.

5.4 Suche nach mehreren Mustern

Problem

Gegeben: $T, x_1, \dots, x_k \in \Sigma^*$

Gesucht: Alle Stellen in T, an denen ein x_i als Teilwort vorkommt.

Anwendung

Wortersetzungssysteme

Regeln $x_i \to y_i$, $i = 1, \dots, k$
Reduktionsrelation $\to$ auf Σ^*:
$w \to_R v$ gdw
$\exists x_i \to y_i, w_1, w_2 : \quad w = w_1 x_i w_2, v = w_1 y_i w_2$

Beispiel

$R : ba \to ab,\ a^3 \to b,\ b^2 \to e$
$\Rightarrow a\underline{ba}bab \to_R aab\underline{ba}b \stackrel{*}{\to}_R \underline{a^3}b^3 \to_R \underline{b^2}b^2 \to_R \underline{b^2} \to_R e$

Gesucht ist hier also in jedem Reduktionsschritt $w \to_R v$ ein „Redex“ x_i in w.

Idee Aho-Corasick-Algorithmus

- Speichere die $x_1, \dots, x_k$ als Baum (trie).
- Speichere die Π-Werte zu den x_i an den Knoten. Berechne daraus wie eben einen endlichen Automaten $M = (Q, \Sigma, q_0, \delta, F)$ mit $\delta(q_0, z) \in F$ gdw z hat ein x_i als Endwort.

Mehrere Muster: Das Verfahren

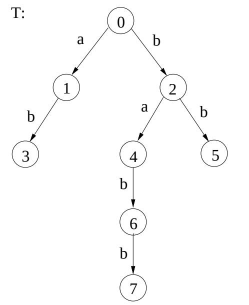
Sei $x_1, \dots, x_k \in \Sigma^*$ gegeben.

Setze

- T trie zu $x_1, \dots, x_k$
  T habe die Knoten $0, 1, \dots, m$; 0 ist die Wurzel.
- $w(i)$= Wort zum Knoten i, $W = \{w(0), \dots, w(m)\}$
- $\Pi(i) = j$ gdw $w(j)$ ist das längste Endwort $\neq w(i)$ von $w(i)$ unter den $w \in W$ (1)
- $E(i) = \{j \mid x_j$ ist Endwort von $w(i)\}$ (2)
- $M = \{Q, \Sigma, q_0, \delta, F)$ mit
  $Q = \{0, 1, \dots, m\}$, $q_0 = 0$
  $F = \{i \mid E(i) \neq \emptyset\}$
  $\delta(i, a) = j$ gdw $w(j)$ ist das längste Endwort von $w(i)a$ in W (3)

Beispiel

Sei $x_1 = ab$, $x_2 = babb$, $x_3 = bb$.



Mehrere Muster: Das Verfahren (Forts.)

Berechnung von Π und δ parallel und von E „breadth-first“ in T.
Für $i \in Q$, $a \in \Sigma$ setze

$$\begin{array}{ll} \Pi(i) = 0 & \text{falls Tiefe}(i) \leq 1 \\ \Pi(i) = \delta(\Pi(j), a) & \text{falls } (j) \xrightarrow{a} (i) \text{ in } T, j \neq 0 \end{array}$$

$$\delta(0,a) = \begin{cases} j & w(j) = a \\ 0 & \text{sonst} \end{cases}$$

$$\delta(i,a) = \begin{cases} j & w(j) = w(i)a \\ \delta(\Pi(i), a) & \text{sonst} \end{cases}$$

$$E(0) = \emptyset$$

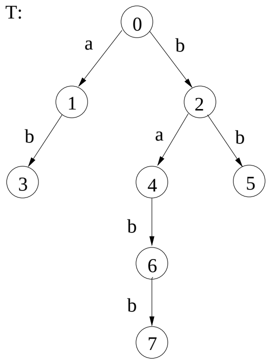
$$E(i) = \begin{cases} E(\Pi(i)) \cup \{j\} & w(i) = x_j \\ E(\Pi(i)) & w(i) \notin \{x_1, \ldots, x_k\} \end{cases}$$

Dann gilt:
a) (1) bis (3) für alle $i \in Q$, $a \in \Sigma$.
b) Für alle $z \in \Sigma^*$ gilt:
$\delta(q_0, z) = i \text{ gdw } w(i)$ ist das längste Endwort von z in W

Beweis:
a) Induktion nach der Tiefe von (i) in T.
b) Dies folgt aus a) durch Induktion nach $|z|$.

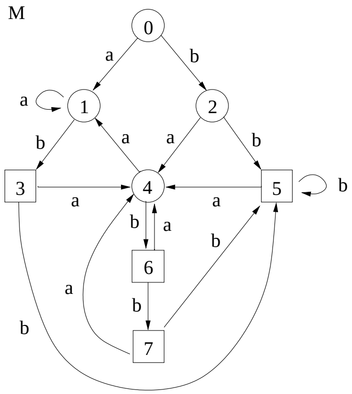
Ein Beispiel

Beispiel
Sei $\Sigma = \{a, b\}$, $x_1 = ab$, $x_2 = babb$, $x_3 = bb$.
Es ergibt sich



i	Π	E
0	0	$\emptyset$
1	0	$\emptyset$
2	0	$\emptyset$
3	2	$\{1\}$
4	1	$\emptyset$
5	2	$\{3\}$
6	3	$\{1\}$
7	5	$\{2,3\}$

δ	a	b
0	1	2
1	1	3
2	4	5
3	4	5
4	1	6
5	4	5
6	4	7
7	4	5



Das Resultat

Satz 5.1.

Sei m die Größe von T (also $m \leq |x_1| + \cdots + |x_k|$).
Man kann in Zeit $O(m)$ einen endlichen Automaten M berechnen, so dass für alle $z \in \Sigma^*$ gilt:

$\delta(q_0, z) = i$ gdw $w(i)$ ist das längste Endwort von z unter den $w \in W$

$\delta(q_0, z) = i \in F$ gdw $E(i) = \{j \mid x_j$ ist Endwort von $z\} \neq \emptyset$

5.5 Das Boyer-Moore-Verfahren

Vorteile

Dies ist in der Praxis das schnellste Verfahren für den Fall, dass

- das Muster P kurz ist,
- P nur wenige Buchstaben aus Σ enthält.

Idee

Verschiebe das Muster P von links nach rechts über den Text T, aber

- vergleiche P mit T^i von rechts nach links
- benutze für die Verschiebung nicht nur Teilwörter von P (wie beim KMP-Verfahren), sondern auch die $a \in \Sigma$.

Verschiebestrategien

Es trete der Mismatch beim Vergleich T_i mit P_j auf:

1. Verschiebe P so weit nach rechts, dass $T_i = P_j$ gilt oder $T^k = P$ nicht sofort ausgeschlossen werden kann. „Bad-Character-Strategie".
2. Verschiebe P in Abhängigkeit von $P_{j+1} \ldots P_m$ „Good-Suffix-Strategie" (analog zum KMP-Verfahren). Vergleiche wieder ab rechtem Ende von P.

Die Bad-Character-Strategie (1)

Sei

$P = P_1 \dots P_m$ das Muster.

Setze für alle $a \in \Sigma$

$$\Delta_1(a) = \begin{cases} m & a \text{ kommt in } P \text{ nicht vor} \\ m - j & P_j = a \text{ und } P_k \neq a \text{ für alle } k > j \end{cases}$$

$\Rightarrow$ Also: Das Endwort von P der Länge $\Delta_1(a)$ ist a-frei.

Berechnung der Verschiebung

v von P falls

$T_{i+k} = P_{j+k}$ für $k = 1, \dots, m - j$

$a = T_i \neq P_j$,

d. h., beim Vergleich $T_i = a$ mit P_j tritt Mismatch auf.

Es gibt 3 Fälle:

a) a tritt in P auf als P_k mit $k > j$, also $m - j > \Delta_1(a)$

$$\begin{array}{lllllllllllllll} T: & \cdots\cdots & a & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ P: & P_1 \;\cdots\; P_j \cdots a \;\cdots\; P_m \\ & \qquad\qquad\quad P_1 \;\cdots\; P_j \cdots a \cdots\cdots\cdots P_m \\ & 1\;\;2\;\;3\;\;4\;\;5\;\;6\;\;7\;\;8\;\;9\;\;10\;\;11\;\;12\;\;13 \end{array}$$

Es ist $P_{k+1} \dots P_m$ a-frei, also auch der darüberliegende Text.
Also kann man P um $v = \Delta_1(a) + 1$ Stellen verschieben.

Die Bad-Character-Strategie (2)

b) a tritt in P auf als P_k mit $k < j$, also $m - j < \Delta_1(a)$

$$\begin{array}{ll} T: & \cdots\cdots\cdots\cdots\cdots\cdots a \cdots\cdots\cdots\cdots\cdots\cdots \\ P: & P_1 \;\cdots\cdot a \cdots P_j \cdots P_m \\ & \qquad\quad P_1 \;\cdots\cdot a \cdots\cdot P_j \cdots P_m \\ & 1\;\;2\;\;3\;\;4\;\;5\;\;6\;\;7\;\;8\;\;9\;\;10\;\;11 \end{array}$$

Es ist $P_{k+1} \dots P_m$ a-frei.
Also kann man P um $v = \Delta_1(a) - (m - j)$ Stellen verschieben.

c) a tritt in P nicht auf, also $m - j < \Delta_1(a) = m$

$$\begin{array}{ll} T: & \cdots\cdots\cdots a \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ P: & P_1 \;\cdots\cdot P_j \cdots\cdots\cdot P_m \\ & \qquad\qquad\quad P_1 \cdots\cdot P_j \cdots\cdots\cdots\cdots P_m \\ & 1\;\;2\;\;3\;\;4\;\;5\;\;6\;\;7\;\;8\;\;9\;\;10\;\;11\;\;12 \end{array}$$

Man kann P um $v = j$ Stellen verschieben.

Also

Es werde T_i mit P_j verglichen.

Gilt $T_i = P_j$, so setze $i \leftarrow i - 1;\;\; j \leftarrow j - 1$.
Gilt $T_i \neq P_j$, so setze $i \leftarrow i + (m - j) + v;\;\; j \leftarrow m$

mit v wie oben. Dies ergibt:

$$\begin{array}{ll} i \leftarrow i + m - j + \Delta_1(a) + 1;\; j \leftarrow m & \text{falls } m - j > \Delta_1(a) \\ i \leftarrow i + \Delta_1(a);\; j \leftarrow m & \text{falls } m - j < \Delta_1(a) \end{array}$$

Ein einfacher Boyer-Moore-Algorithmus

Sei

$P = P_1 \dots P_m$ das Muster.
$T = T_1 \dots T_n$ der Text.

Berechnung von Δ_1

$\mathcal{A}_1$:
for all $a \in \Sigma$ **do**
    $\Delta_1(a) \leftarrow m$
od
for $j = 1$ **to** m **do**
    $\Delta_1(P_j) \leftarrow m - j$
od

Zeitbedarf

$$T_{\mathcal{A}_1}(m, \Sigma) \in \Theta(m + \|\Sigma\|)$$

Ein einfacher Boyer-Moore-Algorithmus (Forts.)

Der BM-Algorithmus

$\mathcal{A}$: BM-einfach(T, n, P, m)=
$\mathcal{A}_1$
$i \leftarrow m;\ j \leftarrow m;$
while $i \leq n$ **do**
    if $T_i = P_j$
        then
            $i \leftarrow i - 1;\ j \leftarrow j - 1$
        else
            $i \leftarrow i + \delta_1(m, j, T_i);$
            $j \leftarrow m$
    fi
    if $j = 0$
        then
            output $i + 1;$
            $i \leftarrow i + m + 1;$
            $j \leftarrow m$
    fi
od

Ein einfacher Boyer-Moore-Algorithmus (Forts.)

Hier ist

$$\delta_1(m,j,a) = \begin{cases} m - j + \Delta_1(a) + 1 & \text{falls } m - j > \Delta_1(a) \\ \Delta_1(a) & \text{falls } m - j < \Delta_1(a) \end{cases}$$

Zeitbedarf von BM-einfach (bei festem Σ)

$T_{\max}(n,m) = O(n \cdot m)$ (Übung)
$T_{\text{mittel}}(n,m) = O(n+m)$ (ohne Beweis)

Die Good-Suffix-Strategie

Man kann ähnlich wie beim KMP-Verfahren eine Tabelle Δ_2 ausrechnen, so dass beim Mismatch $T_i \neq P_j$ gesetzt werden kann

$$i \leftarrow i + \Delta_2(j); \quad j \leftarrow m$$

Berechnung
der Tabelle Δ_2: Siehe z. B.
T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen.
R.S. Boyer, J.S. Moore: A fast string searching algorithm, C.ACM 20 (1977) 762-772.

Bemerkung

Man kann beide Strategien kombinieren, um die größtmögliche Verschiebung zu berechnen.
Die Verwendung der Good-Suffix-Strategie trägt in der Praxis nur wenig zur Schnelligkeit des Verfahrens bei, da selten Endwörter $P_{j+1} \ldots P_m$ mehrfach in P auftreten.
Die 1. Strategie allein hat Zeitbedarf $T_{\max}(n,m) \in \Theta(n \cdot m)$, der Einsatz der 2. Strategie reduziert das zu $T_{\max}(n,m) \in \Theta(n+m)$.

Der Boyer-Moore-Algorithmus

Zusammenfassung beider Strategien

Sei

- $P = P_1 \ldots P_m$ das Muster $P_j, T_i \in \Sigma$
- $T = T_1 \ldots T_n$ der Text $n \geq m$
- δ_1 die Verschiebung nach der Bad-Character-Strategie

$$\delta_1(m,j,a) = \begin{cases} m - j + \Delta_1(a) + 1 & \text{falls } m - j > \Delta_1(a) \\ \Delta_1(a) & \text{falls } m - j \leq \Delta_1(a) \end{cases}$$

- Δ_2 die Verschiebetabelle (analog zum KMP-Verfahren) nach der Good-Suffix-Strategie

Der Boyer-Moore-Algorithmus (Forts.)

Algorithmus

$\mathcal{A}$: BM-StringMatch(T, n, P, m)=

$i \leftarrow m;\ j \leftarrow m;$
while $i \leq n$ **do**
  if $T_i = P_j$
   then
    $i \leftarrow i - 1;\ j \leftarrow j - 1;$
   else
    $i \leftarrow i + \max\{\delta_1(m,j,T_i), \Delta_2(j)\};$
    $j \leftarrow m$
  fi
  if $j = 0$
   then
    output $i + 1;$
    $i \leftarrow i + m + 1;\ j \leftarrow m$
  fi
od

Es werden alle i ausgegeben mit $T^i = P$.

6 Suchverfahren: Backtracking

Viele Probleme lassen sich als Suchprobleme formulieren. Dies gilt z. B. für Optimierungsprobleme.

Beispiele

a) Erfüllbarkeitsproblem SAT

Gegeben:	Formel A.
Frage:	Ist A erfüllbar?
Lösung:	Suche nach einer Belegung $\varphi : V \to \mathbb{B}$ mit $\varphi(A) = 1$.

b) Rucksackproblem RSP

Gegeben:	$P = (p_i)_{i=1,\dots,n}$, $W = (w_i)_{i=1,\dots,n}$, K.
Frage:	Welcher Profit lässt sich maximal erzielen?
Lösung:	Suche zulässigen Vektor $x = (x_i)_{i=1,\dots,n}$, $x_i \in \{0,1\}$, der $\sum_{i=1}^n p_i x_i$ maximiert.

c) Problem des Handlungsreisenden TSP

Gegeben:	$\Delta = (d_{ij})_{i,j=1,\dots,n}$ Distanz-Matrix.
Frage:	Wie lang ist die kürzeste Rundreise?
Lösung:	Suche eine Permutation $\pi = (\pi_1, \dots, \pi_n)$, so dass $d_{\pi_1,\pi_2} + d_{\pi_2,\pi_3} + \cdots + d_{\pi_{n-1},\pi_n} + d_{\pi_n,\pi_1}$ minimal ist.

Fragen

Wie stellt man den Suchraum dar?
Wie sucht man nach der Lösung?

6.1 Der Suchraum

Sei P ein Suchproblem.

Ein **Lösungsansatz** zu P definiert

- den Suchraum S mit Startzustand $z_0 \in S$ und Zielzuständen (goals) $G \subseteq S$,
- eine Übergangsrelation $\varrho \subseteq S \times S$, die angibt, in welche direkten Folgezustände z' man von einem Zustand z aus kommen kann
- ein Verfahren, wie man (gemäß ϱ) von z_0 zu einem $z \in G$ kommen kann.

Aufgaben zur Entwicklung eines Lösungsansatzes:

- Modellierung von S: Was sind Zustände?
- Angabe von ϱ: Welche Suchschritte sind möglich?
- Aufbau und Durchsuchung von S: Wie sucht man effizient? Was muss man zwischenspeichern?

Beachte

S ist häufig endlich, kann aber auch unendlich sein.

Durchsuchung des Suchraums

Modellierung von S (konzeptionell)

Wir modellieren S als Baum, die Knoten tragen einen Zustand als Marke:

- z_0 ist die Wurzel.
- Knoten z hat alle z' als Söhne mit $z \varrho z'$.

Aufbau von S

Zu jedem Zeitpunkt ist nur ein Anfangsbaum T von S aufgebaut (und durchsucht).
Es gibt unterschiedliche Aufbaustrategien:

- Tiefensuche (depth first search).
- Breitensuche (breadth first search).
- Mischformen hierzu: z. B. iterative deepening:
  Baue T depth first bis zur vorgegebenen Tiefe t_0 auf. Falls dabei kein $z \in G$ gefunden wurde, erhöhe t_0 zu t_1 und baue T bis zur Tiefe t_1 auf. Iteriere diesen Prozeß.
- Heuristische Suche: Schätze die Erfolgschancen der Suche unter jedem Blatt von T ab und baue T unter dem besten Knoten weiter auf.

Bemerkungen zur Suchstrategie

Heuristische Suchverfahren werden später besprochen.

Breitensuche ist häufig ineffizient, weil nahe der Wurzel selten Zielknoten liegen.

Tiefensuche ist gefährlich, falls der Suchbaum unendliche Tiefe hat.

Dann empfiehlt sich „iterative deepening“.

Zentraler Punkt bei der Tiefensuche: Backtracking.

Gehe auf dem aktuellen Ast rauf, bis zu einem Knoten, der noch nicht voll expandiert ist, d. h., einen nicht-aufgebauten Sohn hat. Starte hier die Tiefensuche erneut.

Beschränkung der Suche.

Manchmal kann man abschätzen, dass unter dem Knoten z kein Zielknoten liegt. Dann die Suche hier abbrechen, also den Suchbaum hier nicht weiter entwickeln.
$\Rightarrow$ Beschränkungsprädikat $B(z)$ verwenden.

6.2 Das n-Damen-Problem

Problem

Gegeben: Feld der Größe $n \times n$ (Schachbrett).

Gesucht: Positionierung von n Damen so, dass keine die andere bedroht.

Modellierung

Dame i steht in Zeile i und Spalte x_i $(i, x_i = 1, \ldots, n)$.
„i bedroht j“ gdw $|i-j| = |x_i - x_j| \vee x_i = x_j$ (Übung!).

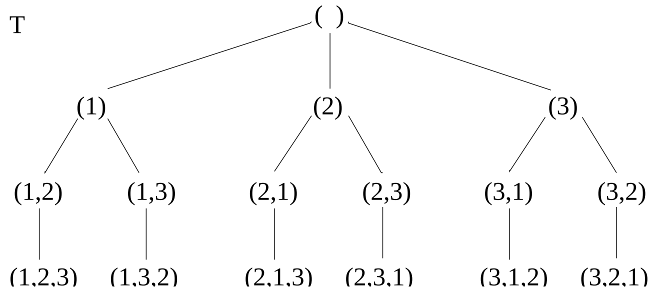
Wir positionieren die Damen in der Reihenfolge $k = 1, \ldots, n$.
Der volle Suchraum besteht dann aus

- Knoten der Form: $z = (x_1, \ldots, x_k)$ mit
  $k \geq 0$, $1 \leq x_i \leq n$, $x_i \neq x_j$ für $i \neq j$.
- Wurzel: $z_0 = ()$, also $k = 0$.
- Kanten: $(x_1, \ldots, x_k) \to (x_1, \ldots, x_k, x_{k+1})$ mit
  $x_{k+1} \notin \{x_1, \ldots, x_k\}$.
- Zielknoten: $(x_1, \ldots, x_n)$ mit
  $|i-j| \neq |x_i - x_j|$ für alle $1 \leq i < j \leq n$.

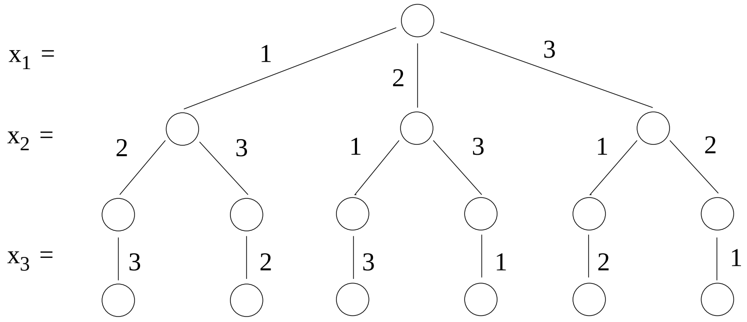
Modellierung des n-Damen-Problems (Forts.)

Beispiel $n = 3$

Der volle expandierte Suchbaum ist



oder kompakter als Bild



Modellierung des $\mathbf{n}$-Damen-Problems (Forts.)

Interpretation des Knotens $z = (x_1, \ldots, x_k)$

Es sind k Damen gesetzt, und zwar Dame i auf das Feld (i, x_i) für $i = 1, \ldots, k$.

Beschränkung der Suche

Setze $B(z)$ $\;\mathrm{gdw}\;$ z enthält eine Bedrohung, also

$B(x_1, \ldots, x_k)$ $\;\mathrm{gdw}\;$ es gibt $1 \leq i < j \leq k$ mit
$|i - j| = |x_i - x_j|$ oder $x_i = x_j$.

Alle Knoten z im Baum T erfüllen $\neg B(z)$. Zum Test, ob der Sohn $z' = (x_1, \ldots, x_{k-1}, x_k)$ von $z = (x_1, \ldots, x_{k-1})$ in T aufgenommen wird, hat man also

$$B(z') \;\;\mathrm{gdw}\;\; |i - k| = |x_i - x_k| \vee x_i = x_k \text{ für ein } 1 \leq i < k$$

Der Suchalgorithmus (informell)

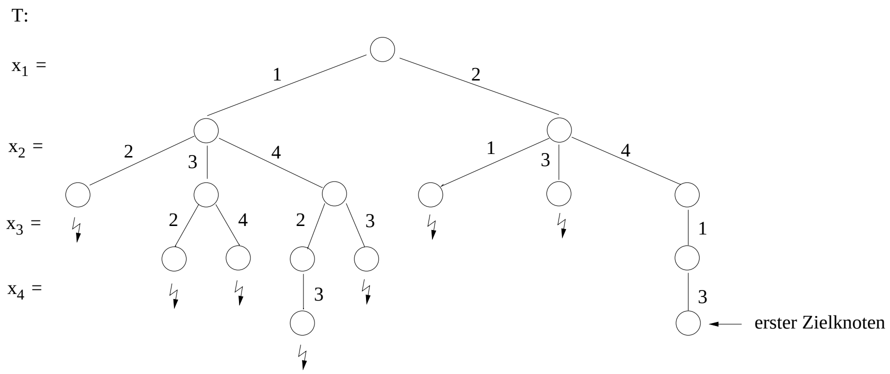
Algorithmus $\mathcal{A}$:

Baue den Suchbaum depth first von links nach rechts auf. Stoppe den Aufbau unter z, falls $B(z)$ gilt.

Gib z aus, falls z ein Zielknoten ist.

Beispiel $n = 4$

Es ergibt sich (Abbruch bei z weil $B(z)$ gilt, mit $\lightning$ markiert)



Die Suche wurde hier nach dem Finden des ersten Zielknotens eingestellt. Der gefundene Zielknoten ist (als Schachbrett dargestellt)

	0		
			0
0			
		0	

Der Algorithmus Damen

Sei ein Algorithmus für B gegeben.

Algorithmus

$\mathcal{A}$: Damen(n)=
  suche($1, n, x$)
mit suche(k, n, x)=

```
Ref: x;  x_k ← 1;
while x_k ≤ n do
    if not B(x_1, ..., x_k)
      then
        if k = n
          then
              output(x_1, ..., x_n);
              x_n ← n
          else
              suche(k + 1, n, x)
        fi fi
    x_k ← x_k + 1
od
```

Es werden alle Lösungen ausgegeben.

Beachte

Das Backtracking wird hier durch die rekursive Programmierung automatisch realisiert. Bei nicht-rekursiver Programmierung muss das Backtracking explizit programmiert werden.

Beispiel: Rechnung für $\mathbf{n} = 4$

Übung

Man verfolge die Rechnung für $n = 4$. Es ergibt sich als Anfang der Rechnung.

k	x_k	
1	1	
2	$1, 2, 3$	
3	$1, 2, 3, 4$	$\lightning$
2	4	
3	$1, 2$	
4	$1, 2, 3, 4$	$\lightning$
3	$3, 4$	$\lightning$
2		$\lightning$
1	2	
2	$1, 2, 3, 4$	
3	1	
4	$1, 2, 3$	$+$

Hier bedeutet:

$\lightning$ Backtrack-Punkt.
Die k-te Dame konnte nicht gesetzt werden.

$+$ Lösung gefunden.
Alle Damen wurden gesetzt: $x_1 = 2$, $x_2 = 4$, $x_3 = 1$, $x_4 = 3$.

Übung

Man verfolge die Rechnung für $n = 3$ und $n = 5$.
Gibt es da Lösungen?

Zeitbedarf für $\mathcal{A}$=Damen

$\mathcal{A}$ baut den vollen Suchbaum T auf.
T hat Tiefe n, auf Tiefe k gibt es maximal n^k Knoten, jeder Knoten verursacht beim B-Test $\Theta(k)$ Kosten. Also

$$T_{\mathcal{A}}(n) \in \sum_{k=1}^{n} O(k \cdot n^k) \subseteq O(n \cdot n^n)$$

Also: $\mathcal{A}$ hat Zeitbedarf

$$T_{\mathcal{A}}(n) \in O(n^{n+1})$$

Beachte hierzu

Die Abschätzungstechniken aus Abschnitt 1.3 liefern

$$\sum_{k=1}^{n} k \cdot m^k \approx (\ln m)^{-1} \cdot n \cdot m^n$$

Dies gilt wegen $m^k = e^{k \cdot \ln m}$ und

$$\begin{aligned} \int_0^n x e^{cx} dx &= (x - \frac{1}{c})\frac{1}{c} e^{cx} \Big|_0^n = (n - \frac{1}{c})\frac{1}{c} e^{cn} + \frac{1}{c^2} \\ &\in \begin{cases} \Theta(n e^{cn}) & c > 0 \\ \Theta(1) & c < 0 \end{cases} \end{aligned}$$

6.3 Graphen-Färbung

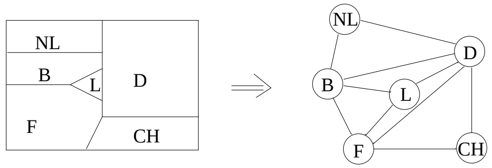
Definition 6.1

Sei $G = (V, E)$ ein ungerichteter Graph, $m \in \mathbb{N}$ und φ eine Abbildung $\varphi : V \to \{1, \ldots, m\}$. φ heißt eine **m-Färbung** von G, falls $\varphi(u) \neq \varphi(v)$ für alle $u, v \in V$ mit $(u, v) \in E$ gilt.

Anwendungsbeispiel

Frage: Ist jede Landkarte 4-färbbar, so dass Länder mit gemeinsamer Grenze unterschiedlich gefärbt sind?
Transformation in Graphen-Problem $G = (V, E)$:
V=Menge der Länder.
$E = \{(u, v) \mid u, v$ haben gemeinsame Grenze$\}$.
Ist G 4-färbbar?

Beispiel: Westliche Nachbarn Deutschlands



Übung: G ist 4-färbbar, aber nicht 3-färbbar.

Satz 6.1. [Lösung des 4-Farben-Problems]

Jeder planare Graph ist 4-färbbar.

Dies gilt nicht für jeden Graphen!

Das Suchproblem

Problem

Gegeben: Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$, $m \in \mathbb{N}$.
Frage: Ist G m-färbbar?

Modellierung

Wir färben die Knoten $k = 1, \ldots, n$. Dabei sind m Farben möglich. Der volle Suchbaum besteht dann aus

- Knoten: $z = (x_1, \ldots, x_k)$ mit
  $0 \leq k \leq n$, $1 \leq x_i \leq m$, $\neg B(z)$.

- Wurzel: ( ), also $k = 0$.

- Kanten: $z = (x_1, \ldots, x_k) \to z' = (x_1, \ldots, x_k, x_{k+1})$
  mit $\neg B(z')$.

- Zielknoten: $z = (x_1, \ldots, x_n)$ mit $\neg B(z)$.

- Beschränkungsprädikat: Sei $z = (x_1, \ldots, x_k)$,

$$B(z) \quad \text{gdw} \quad x_i = x_j \text{ für ein } i, j \text{ mit } 1 \leq i < j \leq k \text{ und } (i, j) \in E.$$

Der Färbe-Algorithmus

Algorithmus

```
A : GraphFärbung(V, E, n, m)=
      x_1 ← 1;
      färbe(2, n, m, x)
mit färbe(k, n, m, x)=
        Ref: x;
        for x_k = 1 to m do
            if not B(x_1, ..., x_k)
              then
                   if k = n
                     then
                            output(x_1, ..., x_n)
                     else
                            färbe(k + 1, n, m, x)
                   fi
            fi
        od
```

Es werden alle m-Färbungen ausgegeben.

Bemerkungen

Beachte zu Knoten 1

Man kann für jede Färbung $x_1 = 1$ voraussetzen. (Dies ist durch Umbenennung der Farben stets erreichbar.)

Beachte zum Beschränkungsprädikat B

Es gilt $\neg B(z)$ für alle erzeugten Knoten z. Um den Sohn $z' = (x_1, \ldots, x_{k-1}, x_k)$ von $z = (x_1, \ldots, x_{k-1})$ zu erzeugen, setze man also

$$B(x_1, \ldots, x_k) \quad \text{gdw} \quad x_k = x_i \text{ für ein } i \text{ mit } 1 \leq i < k \text{ und } (i, k) \in E.$$

Zeitbedarf von $\mathcal{A}$

Sei E als $n \times n$-Matrix (Inzidenzmatrix) gegeben.

Der Algorithmus $\mathcal{A} = \text{GraphFärbung}(V, E, n, m)$ baut den vollen Suchbaum T auf.

T hat Tiefe n, auf Tiefe k gibt es maximal m^k Knoten, jeder Knoten verursacht beim B-Test Kosten $O(k)$.

Der Zeitbedarf von $\mathcal{A}$ lässt sich also wie in Abschnitt 6.2 abschätzen durch

$$T_{\mathcal{A}}(n, m) \in \sum_{k=1}^{n} O(k \cdot m^k) \subseteq O(n \cdot m^n)$$

6.4 Das 0/1-Rucksack-Problem

Wir betrachten das Optimierungs-, nicht das Entscheidungsproblem.

Problem

Gegeben: $p = (p_1, \ldots, p_n)$, $w = (w_1, \ldots, w_n)$, K wie früher.

Gesucht: $x = (x_1, \ldots, x_n) \quad x_i \in \{0, 1\}$, so dass

$\sum_{i=1}^{n} p_i x_i$ maximal unter der Nebenbedingung

$$\sum_{i=1}^{n} w_i x_i \leq K.$$

Es gibt verschiedene Möglichkeiten, dies Problem als Suchproblem zu modellieren. Wir betrachten zwei davon.

1. Modellierung

Idee

Wähle im k-ten Schritt ein neues Element $y_i \in \{1, \ldots, n\}$ aus und packe es in den Rucksack.

Als **Suchbaum** ergibt sich

- Knoten: $(y_1, \ldots, y_k)$ mit
  $0 \leq k \leq n, 1 \leq y_i \leq n, \neg B(y_1, \ldots, y_k)$.
- Wurzel: ( ), also $k = 0$.
- Kanten: $(y_1, \ldots, y_k) \to (y_1, \ldots, y_k, y_{k+1})$ mit
  $\neg B(y_1, \ldots, y_{k+1})$.
- Beschränkungsprädikat:

$$B(y_1, \ldots, y_k) \quad \text{gdw} \quad y_i = y_j \text{ für ein } i \neq j \text{ oder } \sum_{i=1}^{k} w_{y_i} > K$$

- Zielknoten: alle Blätter des Suchbaumes.
  Gesucht ist ein optimaler Zielknoten.

2. Modellierung

Idee

Man entscheidet im k-ten Schritt, ob das Element $k \in \{1, \ldots, n\}$ in den Rucksack aufgenommen werden soll, d. h. $x_k = 0$ oder $x_k = 1$ gelten soll.

Dies ergibt den **Suchbaum**

- Knoten: $x = (x_1, \ldots, x_k)$ mit
$$0 \leq k \leq n, \; x_i \in \{0,1\}, \; \sum_{i=1}^{k} w_i x_i \leq K. \qquad (*)$$
- Wurzel: ( ), also $k = 0$.
- Kanten: $(x_1, \ldots, x_k) \rightarrow (x_1, \ldots, x_k, x_{k+1})$ mit
  Sohn erfüllt (*).

Der Baum hat also die Tiefe n und daher maximal 2^n Blätter. Die Blätter beschreiben die zulässigen Rucksackpackungen.

Beschränkungsprädikat: Der Ansatz

Bei Optimierungsproblemen setzt man häufig
$B(x)$ gdw unter x kann kein Zielknoten liegen, der besser ist als der bisher beste besuchte Zielknoten.

Das Beschränkungsprädikat B (1)

Setze für jeden Knoten $x = (x_1, \ldots, x_k)$

$$\text{APROF}(x) = \sum_{i=1}^{k} p_i x_i \qquad \text{aktueller Profit}$$

$$\text{AKOST}(x) = \sum_{i=1}^{k} w_i x_i \qquad \text{aktuelle Kosten}$$

Also gilt für Kanten
$x = (x_1, \ldots, x_k) \rightarrow x' = (x_1, \ldots, x_k, x_{k+1})$

$$\begin{aligned} \text{APROF}(x') &= \text{APROF}(x) + p_{k+1} x_{k+1} \\ \text{AKOST}(x') &= \text{AKOST}(x) + w_{k+1} x_{k+1} \end{aligned}$$

Sei $\text{OPT} = \max\{\text{APROF}(z) \mid z \text{ schon besuchtes Blatt}\}$

Man kann die Suchbeschränkung unterschiedlich scharf wählen.

Das Beschränkungsprädikat B (2)

1. Realisierung

Setze

$$B(x) \text{ gdw } \mathsf{AKOST}(x) > K$$

Diese Wahl von B berücksichtigt $\mathsf{APROF}(x)$ gar nicht. Wir schätzen jetzt die Profite unterhalb von x nach oben ab. Liefert diese Schätzung einen Wert $<$ OPT, so kann die Suche abgebrochen werden.

2. Realisierung

Setze für $x = (x_1, \ldots, x_k)$

$$B(x) \text{ gdw } \mathsf{AKOST}(x) > K \vee \mathsf{APROF}(x) + \sum_{i=k+1}^{n} p_i < \mathsf{OPT}$$

Die Summe

$$\sum_{i=k+1}^{n} p_i$$

ist eine schwache Abschätzung der möglichen Profite unterhalb von x.

Das Beschränkungsprädikat B (3)

3. Realisierung

Bessere Abschätzung des möglichen Profits unter x ist

$$P(x) = \max\left\{ \sum_{i=k+1}^{n} p_i y_i \mid 0 \leq y_i \leq 1, \sum_{i=k+1}^{n} w_i y_i \leq K - \mathsf{AKOST}(x) \right\}$$

Dies ist mit der Greedy-Methode in Zeit $O(n-k)$ berechenbar, falls gilt:

Voraussetzung

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \text{ für } i = 1, \ldots, n-1$$

Setze

$$B(x) \text{ gdw } \mathsf{AKOST}(x) > K \vee \mathsf{APROF}(x) + P(x) < \mathsf{OPT}$$

Der Suchalgorithmus OptRSP

Ein Algorithmus für B sei gegeben.

Algorithmus

$\mathcal{A}$: OptRSP(p, w, n, K)=
  OPT $\leftarrow 0$;
  $\vec{y} \leftarrow (0, \ldots, 0)$;   % $\vec{y} = (y_1, \ldots, y_n)$
  suche$(1, 0, 0, x, y,$ OPT$)$;
  STOP: output (OPT, $\vec{y}$)

mit suche(k, AKOST, APROF, x, y, OPT)=

```
Ref: x, y, OPT;
for x_k = 0 to 1 do
    AK ← AKOST + w_k x_k;  AP ← APROF + p_k x_k;
    if not B(x_1, ..., x_k)
      then
          if k < n
            then suche(k + 1, AK, AP, x, y, OPT)
             else
                 if AP > OPT
                   then
                        y⃗ ← x⃗;  OPT ← AP
                 fi fi fi
od
```


Zeitbedarf für $\mathcal{A}$ = OptRSP

Der Algorithmus $\mathcal{A}$ baut im schlimmsten Fall den Suchbaum T ganz auf.

T hat Tiefe n, auf Tiefe k gibt es 2^k Knoten, jeder Knoten verursacht $O(n-k)$ Kosten. Also gilt für den Zeitbedarf im schlechtesten Fall

$$T_{\mathcal{A}}(n) \in \sum_{k=1}^{n} O((n-k) \cdot 2^k)$$

Es gilt mit $c = \frac{1}{\ln 2}$ (vergleiche Abschnitt 6.2)

$$\begin{aligned} \sum_{k=1}^{n} (n-k) \cdot 2^k &= n \sum_{k=1}^{n} 2^k - \sum_{k=1}^{n} k \cdot 2^k \\ &\approx n \cdot 2^{n+1} - c \cdot n \cdot 2^n \\ &= (2-c) n \cdot 2^n \in \Theta(n \cdot 2^n) \end{aligned}$$

Also

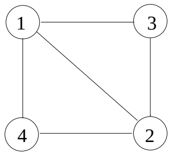
$$T_{\mathcal{A}}(n) \in O(n \cdot 2^n)$$

6.5 Hamilton-Zyklen

Definition 6.2

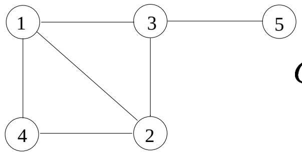
$G = (V, E)$ ein ungerichteter Graph, $V = \{1, 2, \ldots, n\}$.
Ein **Hamilton-Zyklus** ist ein geschlossener Pfad z durch G, der jeden Knoten genau einmal besucht, genauer
$z = (v_1, v_2, \ldots, v_n)$ mit $v_i \neq v_j$ für $1 \leq i, j \leq n$, $v_1 = 1$,
$(v_i, v_{i+1}) \in E, (v_n, v_1) \in E$

Beispiele

a) G: 

G hat zwei Hamilton-Zyklen
$z_1 = (1, 3, 2, 4)$
$z_2 = (1, 4, 2, 3)$

Also liegt die Kante $(1, 2)$ auf keinem H-Zyklus.

b) G: 

G hat keinen H-Zyklus

Hamilton-Zyklen (Forts.)

Probleme U-Hamilton

Gegeben: $G = (V, E), V = \{1, \ldots, n\}$.

Gesucht: Alle Hamilton-Zyklen in G.

Beachte

Man kann dies Problem auch für gerichtete Graphen betrachten: Problem G-Hamilton.

Der Suchraum

Modellierung

Baue für $k = 1, 2, \ldots, n$ alle Wege der Länge $k-1$ ab $v_1 = 1$ auf, die Anfangswege eines Hamilton-Zyklus sein können. Dies ergibt den Suchbaum T mit

- Knoten: $z = (x_1, \ldots, x_k)$ mit
  $1 \leq k \leq n,\ x_i \neq x_j$ für $i \neq j$,
  $(x_i, x_{i+1}) \in E$, $1 \leq x_i \leq n$, $x_1 = 1$.
- Wurzel: (1), also $k = 1$.
- Kanten: $z = (x_1, \ldots, x_k) \to z' = (x_1, \ldots, x_k, x_{k+1})$
  mit $\neg B(z')$.
- Beschränkungsprädikat:

$$B(z) \text{ gdw } x_i = x_j \text{ für ein } i \neq j \text{ oder } (x_i, x_{i+1}) \notin E.$$

- Zielknoten: $z = (x_1, \ldots, x_k)$ mit $(x_n, x_1) \in E, \neg B(z)$.

Beachte

Sei $z = (x_1, \ldots, x_k) \to z' = (x_1, \ldots, x_k, x_{k+1})$ und es gelte schon $\neg B(z)$. Dann gilt

$$B(z') \text{ gdw } x_i = x_{k+1} \text{ für ein } 1 \leq i \leq k \text{ oder } (x_k, x_{k+1}) \notin E$$

Dies ist in Zeit $O(k)$ testbar.

Der Suchalgorithmus Hamilton

Algorithmus

```
A : Hamilton(V, E, n)=
      x_1 ← 1;
      sucheweg(2, n, x)
mit sucheweg(k, n, x)=
      Ref: x;
      for x_k = 1 to n do
          if not B(x_1, ..., x_k)
            then
                if k < n
                  then sucheweg(k + 1, n, x)
                  else
                      if (x_n, 1) ∈ E
                        then output (x_1, ..., x_n)
                      fi
                fi
          fi
      od
```

Zeitbedarf für $\mathcal{A}$=Hamilton

Wie beim n-Damen-Problem ergibt sich

$$T_{\mathcal{A}}(n) \in O(n^{n+1})$$

7 Heuristische Suchverfahren

Problemstellung

Gegeben: Suchbaum S mit Zielknoten wie in Kapitel 6.

Frage: Wie findet man schnell einen (alle, den „besten") Zielknoten?

Zentrale Fragen (lokal)

- Welcher Knoten z soll als nächster expandiert werden?
- Sollen alle Söhne von z gleichzeitig erzeugt werden?
- Wie steuert man dies „wissensbasiert" und wie vermeidet man so die blinde Suche?

Beachte

T ist zu jedem Zeitpunkt erst partiell aufgebaut.

7.1 Die Modellierung der Suche

Sei

S der (abstrakt definierte) Suchbaum.
T ein Anfangsbaum von S.

Definition 7.1

T heißt ein **expandierter Anfangsbaum** von S, falls gilt:
Ist v ein innerer Knoten von T, so liegen alle seine Söhne in S auch in T. Die Blätter von T heißen die **lebenden Knoten von** T. Der Baum T wird am Blatt v dadurch **expandiert**, dass alle S-Söhne von v auch Söhne von v in T werden.

Also

T ist an den inneren Knoten voll aufgebaut.
T kann nur bei Blättern weiter aufgebaut werden.
Wird ein Blatt expandiert, so werden alle Söhne aufgebaut.

Beachte

Die Effizienz der Suche hängt stark davon ab, welcher der lebenden Knoten in jedem Schritt expandiert wird. Man sollte den (einen) „richtigen" Knoten expandieren, also den (einen), unter dem ein gesuchter Zielknoten liegt.

Verwaltung der lebenden Knoten

Suchheuristik (Auswahl des nächsten lebenden Knotens)
und Verwaltung der lebenden Knoten

a) „first-in-first-out": Verwaltung als Schlange
b) „last-in-first-out": Verwaltung als Keller
c) „best-first": Verwaltung als Min-Heap
d) ...

Beachte

a) und b) führen zu „blinder" Suche.
c) benötigt eine Ordnung auf den Knoten v, realisiert durch ein Gewicht $\varphi(v)$. Das Blatt mit dem kleinsten Gewicht ist das beste.
$\Rightarrow$ wissensbasierte Suche

Fragen

Woher kommt das Wissen?
Wie fasst man es algorithmisch?

Die Strategie c) heißt **Branch-and-Bound**

$\Rightarrow$ beschränkte Tiefensuche, durch Schranken gesteuert.

7.2 Ein Beispiel: Das $\mathbf{n} \times \mathbf{n}$-Puzzle

Das $\mathbf{n} \times \mathbf{n}$-Puzzle

Gegeben: ist ein $n \times n$-Feld, $m = n^2 - 1$ Steine mit Nummern $1, \ldots, n^2 - 1$. Jeder Stein liegt auf einem Feld, ein Feld ist leer.

Aufgabe: Man verschiebe die Steine so, dass die Steine sortiert liegen (zeilenweise von links nach rechts aufsteigend) und das letzte Feld das Leerfeld ist.

Beispiel $n = 3$

Start (z. B.)

3	1	4
2		6
5	7	8

Ziel

1	2	3
4	5	6
7	8	

Das $\mathbf{n} \times \mathbf{n}$-Puzzle (Forts.)

Modellierung

- Nummerierung der Felder: zeilenweise von links nach rechts.

- Konfiguration: $k[1, \ldots, n^2]$ mit
  $k[i] = j \text{ gdw }$ Stein j liegt auf Feld i, für $j > 0$.
  $k[i] = 0 \text{ gdw }$ Feld i ist das Leerfeld.

- Zug: überführt das Leerfeld nach rechts, links, oben oder unten.

Beispiel $n = 3$

$k = (3, 1, 4, 2, 0, 6, 5, 7, 8) \rightarrow k' = (3, 1, 4, 2, 6, 0, 5, 7, 8)$

$$\boxed{\begin{array}{ccc} 3 & 1 & 4 \\ 2 & & 6 \\ 5 & 7 & 8 \end{array}} \rightarrow \boxed{\begin{array}{ccc} 3 & 1 & 4 \\ 2 & 6 & \\ 5 & 7 & 8 \end{array}}$$

Steuerung der Suche

Problem

Gegeben: n, Startkonfiguration k.

Gesucht: Weg zur Zielkonfiguration (falls existent).

Suchbaum

- Knoten: Knoten v mit Marke k_v, k_v eine Konfiguration.

- Kanten: $v \rightarrow v'$, so dass $k_{v'}$ aus k_v mit einem Zug erreichbar (jeder Knoten hat max. 4 Söhne).

- Zielknoten: v mit k_v=Zielkonfiguration.

Beachte

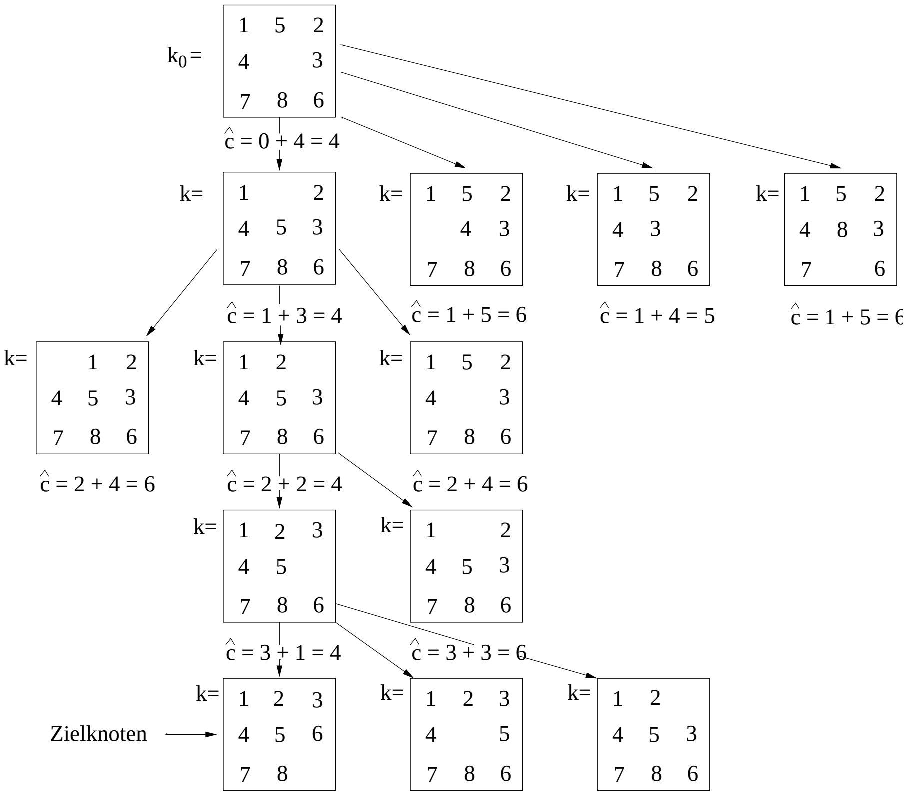
Es gibt Knoten $v \neq v'$ mit $k_v = k_{v'}$.
S ist unendlich.

Suchheuristik

Setze $\hat{g}(v) = ||\{i \mid k_v[i] \neq i \text{ und } k_v[i] \neq 0\}||$
$=$ Anzahl der Steine, die noch falsch liegen
$\hat{c}(v) = \hat{g}(v) +$ Tiefe von v in T
$v \leq v' \text{ gdw } \hat{c}(k_v) \leq \hat{c}(k_{v'})$

Wähle für den nächsten Expansionsschritt das kleinste v unter den lebenden Knoten von T.

Beispiel einer konkreten Suche ($\mathbf{n = 3}$)



Beachte

Von S wurde nur **ein** Ast aufgebaut.

Frage

Wie kann man dieses Vorgehen verallgemeinern?

Bemerkungen zur Modellierung beim $\mathbf{n} \times \mathbf{n}$-Puzzle

Die Modellierung ist nicht optimal:

- Der Suchraum ist unendlich, da jeder Sohn v' eines Knotens v einen Sohn v'' besitzt, dessen Konfiguration der von v entspricht, d. h. $k_v = k_{v''}$.
- Die Suchheuristik $\hat{c}$ ist sehr einfach gewählt.

Übung

- Modellierung mit endlichem Suchraum durch Beschränkungsprädikat angeben.
- Bessere Suchheuristik $\hat{c}$ angeben.

7.3 LC-Suche

Systematisierung der Branch-and-Bound-Suche.

Setze für Knoten v im Suchbaum

$h(v)$ = bisherige Kosten für das Erreichen von v.
$\hat{g}(v)$ = Abschätzung der Kosten für das Erreichen eines Zielknotens unter v.
$\hat{c}(v) = f(h(v), \hat{g}(v))$
$v \leq v' \text{ gdw } \hat{c}(v) \leq \hat{c}(v')$

Also

a) $h(v)$=Tiefe von v, $f(y,z) = -y$, $\hat{g}(v) = 0$.
$\Rightarrow$ Es entsteht die Tiefensuche.

b) $f(y,z) = z$, $\hat{g}(v) =$ Tiefe von v.
$\Rightarrow$ Es entsteht die Breitensuche.

c) $f, h, \hat{g}$ geeignet (Einbau von Wissen über das Suchproblem).
$\Rightarrow$ Es entsteht die heuristische (wissensbasierte) Suche.
Häufig wird $f(y,z) = y + z$ gewählt.

d) Im Beispiel $n \times n$-Puzzle.
$h(v)$=Tiefe von v, $f(y,z) = y + z$,
$g(v)$=Anzahl der noch falsch plazierten Steine.

Beachte

LC steht für „least cost“.

Algorithmus für LC-Branch-and-Bound

Eingabe

- Suchbaum S (konzeptionell) mit Knotenmenge V und Zielknotenmenge $Z \neq \varnothing$
- Rangfunktion $\hat{c} : V \to \mathbb{Z}$
- Beschränkungsprädikat B (wie in Kap. 6)

Ausgabe

Anfangsbaum T von S bis erster Zielknoten v^* gefunden ist.

Algorithmus für LC-Branch-and-Bound (Forts.)

Algorithmus

$\mathcal{A}$: LC-BB=

$T \leftarrow$ Baum, der nur aus der Wurzel r von S besteht;
$v^* \leftarrow$ nil
$V_0 \leftarrow \{v \mid r \rightarrow v \text{ in } S\}$ % Menge der lebenden Knoten
$V_1 \leftarrow V_0$ % Menge der „neuen“ Knoten

repeat

if es gibt $v \in V_1 \cap Z$
then $v^* \leftarrow v$
else
$v \leftarrow$ Knoten mit kleinstem $\hat{c}$-Wert in V_0
$V_0 \leftarrow V_0 - \{v\}$
if **not** $B(v)$
then
$V_1 \leftarrow \{v' \mid v \rightarrow v' \text{ in } S\}$
$V_0 \leftarrow V_0 \cup V_1$
expandiere T bei v
fi
fi

until $v^* \neq$ nil

Bemerkung zu LC-BB

Termination

- Ist S endlich, so wird S durchlaufen, bis ein Zielknoten gefunden ist.
- Ist S unendlich, so muss LC-BB nicht terminieren. Man benötigt weitere Voraussetzungen dafür, dass ein $v^* \in Z$ gefunden wird.

Kosten

Kosten entstehen durch

- Berechnung der $\hat{c}$-Werte $\hat{c}(v)$,
- Berechnung des Beschränkungsprädikats $B(v)$,
- Verwaltung der Menge V_0.

Zu B und $\hat{c}$: Sie sollten „scharf“ sein, so dass die Voraussetzungen b), c) in Voraussetzung LC (siehe unten) erfüllt sind. Aber sie sollten auch leicht berechenbar sein.

Zu V_0: Hier bietet sich ein Heap als Datenstruktur an.

Korrektheit von LC-BB

Voraussetzung LC

a) Ist S unendlich, so hat jeder Knoten nur endlich viele Söhne und zu jedem $k \in \mathbb{Z}$ gibt es nur endlich viele $v \in V$ mit $\hat{c}(v) \leq k$.

b) Es gibt eine Bewertung der Zielknoten $c : Z \to \mathbb{Z}$ mit
$\hat{c}(v) \leq c(u)$ falls $v \in V$, u Zielknoten unter v
$\hat{c}(u) = c(u)$ falls $u \in Z$.

c) Gilt $B(v)$, so liegt unter v kein Zielknoten.

d) $Z \neq \varnothing$.

Satz 7.1.

Gilt die Voraussetzung LC, so hält LC-BB mit Ausgabe $v^* \in Z$.

Beweis:

Wegen a) ist V_0 stets endlich und jedes $v \in V$, das einmal in V_0 wandert, wird hier auch einmal minimal und damit aus V_0 gestrichen.

Es gilt die Schleifeninvariante: Jedes $u \in Z$ liegt unter einem $v \in V_0$.

LC-BB führt also eine modifizierte Breitensuche in S aus und findet daher einen Zielknoten.

LC-Suche bei Optimierungsproblemen

Beachte

Bei Minimierungsproblemen sucht man nicht einen beliebigen, sondern einen kleinsten Zielknoten. Wie garantiert man, dass man den findet?

Ansatz

Ein minimales $u \in Z$ ist gefunden, sobald das erste $u \in Z$ aktuell (also minimal in V_0) wird.

Name des resultierenden Algorithmus

A^*-Algorithmus von Nilsson.

Bemerkungen

Zur Voraussetzung LC

Wir setzen dies ab jetzt wieder voraus.

Es wurde die Bedingung b) bisher nicht wirklich benutzt, z. B. für die Korrektheit von LC-BB. Die Rangfunktion $\hat{c} : V \to \mathbb{Z}$ diente nur zur Definition der Suchheuristik. Es wird b) jetzt auch für die Korrektheit benötigt.

Zu Maximierungsproblemen

Hier wird ein größter Zielknoten gesucht.
Reduktion auf ein Minimierungsproblem:

1. Multipliziere die Werte $\hat{c}(v)$, $c(u)$ mit -1 oder
2. ersetze in allen Bedingungen
   $\leq$, kleinstes, ... durch $\geq$, größtes ...

Formulierung der A^*-Algorithmus hier nur für Minimierungsprobleme.

Der A^*-Algorithmus (1)

Eingabe

- Suchbaum S mit Knotenmenge V und Zielknotenmenge Z
- Bewertung der Zielknoten $c : Z \to \mathbb{Z}$
- Rangfunktion $\hat{c} : V \to \mathbb{Z}$
- Beschränkungsprädikat B

Ausgabe

Minimaler Zielknoten v^* (also $c(v^*) \leq c(u)$ für alle $u \in Z$).

Der A^*-Algorithmus (2)

Algorithmus

$\mathcal{A}$: LC-BB-OPT=
$T \leftarrow$ Baum, der nur aus der Wurzel r von S besteht;
$v^* \leftarrow$ nil
$V_0 \leftarrow \{v \mid r \to v \text{ in } S\}$ % Menge der lebenden Knoten
$V_1 \leftarrow V_0$ % Menge der „neuen" Knoten

<u>**repeat**</u>
$v \leftarrow$ Knoten mit kleinstem $\hat{c}$-Wert in V_0
$V_0 \leftarrow V_0 - \{v\}$
<u>**if**</u> **not** $B(v)$
<u>**then**</u>
<u>**if**</u> $v \in Z$
<u>**then**</u> $v^* \leftarrow v$
<u>**else**</u>
$V_1 \leftarrow \{v' \mid v \to v' \text{ in } S\}$
$V_0 \leftarrow V_0 \cup V_1$
expandiere T bei v
<u>**fi**</u>
<u>**fi**</u>
<u>**until**</u> $v^* \neq$ nil

Korrektheit von LC-BB-OPT

Satz 7.2.

Ist die Voraussetzung LC erfüllt, so hält LC-BB-OPT mit dem kleinsten Zielknoten v^* als Ausgabe, es gilt also $c(u) \geq c(v^*)$ für alle $u \in Z$.

Beweis:

a) Wie LC-BB, so hält auch LC-BB-OPT mit $v^* \in Z$ als Ausgabe.

b) Es gilt die Schleifeninvariante: Jedes minimale $u \in Z$ liegt unter einem $v \in V_0$:
   - Dies gilt beim Start wegen $V_0 = \{v \mid r \to v \text{ in } S\}$.
   - Es werde $v \in V_0$ gestrichen:
     Gilt $B(v)$, so liegt unter v kein Zielknoten.
     Sonst werden alle Söhne von v in V_0 aufgenommen.
     Jedes $u \in Z$, das unter v liegt, liegt also wieder unter einem $v' \in V_0$.

c) Sei $v^* \in Z$ die Ausgabe und $u \in Z$ minimal. Es gibt nach b) beim Stopp ein $v \in V_0$, so dass u unter v liegt. Es gilt $c(v^*) = \hat{c}(v^*) \leq \hat{c}(v) \leq c(u)$, also $c(v^*) \leq c(u)$. Also ist $v^* \in Z$ minimal.

Beispiel RSP

Beachte

RSP ist ein Maximierungsproblem.

Problem

- Gegeben: $n, p = (p_1, \ldots, p_n), w = (w_1, \ldots, w_n), K$.
- Knoten: $(x_1, \ldots, x_k),\ 1 \le k \le n,\ x_i \in \{0, 1\}$.
- Zielknoten: $x = (x_1, \ldots, x_n)$ mit $\neg B(x)$.
- Beschränkung: $B(x_1, \ldots, x_k)$ gdw $\sum_{i=1}^{k} w_i x_i > K$.
- Bewertung der Zielknoten: $c(x) = \sum_{i=1}^{n} p_i x_i$.
- Setze $x \le x'$ gdw $c(x) \le c(x')$.
- Gesucht: größter Zielknoten.

Beispiel RSP (Forts.)

Ansatz

Setze für $x = (x_1, \ldots, x_k) \in V - Z$

$$\begin{aligned}
\hat{c}(x) &= h(x) + g(x) \\
h(x) &= \sum_{i=1}^{k} p_i x_i && \text{„Profit bis } x\text{“} \\
g(x) &= \left\lfloor \max \left\{ \sum_{i=k+1}^{n} p_i y_i \mid 0 \le y_i \le 1, \right.\right. \\
&\qquad \left.\left. \sum_{i=k+1}^{n} w_i y_i \le K - \sum_{i=1}^{k} w_i x_i \right\} \right\rfloor && \text{„max. Profit unter } x\text{“} \\
\hat{c}(x) &= c(x) && \text{für } x \in Z
\end{aligned}$$

Es gilt

$$\begin{array}{ll}
\hat{c}(x) \ge c(u) & x \in V, u \in Z \text{ unter } x \\
\hat{c}(u) = c(u) & u \in Z
\end{array}$$

Also

LC-BB-OPT liefert einen Zielknoten mit maximalem Profit. Dieser liefert auch die optimale Rucksackpackung.

7.4 Auftragsplanung bei unterschiedlichen Bearbeitungszeiten

Problem

Gegeben: n Aufträge jeweils mit Bearbeitungszeit t_i, Termin d_i und Profit p_i.

Gesucht: $J \subseteq \{1, \ldots, n\}$, so dass alle Aufträge $i \in J$ rechtzeitig fertiggestellt werden können und dabei der Profit $p(J) = \sum_{i \in J} p_i$ maximiert wird.

Beachte

Sind alle t_i gleich, etwa $t_i = 1$, so ist das Problem leicht: Es kann in Zeit $O(n^2)$ gelöst werden (siehe Abschnitt 4.5). Das obige Problem ist schwer: Das Entscheidungsproblem dazu ist NP-vollständig.

Modellierungen des Suchraums: 2 Ansätze

a) Man entscheide sich im Schritt i, welches Element als nächstes in J aufgenommen wird.

b) Man entscheide sich im Schritt i, ob das Element i in J aufgenommen werden soll.

Bei a) hat jeder innere Knoten der Tiefe k maximal $n - k$ Söhne.

Bei b) hat jeder innere Knoten maximal 2 Söhne.

Anwendung des A^*-Algorithmus

Wir verwenden die Modellierung b) des Suchraums.

Also

- Knoten: $x = (x_1, \ldots, x_k)$ mit $0 \leq k \leq n$, $x_i \in \{0, 1\}, \neg B(x)$.
- Wurzel: $x = ()$, also $k = 0$.
- Zielknoten: $x = (x_1, \ldots, x_n)$ mit $\neg B(x)$.
- Bezeichnung: Für $x = (x_1, \ldots, x_k)$ sei $J(x) = \{i \mid x_i = 1,\ 1 \leq i \leq k\}$.
- Bewertung der Zielknoten $x \in Z$: $c(x) = \sum_{i \in J(x)} p_i$.
- Beschränkung: $B(x)$ gdw nicht alle Aufträge $i \in J(x)$ können rechtzeitig erledigt werden.

Anwendung des A^*-Algorithmus (Forts.)

Suchheuristik

Dies ist ein Maximierungsproblem.
Gesucht ist also ein $\hat{c} : V \to \mathbb{Z}$ mit

$$\begin{array}{ll} \hat{c}(x) \geq c(u) & x \in V, u \in Z \text{ unter } x \\ \hat{c}(u) = c(u) & u \in Z \end{array}$$

Setze z. B. für $x = (x_1, \ldots, x_k)$

$$\begin{aligned} \hat{c}(x) &= h(x) + g(x) \\ g(x) &= \sum_{i=k+1}^{n} p_i \\ h(x) &= \sum_{i \in J(x)} p_i \end{aligned}$$

Bemerkungen

Verbesserung der Rangfunktion

Es ist i. Allg. sinnvoll, die Funktion $\hat{c}$ (zur Steuerung der Suchheuristik) möglicht scharf zu wählen, hier also möglichst klein, so dass noch $\hat{c}(x) \geq c(u)$ gilt für alle $x \in V$ und $u \in Z$ unter x.

Hier z. B.: Setze $d = \max\{d_i \mid 1 \leq i \leq n\}$.
Ist $x = (x_1, \ldots, x_k)$ und $u \in Z$ unter x mit $\neg B(x)$, so gilt

$$\sum_{i \in J(x)} t_i + \sum_{i=k+1}^{n} t_i u_i \leq d$$

Dabei ist die 2. Summe der Profit unter x. Er lässt sich abschätzen durch

$$g(x) = \max \left\{ \sum_{i=k+1}^{n} p_i y_i \mid 0 \leq y_i \leq 1, \sum_{i=k+1}^{n} t_i y_i \leq d - \sum_{i \in J(x)} t_i \right\}$$

Die Berechnung führt auf das Bruchteil-Rucksackproblem, ist also in Zeit $O(n \cdot \log n)$ machbar.

Setze wie oben:

$$\hat{c}(x) = h(x) + g(x)$$

Bemerkungen (Forts.)

Algorithmus für $\mathbf{B}(\mathbf{x})$

Planungsstrategie: „Starte die dringendsten $i \in J(x)$ mit höchster Priorität, und zwar zum frühest möglichen Zeitpunkt.

Verfahren: Sortiere $J(x)$ aufsteigend nach den d_i.
Sei also $d_{i_1} \le d_{i_2} \le \cdots \le d_{i_q}$. Es gilt

$$\neg B(x) \text{ gdw } \sum_{j=1}^{l} t_{i_j} \le d_l \text{ für alle } l = 1, \ldots, q.$$

Beachte

Es gilt die Voraussetzung LC.

Also liefert LC-BB-OPT einen Zielknoten mit maximalem Profit und eine optimale Teilmenge von Aufträgen, die alle noch rechtzeitig fertiggestellt werden können.

7.5 Das Problem des Handlungsreisenden

Problem

Gegeben: Gewichteter Graph $G = (V, E, \gamma)$ mit $V = \{1, \ldots, n\}$ und $\gamma : E \to \mathbb{N}$.
Gesucht: Kürzeste Rundreise π.

Setze

- für die Kostenmatrix $D = (d_{ij})_{i,j \in V}$

$$d_{ij} = \begin{cases} \gamma(i,j) & (i,j) \in E \\ \infty & \text{sonst} \end{cases}$$

- für die Rundreise π
  $\pi = (\pi_1, \ldots, \pi_n)$ mit $\pi_1 = 1$ und
  $\pi_2, \ldots, \pi_n \in \{2, 3, \ldots, n\}$ paarweise verschieden.

Ziel

Algorithmus $\mathcal{A}$ mit

$T_{\mathcal{A},\max}(n) \in O(n^2 \cdot 2^n)$ wie in Abschnitt 3.4.
$T_{\mathcal{A},\text{mittel}}$ viel kleiner.

Modellierung des Suchraums

Ansatz

Starte die Rundreise bei Knoten 1. Entscheide im Schritt k, welcher Knoten als nächstes besucht werden soll.

Also

S ist gegeben durch

- Knoten: $x = (x_1, \ldots, x_k)$ mit
  $1 \le k \le n$, $x_i \in V$, $x_1 = 1$, $\neg B(x)$.
- Wurzel: $x = (1)$.
- Zielknoten: $x = (x_1, \ldots, x_n)$ mit $x_i \in V$, $x_1 = 1$, $\neg B(x)$.
- Beschränkung: Sei $x = (x_1, \ldots, x_k)$
  $B(x)$ gdw es gibt $1 \le i < j \le k$ mit $x_i = x_j$.
- Gewichtung der Zielknoten: $x = (x_1, \ldots, x_n)$
  $c(x) = d_{x_1,x_2} + \cdots + d_{x_{n-1},x_n} + d_{x_n,x_1}$.

Beachte

Dies ist ein Minimierungsproblem. Gesucht ist also $\hat{c} : V \to \mathbb{Z}$ mit

$$\begin{array}{ll} \hat{c}(x) \le c(u) & x \in V, u \in Z \text{ unter } x \\ \hat{c}(u) = c(u) & u \in Z \end{array}$$

Eine einfache Rangfunktion

Setze

$$\begin{array}{rcl} \hat{c}(x) & = & h(x) + g(x) \\ h(x) & = & \begin{cases} d_{x_1,x_2} + \cdots + d_{x_{k-1},x_k} & x \in V - Z \\ c(x) & x \in Z \end{cases} \\ g(x) & = & 0 \end{array}$$

Beachte

$\hat{c}(x)$ berücksichtigt nur die Kosten bis x. Die Kosten unter x werden mit 0 abgeschätzt.

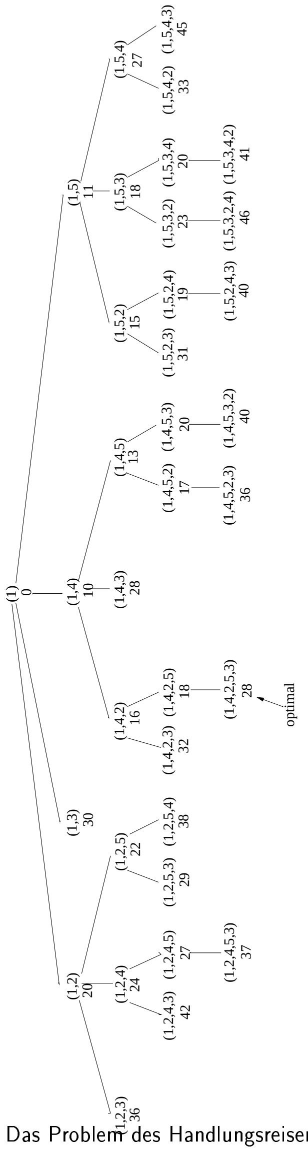
Es gilt die Voraussetzung LC.

Beispiel

Sei $n = 5$ und D gegeben durch

$$D = \begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$$

Der A* – Algorithmus liefert



Verbesserte Rangfunktion: Die Idee

Beachte

Sei $g(\pi)$ das Gewicht einer Rundreise π. Dann ist $g(\pi)$ die Summe von n Werten d_{ij}, wobei aus jeder Zeile und jeder Spalte von D ein d_{ij} auftritt.

$\Rightarrow$ Also: Vermindert man eine feste Zeile bzw. Spalte von D um r, so reduziert sich $g(\pi)$ um r.

Definition 7.2

Man kann D zu D' **reduzieren**, indem man alle Elemente einer festen Zeile bzw. Spalte um $r \in \mathbb{N}$ vermindert, so dass für die d'_{ij} noch $d'_{ij} \geq 0$ gilt.

Entsteht D' aus D durch k-malige Reduktion um $r_1, \ldots, r_k$, so lässt sich D um $r = r_1 + \cdots + r_k$ zu D' reduzieren.

Idee

Lässt sich D um r zu D' reduzieren, so gilt $g_D(\pi) = r + g_{D'}(\pi)$ für jede Rundreise π. Also gilt $g_D(\pi) \geq r$ für jedes π.

$\Rightarrow$ Benutze r zur Definition von $\hat{c}(x)$.
$\hat{c}(x) = r$ für $x = (1)$, die Wurzel von S.

Reduktion einer Matrix D am Beispiel

Sei $n = 5$ und

$$D = \begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$$

Zeilenreduktion ergibt

$$D_1 = \begin{pmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{pmatrix} \begin{array}{l} -10 \\ -2 \\ -2 \\ -3 \\ -4 \end{array}$$

$$r_1 = 21$$

Spaltenreduktion ergibt

$$D_{\text{red}} = \begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$$

$$-1 \quad -0 \quad -3 \quad -0 \quad -0 \qquad r_2 = 4$$

$$r = r_1 + r_2 = 25$$

Definition der Rangfunktion ĉ (1)

Ansatz

- Sei $x = (1)$ die Wurzel von S.
- D lasse sich um r zu D' reduzieren: Setze $\hat{c}(x) = r$.
  Sei $x = (x_1, \ldots, x_k) \to x' = (x_1, \ldots, x_k, x_{k+1})$ Kante in S, $x' \in V - Z$.
- Gesucht ist $\hat{c}(x') \leq$ Länge des Weges durch $x_1, \ldots, x_{k+1}$, dann durch die restlichen Knoten nach $x_1 = 1$. Für den Restweg sind also die Knoten $x_1, \ldots, x_{k+1}$ gesperrt. Im Vergleich zu x sind also im Graph G zusätzlich zu sperren die Kanten aus x_k heraus, in x_{k+1} hinein und $x_{k+1} \to x_1$. Die Länge des Weges wird dann wieder durch Reduktion abgeschätzt.

Realisierung

Wir definieren für jedes $x \in V - Z$ den Wert $\hat{c}(x)$ und die Matrix $D(x)$.

Gilt $x \to x'$ in S, so werden $\hat{c}(x')$ und $D(x')$ mit Hilfe von $\hat{c}(x)$ und $D(x)$ definiert.

(Dies beansprucht auch Speicherplatz!)

Definition der Rangfunktion ĉ (2)

Definition 7.3

a) Sei $x = (1)$ die Wurzel von S.
D lasse sich um r zu D' reduzieren. Setze

$$\hat{c}(x) = r \qquad D(x) = D'$$

b) Sei $x = (x_1, \ldots, x_k) \to x' = (x_1, \ldots, x_k, x_{k+1})$ Kante in S.

c) Sei $x' \in Z$. Setze $\hat{c}(x') = c(x')$.

d) Sei $x' \notin Z$. Sei $i = x_k$, $j = x_{k+1}$. Sei $A = D(x)$ und B gegeben durch

$$b_{lm} = \begin{cases} \infty & \text{falls } l = i \vee m = j \vee (l = j \wedge m = 1) \\ a_{lm} & \text{sonst} \end{cases}$$

Es lasse sich B um r reduzieren zu D'. Setze

$$\hat{c}(x') = \hat{c}(x) + a_{ij} + r \qquad D(x') = D'$$

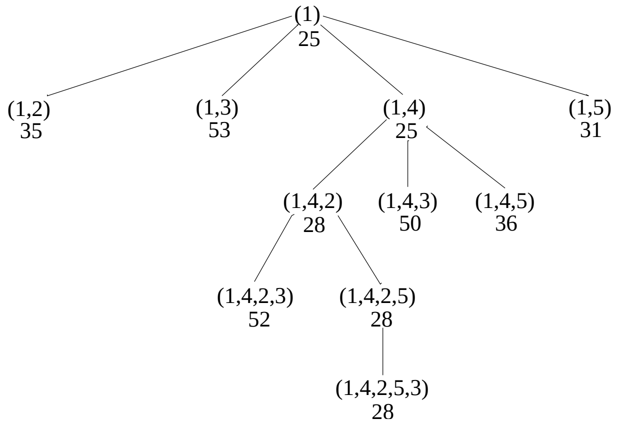
Beachte

Hier liefert die Rangfunktion $\hat{c}(x)$ eine viel bessere Abschätzung für das kleinste $c(u)$, $u \in Z$ unter x. Insbesondere gehen auch Kosten unterhalb von x mit in $\hat{c}(x)$ ein. Also: Es ist zu erwarten, dass $\hat{c}$ eine bessere Suchheuristik definiert als das frühere $\hat{c}$.

Beispiel

Sei $n = 5$ und D wie früher.

Der A^*-Algorithmus liefert mit dem verbesserten $\hat{c}$ folgenden Baum T.



Beispiel (Forts. (1))

Berechnung der $\hat{c}(x')$ aus $\hat{c}(x)$, soweit aufgebaut.

x'	x	$\hat{c}(x)$	a_{ij}	r	$\hat{c}(x')$
	(1)	25			
$(1,2)$		25	10	0	35
$(1,3)$		25	17	11	53
$(1,4)$		25	0	0	25
$(1,5)$		25	1	5	31
	$(1,4)$	25			
$(1,4,2)$		25	3	0	28
$(1,4,3)$		25	12	13	50
$(1,4,5)$		25	0	11	36
	$(1,4,2)$	28			
$(1,4,2,3)$		28	11	13	52
$(1,4,2,5)$		28	0	0	28

Beispiel (Forts. (2))

Entwicklung der Matrizen $D(x)$. Der Pfeil heißt „reduziert zu“. Fehlt der Pfeil, so ist $D(x)$ nicht reduzierbar oder r ist angegeben.

$$(1)\begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix} \quad (1,2)\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$$

$$(1,3)\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{pmatrix} \rightarrow \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{pmatrix}$$

$r = 11$ (1. Spalte)

$$(1,4)\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

Beispiel (Forts. (3))

$$(1,5) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix} \to \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix}$$

$r = 5$ (2.,3. Zeile)

$$(1,4,2) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

$$(1,4,3) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{pmatrix}$$

$r = 11+2$ (1. Spalte, 3. Zeile)

$$(1,4,5) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

$r = 11$ (2. Zeile)

Beispiel (Forts. (4))

$$(1,4,2,3) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{pmatrix}$$

$r = 13$ (3., 5. Zeile)

$$(1,4,2,5) \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

Ergebnis zur verbesserten Rangfunktion

Korrektheit

Es gilt die Voraussetzung LC. Also liefert LC-BB-OPT den kleinsten Zielknoten. Er enthält sowohl die Länge einer kürzesten Rundreise als auch diese Rundreise selbst.

Effizienz

Es werden hier i. Allg. viel weniger Knoten von S aufgebaut als bei der einfachen Rangfunktion (im Beispiel beträgt das Verhältnis etwa 1:3). Dafür ist der Zeit- und Platzbedarf zur Berechnung der $\hat{c}$-Werte erheblich höher.

Tipp: Man vergleiche die Effizienz der beiden Ansätze experimentell!

8 Und-oder-Bäume

Viele Probleme lassen sich zerlegen. Dabei gibt es 2 Arten von Zerlegung

- alternative Zerlegung
- Zerlegung in Teilprobleme

Zur Lösung des Problems reicht es von den Alternativen nur eine zu bearbeiten. Dagegen müssen die Teilprobleme alle bearbeitet werden.

Abstraktion

Beschreibung der Situation als Baum.

- Alternative Zerlegungen entsprechen **oder-Knoten**
- Zerlegungen in Teilprobleme entsprechen **und-Knoten**

Anwendungsbeispiel

a) Analyse von Spielsituationen, z. B. für Computer-Schach
   $\Rightarrow$ Spielbäume

b) Beweisplanung

8.1 Und-oder-Bäume

Definition 8.1

Ein **Und-oder-Baum** T ist ein Baum mit 2 Arten von inneren Knoten, den und-Knoten und den oder-Knoten. Die Blätter sind mit 1 oder 0 markiert.

Der Baum T ist **gelöst**, wenn die Wurzel gelöst ist. Ein und-Knoten ist gelöst, wenn alle seine Söhne gelöst sind. Ein oder-Knoten ist gelöst, wenn einer seiner Söhne gelöst ist. Ein Blatt ist gelöst, wenn es mit 1 markiert ist.

Ist T gelöst, so entsteht ein Lösungsbaum T_1 aus T dadurch, dass man in jedem gelösten oder-Knoten einen gelösten Sohn erhält und alle anderen Söhne inklusive ihrer Teilbäume löscht.

Beispiel 1

Spielbaum

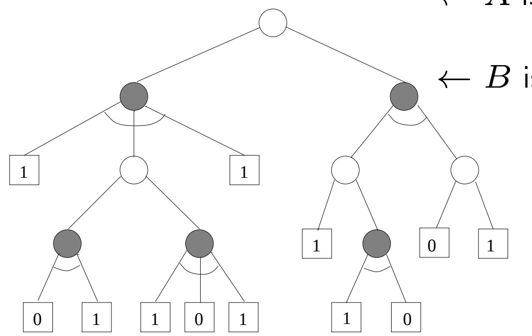
Die Spieler A und B machen abwechselnd einen Zug in einem Spiel. A beginnt. Die Wurzel von T stellt die Startkonfiguration des Spiels dar. Ist ein Spieler in der Konfiguration zum Knoten x am Zug, so stellen die Söhne von x die möglichen Folgekonfigurationen dar. Ein Blatt ist mit 1 (bzw. 0) markiert, wenn A in der entsprechenden Konfiguration gewinnt (bzw. verliert).

Frage: Kann A gewinnen, egal wie B sich verhält?

Der Baum T

$\leftarrow A$ ist am Zug

$\leftarrow B$ ist am Zug

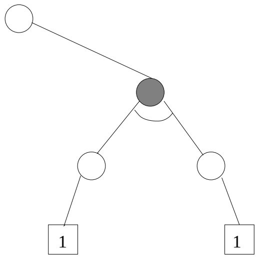


Die Knoten zu A sind oder-Knoten, die zu B sind und-Knoten.

Beispiel 1 (Forts.)

T ist gelöst. Der einzige Lösungsbaum ist

T_1



Beachte

Ein und-Knoten hat den Wert 0, falls einer seiner Söhne den Wert 0 hat.
Ein oder-Knoten hat den Wert 1, falls einer seiner Söhne den Wert 1 hat.

Beispiel 2

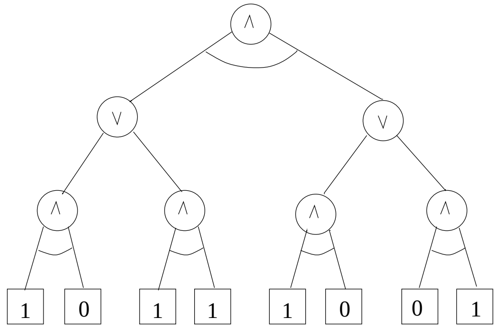
Elektrische Schaltungen

Betrachte eine hierarchische Schaltung aus **and**- und **or**-Gattern. Die Blätter sind die Eingänge der Schaltung, die Wurzel ist die Ausgabe.

and-Knoten	$\wedge$	sind und-Knoten
or-Knoten	$\vee$	sind oder-Knoten

T ist nicht gelöst

T



Lösungsalgorithmus für Und-oder-Bäume

Eingabe: Baum T
Algorithmus

$\mathcal{A}$: löse(root(T))
mit
löse(x)=

```
if x ist Blatt
  then return marke(x) fi
if x ist und-Knoten
  then
        for all y ∈ Söhne(x) do
            if löse(y) = 0
              then return 0 fi
        od
        return 1
fi
if x ist oder-Knoten
  then
        for all y ∈ Söhne(x) do
            if löse(y) = 1
              then return 1 fi
        od
        return 0
fi
```


Lösungsalgorithmus für Und-oder-Bäume (Forts.)

Lemma 8.1

löse(root(T)) liefert 1, falls T gelöst ist, und 0 sonst.

Übung

Man entwerfe einen Algorithmus, der testet, ob T gelöst ist und im positiven Fall einen Lösungsbaum liefert.

8.2 Max-min-Bäume

Motivation

Häufig kann man den Und-oder-Baum nicht ganz aufbauen, weil er zu groß oder gar unendlich ist.

Beispiel Spielbäume: Dies gilt z. B. bei Schach.

Man muss dann die Risiken und Chancen der Teilbäume abschätzen und durch eine Zahl bewerten.

Beachte den Spezialfall

Es werden nur Zahlen aus $\mathbb{B} = \{0, 1\}$ verwendet.

Dann $\max(x, y) = \textbf{or}(x, y)$
$\min(x, y) = \textbf{and}(x, y)$

Es entstehen also wieder Und-oder-Bäume.

Also

Verallgemeinerung von Und-oder-Bäumen.

- Blätter sind mit Zahlen markiert.
- Und-Knoten werden min-Knoten: Minimierung der Risiken.
- Oder-Knoten werden max-Knoten: Maximierung der Chancen.

Definitionen

Definition 8.2

Ein **Max-min-Baum** T ist ein Baum mit 2 Arten von inneren Knoten, den max- und den min-Knoten. Die Blätter sind mit Zahlen $k \in \mathbb{Z}$ markiert.

Der Wert von T ist

$$\text{wert}(T) = \text{wert}(\text{root}(T)).$$

Der Wert eines max-Knoten x ist

$$\text{wert}(x) = \max\{\text{wert}(y) \mid y \text{ Sohn von } x\}.$$

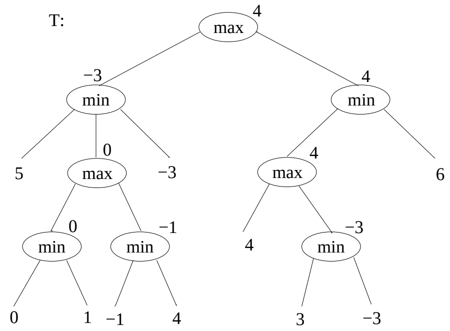
Der Wert eines min-Knotens x ist

$$\text{wert}(x) = \min\{\text{wert}(y) \mid y \text{ Sohn von } x\}.$$

Der Wert eines Blattes x ist

$$\text{wert}(x) = \text{Marke von } x.$$

Beispiel



Angegeben ist der Wert von jedem Knoten. Es gilt also $\text{wert}(T) = 4$.

Ein naiver Auswertungsalgorithmus

Eingabe: Baum T

Ausgabe: wert(T)

Algorithmus

$\mathcal{A}$: wert(T) = wert(root(T))
mit
wert(x)=

```
if x ist Blatt
  then return marke(x)
fi
if x ist max-Knoten
  then k ← −∞
       for all y ∈ Söhne(x) do
           k ← max{k, wert(y)}
       od
       return k
fi
if x ist min-Knoten
  then k ← +∞
       for all y ∈ Söhne(x) do
           k ← min{k, wert(y)}
       od
       return k
fi
```


Eine verfeinerte Auswertungsstrategie

Voraussetzung an T (*)

Jeder Sohn eines min-Knotens ist entweder ein Blatt oder ein max-Knoten.

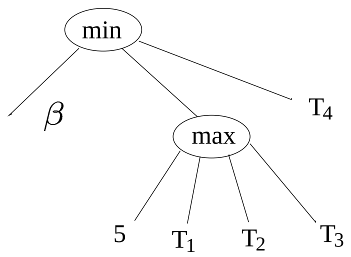
Jeder Sohn eines max-Knotens ist entweder ein Blatt oder ein min-Knoten.

Idee

Auszuwerten sei ein max-Knoten x, dessen Vater (also ein min-Knoten) einen Sohn mit Wert $\leq \beta$ hat.
Hat x einen Sohn mit Wert $> \beta$, so brauchen die weiteren Söhne von x nicht ausgewertet werden.

Eine verfeinerte Auswertungsstrategie (Forts.)

Im Bild



- Ist $\beta \leq 5$, so ist der Wert des max-Knotens $\geq 5 \geq \beta$. Er ist also für den Wert des min-Knotens ohne Bedeutung.

- Ist $\beta > 5$, aber $\beta \leq$ wert(T_1), so brauchen die Bäume T_2 bis T_3 nicht ausgewertet werden, weil wieder der Wert des max-Knotens $\geq \beta$ und damit für den min-Knoten ohne Bedeutung ist.

In beiden Fällen ist der Baum T_4 analog zu behandeln.

α-β-Auswertung (1)

Idee

Man übergebe für max-Knoten den β-Wert, also das bisherige Minimum vom Vater als Parameter.

Man übergebe für min-Knoten den α-Wert, also das bisherige Maximum vom Vater als Parameter.

Voraussetzung an T wie oben.

Algorithmen

alpha-beta-max(x, β)=

if x ist Blatt
  then
    return marke(x)
  else
    $\alpha \leftarrow -\infty$
    for all $y \in$ Söhne(x) **do**
      $k \leftarrow$ alpha-beta-min(y, α)
      if $\alpha < k$ **then** $\alpha \leftarrow k$ **fi**
      if $k \geq \beta$
        then **return** k **fi**
    od
    return α
fi

α-β-Auswertung (2)

```
alpha-beta-min(x, α)=
    if x ist Blatt
      then
          return marke(x)
      else
          β ← ∞
          for all y ∈ Söhne(x) do
              k ← alpha-beta-max(y, β)
              if β > k then β ← k fi
              if k ≤ α
                then
                    return k
              fi
          od
          return β
    fi
```


α-β-Auswertung (3)

Hauptprogramm

```
A : alpha-beta(T)=
      if root(T) ist Blatt
        then
            return marke(root(T))
      fi
      if root(T) ist max-Knoten
        then
            return alpha-beta-max(root(T), ∞)
      fi
      if root(T) ist min-Knoten
        then
            return alpha-beta-min(root(T), −∞)
      fi
```

Ergebnis

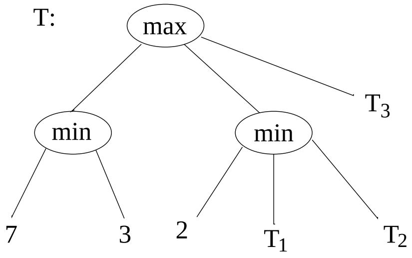
Satz 8.1.

Erfüllt T die Voraussetzung (*), so wertet alpha-beta(T) den Baum korrekt aus.

Name der Säuberungstechnik

α-β-pruning

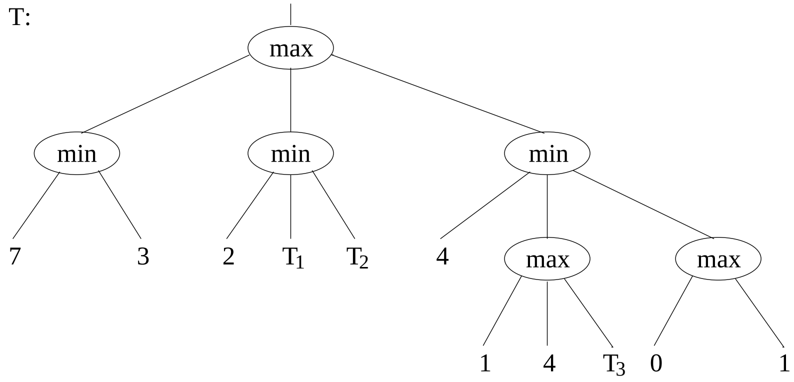
Beispiel



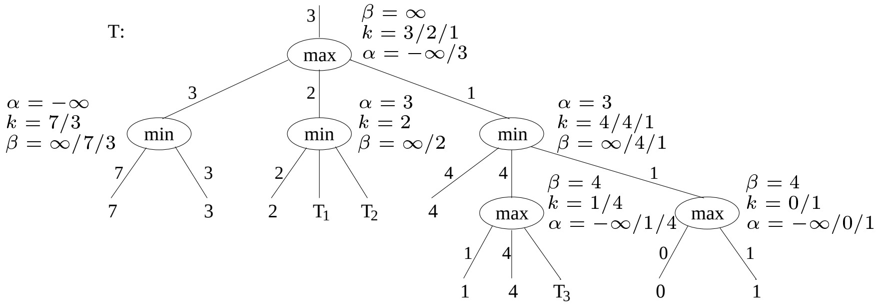
Wertet $\mathcal{A}$ die Söhne von links nach rechts aus, so werden die Bäume T_1 und T_2 nicht ausgewertet, gleichgültig wieviele Knoten sie enthalten.

Beispiel

Auszuwerten sei der Baum



Die Auswertung ergibt



Die Bäume T_1, T_2 und T_3 werden nicht ausgewertet. Ihr Wert ist (gemäß α-β-pruning) unerheblich für den Wert von T.

9 NP-Vollständigkeit

Beobachtung

Es gibt eine große Klasse $\mathcal{L}$ von Problemen mit

- Viele $P \in \mathcal{L}$ sind wirtschaftlich wichtig.
- Für jedes $P \in \mathcal{L}$ sind nur deterministische Lösungsalgorithmen mit exponentiellem Zeitbedarf bekannt.
- Jede geratene Lösung zu $P \in \mathcal{L}$ ist leicht (z. T. mit linearem Zeitbedarf) zu bestätigen.

Fragen

- Gibt es nur exponentielle Lösungsalgorithmen?
- Gibt es Gemeinsamkeiten unter den $P \in \mathcal{L}$?
- Was folgt aus der Existenz eines polynomialen Lösungsalgorithmus für ein $P \in \mathcal{L}$ für die anderen $P' \in \mathcal{L}$?
- Gibt es wenigstens polynomiale Näherungsalgorithmen für die $P \in \mathcal{L}$?

Beispiele für Entscheidbarkeitsprobleme in $\mathcal{L}$

SAT, TSP, RSP, m-Färbbarkeit, . . .

Mächtigkeit des Nicht-Determinismus

Frage

Erhöht die Einführung von nicht-deterministischen Befehlen die Berechnungskraft von Maschinenmodellen?

Beispiele

a) Endliche Automaten
Die deterministischen und die nicht-deterministischen Endlichen Automaten sind gleich mächtig: Sie erkennen genau die Typ-3-Sprachen.

b) Keller-Automaten
Die nicht-deterministischen Kellerautomaten sind echt mächtiger als die deterministischen.

c) Turing-Maschinen
Beide Versionen sind gleich mächtig.

d) Turing-Maschinen mit polynomialem Zeitbedarf
Hier ist die Frage offen $\Rightarrow$ P-NP-Problem.

Frage

Hängt der Begriff „in Polynomzeit berechenbar“ vom Maschinen-Modell ab?

9.1 Nicht-deterministische Algorithmen

Zunächst

Turing-Maschinen als Berechnungsmodell.

Beschränkung auf Entscheidungsprobleme.

Definition 9.1

a) Eine nicht-deterministische Turing-Maschine (NTM) ist ein 5-Tupel $M = (Q, \Sigma, q_0, \delta, F)$ mit

- Zustandsmenge Q,
- Startzustand $q_0 \in Q$
- akzeptierende Zustände $F \subseteq Q$,
- Bandalphabet Σ,
- $\Sigma_0 = \Sigma \cup \{b\}$, b ist das Leerzeichen
- $\delta : Q \times \Sigma_0 \to 2^{Q \times \Sigma_0 \times \{l,0,r\}}$

Intention

Sei q_1 der aktuelle Zustand, a_1 der Inhalt des Arbeitsfeldes und $p = ||\delta(q_1, a_1)||$. Dann kann die NTM aus p Schritten wählen: Ein Schritt $(q_2, a', m) \in \delta(q_1, a_1)$ bedeutet: Schreibe a' ins Arbeitsfeld, gehe in Zustand q_2 und verlege das Arbeitsfeld gemäß $m \in \{l, 0, r\}$.

Nicht-deterministische Algorithmen (Forts.)

b) **Eine Konfiguration** k ist ein Wort $k \in \Sigma_0^*(Q \times \Sigma_0)\Sigma_0^*$.
Ist $k_1 = w_1(q_1, a_1)v_1$ und $k_2 = w_2(q_2, a_2)v_2$, so heißt k_2 eine **Folgekonfiguration** von k_1, falls es $(q_2, a', m) \in \delta(q_1, a_1)$ gibt, so dass einer der folgenden Fälle gilt (mit $a'' \in \Sigma_0$).

$$\begin{array}{llll} m = 0, & w_1 = w_2, & a_2 = a', & v_1 = v_2 \\ m = l, & w_1 = w_2 a'', & a_2 = a'', & v_2 = a' v_1 \\ m = l, & w_1 = e, & a_2 = b, & v_2 = a' v_1 \\ m = r, & w_2 = w_1 a', & a_2 = a'', & v_1 = a'' v_2 \\ m = r, & w_2 = w_1 a', & a_2 = b, & v_1 = e \end{array}$$

Schreibweise $k_1 \vdash k_2$
k_1 heißt **Endkonfiguration**, falls $\delta(q_1, a_1) = \varnothing$.
Gilt außerdem $q_1 \in F$, so heißt k_1 **akzeptierend**.
$(q_0, a_1)a_2 \cdots a_n$ heißt **Startkonfiguration** zur Eingabe $x = a_1 \cdots a_n \in \Sigma^+$.
(q_0, b) ist die Startkonfiguration zur Eingabe $x = e$.

c) M heißt deterministische TM (DTM), falls gilt

$$||\delta(q, a)|| \leq 1 \text{ für alle } q \in Q, a \in \Sigma_0$$

Berechnung der nicht-deterministischen TM

Definition 9.2

Sei M eine TM.

a) Eine **Rechnung** zur Eingabe $x \in \Sigma^*$ ist eine Folge $k_0, k_1, \ldots, k_m$ von Konfigurationen mit $k_i \vdash k_{i+1}$ und k_0=Startkonfiguration zu x. Ist k_m eine Endkonfiguration, so heißt die Rechnung haltend und m ist die Länge der Rechnung. Ist k_m akzeptierend, so heißt die Rechnung **akzeptierend**.

b) $L(M) = \{x \in \Sigma^* \mid$ es gibt eine akzeptierende Rechnung zur Eingabe $x\}$
heißt die von M **akzeptierte Sprache**.

Berechnung der nicht-deterministischen TM (Forts.)

Beachte

a) Sei M nicht-deterministisch.
Zur festen Eingabe $x \in \Sigma^*$ kann es unterschiedliche Rechnungen geben, darunter

- haltende, akzeptierende Rechnungen,
- haltende, nicht akzeptierende Rechnungen,
- nicht haltende Rechnungen.

$x \in \Sigma^*$ wird akzeptiert $\quad \mathrm{gdw} \quad$ es gibt eine akzeptierende Rechnung.

b) Sei M deterministisch.
Es gibt nur eine Rechnung zur Eingabe $x \in \Sigma^*$.
$x \in \Sigma^*$ wird akzeptiert gdw die Rechnung zu x ist akzeptierend.

Zeitkomplexität einer TM/ Die Klassen P und NP

Definition 9.3

Sei M eine deterministische TM.
Die **Zeitkomplexität** T_M von M ist die Funktion

$$T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

$$T_M(n) = \max_{\substack{x \in \Sigma^* \\ |x|=n}} \{m \mid m = \text{ Länge der Rechnung zur Eingabe } x\}$$

Definition 9.4

Sei M eine nicht-deterministische TM. Die Zeitkomplexität T_M von M ist die Funktion

$$T_M : \mathbb{N} \to \mathbb{N}$$

$$T_M(n) = \max_{\substack{x \in L(M) \\ |x|=n}} \{\min\{m \mid m = \text{Länge einer akzeptierenden Rechnung zur Eingabe } x \in \Sigma^*\}\}$$

Zeitkomplexität einer TM/ Die Klassen P und NP (Forts.)

Beachte

Für die Zeitkomplexität einer NTM werden nur die $x \in L(M)$ betrachtet, und für diese x geht nur die Länge der kürzesten akzeptierenden Rechnung ein.

Definition 9.5

Sei Σ ein festes Alphabet.

$P = \{L \subseteq \Sigma^* \mid$ es gibt ein Polynom p und eine deterministische TM M mit $L = L(M)$ und $T_M(n) \leq p(n)$ für alle $n\}$

$NP = \{L \subseteq \Sigma^* \mid$ es gibt ein Polynom p und eine nicht-deterministische TM M mit $L = L(M)$ und $T_M(n) \leq p(n)$ für alle $n\}$

RSP als Beispiel

Kodierung von RSP als Sprach-Problem

$$L_{\text{RSP}} \subseteq \Sigma^*, \Sigma = \{0, 1, *\}.$$

- Sei $e = (n, p, w, K, M)$ mit $n, K, M \in \mathbb{N}_+$, $p, w \in \mathbb{N}_+^n$ eine Eingabe für RSP.

- Kodiere e als Wort $x_e \in \Sigma^*$:
  Schreibe $n, p_1, \ldots, p_n, w_1, \ldots, w_n, K, M$ in Dualdarstellung hintereinander, jeweils getrennt durch $*$.

- e wird akzeptiert (d. h., es gilt RSP(e)) gdw

$$\exists z \in \{0, 1\}^n : \sum_{i=1}^{n} w_i z_i \leq K \wedge \sum_{i=1}^{n} p_i z_i \geq M \qquad (+)$$

- Ordne also RSP folgende Sprache zu

$$L_{\text{RSP}} = \{x_e \mid e = (n, p, w, K, M), \text{ es gilt } (+)\}$$

RSP als Beispiel (Forts.)

Es gilt $L_{\text{RSP}} \in NP$

Beweis: Sei $x \in \Sigma^*$ gegeben

a) Teste, ob x die Form x_e hat, $e = (n, p, w, K, M)$.

b) Schreibe nicht-deterministisch $z = *z_1 * z_2 * \cdots * z_n *$, $z_i \in \{0, 1\}$.

c) Teste, ob $(+)$ gilt.

d) Akzeptiere x, falls a) und c) gilt.

Dies ist auf einer NTM in Polynomzeit machbar.

Bemerkungen

Beachte

1. Es gilt $P \subseteq NP$. Offen ist, ob $P = NP$ gilt.
2. Die Wahl des Alphabets Σ in der Definition von P und NP ist unwichtig, solange $\|\Sigma\| \geq 2$ gilt:
   Sprachen $L \subseteq \Gamma^*$ mit $\|\Gamma\| > 2$ sind leicht (d. h., auf linear beschränktem Platz, in linearer Zeit) auf Sprachen $L' \subseteq \Sigma^*$, $\Sigma = \{a, b\}$ zu transformieren.

Fragen

1. Hängt die Definition von P und NP vom Maschinenmodell (hier der Turing-Maschine) ab?
2. Kann man jede NTM durch eine TM simulieren? Mit welchem Zeitverlust?

Bemerkung

- „berechenbar“ ist fest definiert,
- „praktisch berechenbar“ heißt i. Allg. „in Polynomzeit berechenbar“.

Daher ist „$P = NP$?“ eine wichtige Frage.

9.2 Simulation eines Rechenmodells auf einem anderen Modell

Voraussetzung ab jetzt: $\|\Sigma\| \geq 2$

Ziel

Simulation einer NTM durch eine TM.

Ansatz

1. Wir lassen zu, dass die TM Hilfssymbole auf das Band schreiben kann: Sei $\Gamma \supseteq \Sigma$ das „Bandalphabet“. Durch Kodierung lassen sich die $a \in \Gamma - \Sigma$ entfernen.
   Beispiel: $\Sigma = \{0, 1\}$ $\Gamma = \Sigma \cup \{a, b, c, d\}$

Kodiere	0	durch	$0\,0\,0$
	1	durch	$1\,1\,1$
	a	durch	$0\,0\,1$
	b	durch	$0\,1\,0$
	c	durch	$1\,0\,0$
	d	durch	$1\,0\,1$

2. Wir wählen häufig $\Gamma = \Sigma \cup \Sigma^k$, $k \in \mathbb{N}$ fest. Dadurch wird das Band in k „Spuren“ aufgeteilt. Wir rechnen häufig auf den Spuren getrennt.

Ein Beispiel zur Simulation NTM-DTM

Problemstellung

Eingabe: $M = (Q, \Sigma, q_0, \delta, F)$ eine NTM mit $\|\delta(q, a)\| \leq k$ für alle $q \in Q$, $a \in \Sigma_0$.

Ziel: Eine DTM M', die alle Rechnungen in M der Länge $\leq l$ zur Eingabe $x \in \Sigma^*$ durchführt.

Verfahren

1. Numeriere die Elemente von $\delta(q, a)$ von 0 bis $r - 1$, $r \leq k$, fest durch (für alle q, a).
   Ist (q, a) aktuell, so sind also r Rechenschritte möglich.
2. Zu $0 \leq z < k^l$ sei $z_{l-1} \cdots z_1 z_0$ die k-näre Darstellung von z, also $z = \sum_{i=0}^{l-1} z_i k^i$, $0 \leq z_i < k$.
   Dann beschreibt z folgende Rechnung der Länge l: Führe im i-ten Schritt den z_i-ten Befehl von $\delta(q, a)$ aus, falls (q, a) aktuell ist. Hat $\delta(q, a)$ weniger als z_i Schritte, so stoppe.
3. Das Band habe 3 Spuren: Auf Spur 1 steht $x \in \Sigma^*$. Auf Spur 2 wird gezählt, $z = 0$ bis $z = k^l - 1$. Auf Spur 3 wird zum z auf Spur 2 die Rechnung ausgeführt. Es gibt zur festen Eingabe $x \in \Sigma^*$ $O(k^l)$ Rechnungen in M der Länge $\leq l$.

Simulation einer NTM durch eine DTM

Frage

Kann man jede NTM durch eine DTM simulieren? Mit welchem Zeitverlust?

Definition 9.6

$T : \mathbb{N} \to \mathbb{N}$ heißt eine **Schrittfunktion**, wenn es eine DTM gibt, die für jede Eingabe der Länge n nach genau $T(n)$ Schritten hält.

Beispiele für Schrittfunktionen

$$T_1(n) = n, \quad T_2(n) = n^2, \quad T_3(n) = 2^n.$$

Satz 9.1.

Sei $L \subseteq \Sigma^*$, $T : \mathbb{N} \to \mathbb{N}$ eine Schrittfunktion und M_1 eine NTM mit $L(M_1) = L$ und Zeitbedarf $T_{M_1}(n) \in O(T(n))$.
Dann gibt es eine DTM M_2 mit $L(M_2) = L$ und $T_{M_2}(n) \in O(c^{T(n)})$ für ein $c \in \mathbb{N}$.

Folgerung 9.1

Jede NTM mit polynomialem Zeitbedarf kann von einer DTM mit exponentiellem Zeitverlust simuliert werden.

Beweis-Skizze zu Satz 9.1

Sei $T_{M_1}(n) \leq m \cdot T(n) \quad m \in \mathbb{N}$ fest.
$M_1 = (Q, \Sigma, q_0, \delta, F)$ und $||\delta(q,a)|| \leq k$ für alle $q \in Q$, $a \in \Sigma_0$.

Es liegt $x \in \Sigma^*$ genau dann in L, wenn es eine akzeptierende Rechnung der Länge $l(x) \leq m \cdot T(|x|)$ zur Eingabe x gibt.

Deterministische Simulation aller dieser Rechnungen durch M_2:
Eingabe $x \in \Sigma^*$.

a) M_2 berechnet $l = m \cdot T(|x|)$.
(Dies geht, da T Schrittfunktion ist.)

b) M_2 führt alle Rechnungen von M_1 der Länge $\leq l$ zur Eingabe x aus (siehe oben).

c) M_2 akzeptiert x gdw eine der M_1-Rechnungen akzeptierend ist.

Offene Frage

Gibt es auch eine deterministische Simulation mit polynomialem Zeitverlust?

Simulation einer RAM durch eine DTM

Frage

Die RAM gilt als schnelles, die DTM als langsames Rechenmodell. Mit welchem Zeitverlust lässt sich die RAM auf der DTM simulieren?

Satz 9.2.

Jedes RAM-Programm $\mathcal{P}$ mit logarithmischer Zeitkomplexität $T_{\mathcal{P}}(n)$ lässt sich auf einer DTM M in Zeit $T_M(n) \in O(T_{\mathcal{P}}(n)^5)$ simulieren.

Folgerung 9.2

1. Jedes RAM-Programm lässt sich mit polynomialem Zeitverlust auf einer DTM simulieren.
2. Für jedes $L \subseteq \Sigma^*$ gilt:
   L lässt sich deterministisch in Polynomzeit auf einer RAM entscheiden gdw L lässt sich deterministisch in Polynomzeit auf einer TM entscheiden.

Also

Die Klasse P hängt nicht vom Maschinenmodell (RAM oder TM) ab.

Beweis-Skizze zu Satz 9.2

Sei $\mathcal{P}$ ein RAM-Programm. $\mathcal{P}$ benutze m Speicherplätze. Betrachte folgende DTM M mit mehreren Spuren auf dem Band:

1 Spur für den Akkumulator-Inhalt,
je 1 Spur für den Inhalt der Index-Register,
1 Spur für den Inhalt der m Speicherplätze von $\mathcal{P}$,
einige Spuren für Hilfsrechnungen.

M simuliert Befehl für Befehl das Programm $\mathcal{P}$.
Dies liefert einen Compiler RAM $\rightarrow$ DTM (vergleiche Compiler RM $\rightarrow$ TM aus dem letzten Semester).

Übung: Man zeige die Zeitschranke $T_M(n) \in O(T_{\mathcal{P}}(n)^5)$.

Literatur: Mehlhorn: Effiziente Algorithmen.

9.3 Entscheidungs- und Optimierungsprobleme

Wir betrachten folgende Klasse von Optimierungs- (hier Minimierungs-)Problemen P.

Sei	E	eine Menge von Eingaben.
	$L(e)$	die Menge der zulässigen Lösungen zu $e \in E$.
	$f(e,l)$	eine Zielfunktion zu $e \in E$ und $l \in L(e)$.
	$g \in \mathbb{N}$	ein Grenzwert.

Problemklassen: Gegeben $e \in E$.

1. **Entscheidungsproblem** $P_{\pm}$
   Gibt es ein $l \in L(e)$ mit $f(e,l) \leq g$?

2. **Zeugenproblem** P_z
   Man bestimme ein $l \in L(e)$ mit $f(e,l) \leq g$, falls es solch ein l gibt. Sonst melde man NEIN.

3. **Optimierungsproblem** P_{opt}
   Man bestimme $\min\{f(e,l) \mid l \in L(e)\}$.

4. **Funktionales Problem** P_{funk}
   Man bestimme ein optimales $l_0 \in L(e)$,
   d. h., $f(e,l_0) = \min\{f(e,l) \mid l \in L(e)\}$.

Analog für Maximierungsprobleme.

Beispiel: Rucksackproblem RSP

Eingabe

Eine Eingabe $e \in E$ hat die Form
$e = (n, p_1, \ldots, p_n, w_1, \ldots, w_n, K, M)$.
Zulässige Lösungen $L(e) : x \in \{0,1\}^n$ mit $\sum_{i=1}^{n} w_i x_i \leq K$.

Probleme

Zu $e \in E$ sei $p(x) = \sum_{i=1}^{n} p_i x_i$.

$\mathsf{RSP}_{\pm}(e)$:	Gibt es $x \in L(e)$ mit $p(x) \geq M$.
$\mathsf{RSP}_{z}(e)$:	Bestimme $x \in L(e)$ mit $p(x) \geq M$.
$\mathsf{RSP}_{\mathsf{opt}}(e)$:	Bestimme $\max\{p(x) \mid x \in L(e)\}$.
$\mathsf{RSP}_{\mathsf{funk}}(e)$:	Bestimme optimales $x_0 \in L(e)$, d. h., $p(x_0) \geq p(x)$ für alle $x \in L(e)$.

Frage

Sind diese 4 Probleme gleich schwer?

Sind sie polynomial aufeinander reduzierbar?

Vergleich Optimierungs-/Entscheidungsproblem

Definiere $\leq^{\mathsf{pol}}$, $\approx^{\mathsf{pol}}$ auf Problemen

$P \leq^{\mathsf{pol}} P'$ gdw	Gibt es polynomialen Algorithmus für P', so gibt es auch polynomialen Algorithmus für P.
$P \approx^{\mathsf{pol}} P'$ gdw	$P \leq^{\mathsf{pol}} P' \wedge P' \leq^{\mathsf{pol}} P$.

Es gilt

$\leq^{\mathsf{pol}}$ ist transitiv und reflexiv,
$\approx^{\mathsf{pol}}$ ist eine Äquivalenzrelation.

Satz 9.3.

Es gilt $\mathsf{RSP}_{\pm} \approx^{\mathsf{pol}} \mathsf{RSP}_{z} \approx^{\mathsf{pol}} \mathsf{RSP}_{\mathsf{opt}} \approx^{\mathsf{pol}} \mathsf{RSP}_{\mathsf{funk}}$.

$\Rightarrow$ Also: Ist eines dieser Probleme in Polynomzeit lösbar, so sind alle in Polynomzeit lösbar.

Bemerkung

Der Satz 9.3 gilt nicht nur für das Rucksackproblem RSP, sondern für viele Optimierungsprobleme, z. B. auch für das Problem des Handlungsreisenden TSP.

Beweis zu Satz 9.3

Wir schreiben $\leq, \approx$ statt $\leq^{\text{pol}}, \approx^{\text{pol}}$.

Es gilt $\text{RSP}_{\pm} \leq \text{RSP}_{z} \leq \text{RSP}_{\text{funk}}$.

Wir zeigen $\text{RSP}_{\text{opt}} \leq \text{RSP}_{\pm}$ und $\text{RSP}_{\text{funk}} \leq \text{RSP}_{\text{opt}}$.

Dann gilt $\text{RSP}_{\pm} \leq \text{RSP}_{z} \leq \text{RSP}_{\text{funk}} \leq \text{RSP}_{\text{opt}} \leq \text{RSP}_{\pm}$,

also $\text{RSP}_{\pm} \approx \text{RSP}_{z} \approx \text{RSP}_{\text{funk}} \approx \text{RSP}_{\text{opt}}$.

Lemma 9.1

Es gilt $\text{RSP}_{\text{opt}} \leq \text{RSP}_{\pm}$.

Beweis-Idee: Binäre Suche nach dem maximalen Profit.

Beweis zu Satz 9.3 (1. Forts.)

Konstruktion von $\mathcal{A}_{\text{opt}}$ aus $\mathcal{A}_{\pm}$:

Sei $e = (n, p, w, K)$ eine Eingabe für $\mathcal{A}_{\pm}$ bzw. $\mathcal{A}_{\text{opt}}$.

(1) $g_{\text{oben}} \leftarrow p_1 + \cdots + p_n$;
$g_{\text{unten}} \leftarrow 0$;
$g \leftarrow [g_{\text{oben}}/2]$;

(2) **while** $g_{\text{oben}} - g_{\text{unten}} \geq 2$ **do**
  if $\mathcal{A}_{\pm}(n, p, w, K, g)$
    then $g_{\text{unten}} \leftarrow g$
    else $g_{\text{oben}} \leftarrow g$
  fi
  $g \leftarrow [(g_{\text{oben}} + g_{\text{unten}})/2]$
od

(3) **if** $\mathcal{A}_{\pm}(n, p, w, K, g_{\text{oben}})$
  then $g \leftarrow g_{\text{oben}}$
  else $g \leftarrow g_{\text{unten}}$
fi

(4) output g

Analyse

$\mathcal{A}_{\text{opt}}$ berechnet den optimalen Profit.

Zeitbedarf von $\mathcal{A}_{\text{opt}}$:

$T_{\text{opt}}(|e|) \in O(\log(p_1 + \cdots + p_n) \cdot T_{\pm}(|e|)) \subseteq O(|e| \cdot T_{\pm}(|e|))$

$\Rightarrow T_{\text{opt}}$ wächst polynomial, falls $T_{\pm}$ polynomial wächst.

Beweis zu Satz 9.3 (2. Forts.)

Lemma 9.2

Es gilt $\mathsf{RSP}_{\mathsf{funk}} \leq \mathsf{RSP}_{\mathsf{opt}}$.

Beweis:

Sei $e = (n, p, w, K)$ die Eingabe.

Idee für die Konstruktion von $\mathcal{A}_{\mathsf{funk}}$ aus $\mathcal{A}_{\mathsf{opt}}$.

a) Berechne $p^* = \max\{p(x) \mid x \in L(e)\}$.

b) Berechne aus p^* ein $x \in L(e)$ mit $p^* = p(x)$:
   Für $i = 1, \ldots, n$:
   Wird $x_i = 1$ für opt. Lösung x benötigt, so setze $x_i = 1$.

Beweis zu Satz 9.3 (3. Forts.)

Algorithmus $\mathcal{A}_{\mathsf{funk}}$:

```
(1)    p* ← A_opt(e)
(2)    for i = 1 to n do
           ŵ_i ← w_i;
           x_i ← 0
       od
       a ← 0
(3)    for i = 1 to n do
           ŵ_i ← K + 1        % sei ê = (n, p, ŵ, K)
           if a + A_opt(ê) < p*
             then
                   a ← a + p_i;
                   K ← K − w_i;
                   x_i ← 1
           fi
       od
(4)    output(x_1, ..., x_n)
```

Analyse

$\mathcal{A}_{\mathsf{funk}}$ berechnet eine optimale Lösung zu e.
$\mathcal{A}_{\mathsf{funk}}$ hat polynomialen Zeitbedarf, falls dies auch für $\mathcal{A}_{\mathsf{opt}}$ gilt.

9.4 NP-vollständige Probleme

Ansatz zum P=NP-Problem:

1. Wir verschlüsseln Probleme als Sprachen über $\Sigma = \{0, 1\}$.
   Beispiel: RSP, siehe früher.
   „Vernünftige“ Kodierung:
      Zahlen dual darstellen.
      Graphen als Nachbarschaftsmatrix darstellen.

2. Wir verwenden die TM als Rechenmodell.

3. Wir suchen „schwerste“ Sprachen $L \subseteq \Sigma^*$ in NP:
   - $L \in NP$.
   - $L' \leq_{\mathsf{pol}} L$ für alle $L' \in NP$.

   $\Rightarrow$ L heißt dann NP-vollständig.

4. Wir zeigen: Ist L NP-vollständig, so gilt $P = NP \text{ gdw } L \in P$.

Beachte

Ist $f : \Sigma^* \to \Sigma^*$ in Polynomzeit berechenbar, so gibt es ein Polynom p mit

$$|f(x)| \leq p(|x|) \text{ für alle } x \in \Sigma^*$$

Definition der NP-Vollständigkeit

Definition 9.7

a) Seien $L_1, L_2 \subseteq \Sigma^*$. L_1 heißt **polynomial reduzierbar** auf L_2, falls es eine Funktion $f : \Sigma^* \to \Sigma^*$ gibt mit
   - f ist in Polynomzeit auf einer DTM berechenbar,
   - für alle $x \in \Sigma^*$ gilt $x \in L_1 \text{ gdw } f(x) \in L_2$.

   Bezeichnung $L_1 \leq_{\mathsf{pol}} L_2$.

b) $L \subseteq \Sigma^*$ heißt **NP-vollständig**, falls gilt
   - $L \in NP$.
   - Für alle $L' \in NP$ gilt $L' \leq_{\mathsf{pol}} L$.

Bemerkung

1. Vergleiche den Begriff „polynomial reduzierbar“ mit dem Begriff „reduzierbar“ aus der Theorie der Berechenbarkeit.
   $L_1 \leq_{\mathsf{pol}} L_2 \text{ gdw}$
   $L_1 \leq_{\cdot} L_2$ und die Transformation f ist in Polynomzeit berechenbar.
2. $\leq_{\mathsf{pol}}$ ist Präzisierung von $\leq^{\mathsf{pol}}$ von oben.
3. Definiere $\approx_{\mathsf{pol}}$ durch

$$L_1 \approx_{\mathsf{pol}} L_2 \;\text{ gdw }\; L_1 \leq_{\mathsf{pol}} L_2 \wedge L_2 \leq_{\mathsf{pol}} L_1$$

Erste Ergebnisse (1)

Lemma 9.3

$\leq_{\text{pol}}$ ist reflexiv und transitiv.

Beweis:

a) $\leq_{\text{pol}}$ ist reflexiv: Es gilt $L \leq_{\text{pol}} L$.

b) $\leq_{\text{pol}}$ ist transitiv: Wir schreiben $\leq$ statt $\leq_{\text{pol}}$.
Sei $L_1 \leq L_2$ und $L_2 \leq L_3$, zu zeigen ist $L_1 \leq L_3$.
Es gibt $f_1, f_2 : \Sigma^* \to \Sigma^*$ und monoton wachsende Polynome p, p_1, p_2 mit

$x \in L_1 \text{ gdw } f_1(x) \in L_2$, $f_1(x)$ in Zeit $p_1(|x|)$ berechenbar
$x \in L_2 \text{ gdw } f_2(x) \in L_3$, $f_2(x)$ in Zeit $p_2(|x|)$ berechenbar
$|f_1(x)| \leq p(|x|)$

Setze $f_3(x) = f_2(f_1(x))$ und $p_3(n) = p_1(n) + p_2(p(n))$.

Dann gilt
$x \in L_1 \text{ gdw } f_3(x) \in L_3$.
$f_3(x)$ ist in Zeit $p_3(|x|)$ berechenbar.
p_3 ist ein Polynom.

$\Rightarrow$ Also $L_1 \leq L_3$.

Erste Ergebnisse (2)

Lemma 9.4

a) L NP-vollständig, $L \leq_{\text{pol}} L'$, $L' \in NP$
$\Rightarrow L'$ ist NP-vollständig.

b) $L \in P$, $L' \leq_{\text{pol}} L \Rightarrow L' \in P$.

Beweis:

a) Übung.

b) Konstruktion einer DTM M' für L' aus M für L:

Es gibt $f : \Sigma^* \to \Sigma^*$ und Polynom p mit

$$\left.\begin{array}{l} x \in L' \text{ gdw } f(x) \in L \\ f(x) \text{ in Polynomzeit berechenbar} \\ |f(x)| \leq p(|x|) \end{array}\right\} (*)$$

Eingabe: $x \in \Sigma^*$.

1. Berechne $y = f(x)$ (mit Polynomzeit-Algorithmus für f).
2. Teste, ob $y \in L$ (mit M).
3. Akzeptiere $x \text{ gdw } y \in L$.

Es gilt: M' akzeptiert L'
M' hat polynomialen Zeitbedarf
(denn $T_{M'}(n) \leq T_f(n) + T_M(p(n))$)

$\Rightarrow$ Also $L' \in P$.

Das Hauptergebnis

Satz 9.4.

Sei L NP-vollständig. Dann gilt
$P = NP \text{ gdw } L \in P$.

Beweis:

„$\Rightarrow$“: Es gilt $P = NP$ und $L \in NP$, also $L \in P$.

„$\Leftarrow$“: Es gilt $P \subseteq NP$, zu zeigen ist $NP \subseteq P$.
Sei $L' \in NP$.
Es ist $L' \leq_{\text{pol}} L$, da L NP-vollständig. Also gilt $L' \in P$ nach Lemma 9.4 b).

Bemerkung

Nach Satz 9.4 ist jedes NP-vollständige L ein Test-Problem für die Frage, ob $P = NP$ gilt.

Also: Gibt es überhaupt NP-vollständige Probleme?
Wie findet man solche?

Beweisverfahren zu „L ist NP-vollständig“

Verfahren zum Beweis „L ist NP-vollständig“

Zeige $L \in NP$. Dann

1. Reduktion eines NP-vollständigen Problems L_0 auf L:
   Sei schon bekannt: L_0 ist NP-vollständig.
   Zeige: $L_0 \leq_{\text{pol}} L$. Dann ist auch L NP-vollständig.
2. Zeige „L ist NP-vollständig“ direkt, d. h. nach Definition:
   Zeige: $L_0 \in NP$ beliebig $\Rightarrow L_0 \leq_{\text{pol}} L$.
   Frage: Wie beschreibt man die $L_0 \in NP$ uniform?
   Antwort: Durch die NTM mit polynomialer Zeitschranke.

Strategie für den Nachweis der NP-Vollständigkeit vieler Probleme

a) Zeige nach 2., dass SAT NP-vollständig ist.
b) Wende rekursiv 1. an zum Nachweis, dass andere Probleme NP-vollständig sind.

9.5 NP-Vollständigkeit von SAT

Problem SAT (Erinnerung)

Gegeben: α Formel der Aussagenlogik (Konjunktion von Disjunktionen von Literalen).
Frage: Ist α erfüllbar?

Definition 9.8

Sei $V = \{x_1, x_2, \dots\}$ eine Menge von aussagenlogischen Variablen. Dann sind x_i und $\neg x_i$ (oder $\bar{x}_i$) **Literale**.

$c =$ $y_1 \vee y_2 \vee \cdots y_k$ ist eine **Klausel** vom Grad $\leq k$, falls die y_i Literale sind.

$\alpha =$ $c_1 \wedge \cdots \wedge c_n$ ist eine **Formel in Klausel-Normalform (KNF)**, falls die c_i Klauseln sind. Sind sie vom Grad $\leq k$, so heißt auch α vom Grad $\leq k$.

Eine **Belegung** φ ist $\varphi : V \to \{0,1\}$. Es ist $\varphi(\alpha)$ wie üblich definiert.

α heißt **erfüllbar**, falls es eine Belegung φ gibt mit $\varphi(\alpha) = 1$.

Problem SAT (bzw. SAT(k))

Gegeben: α in KNF (vom Grad $\leq k$).
Frage: Ist α erfüllbar?

Hauptergebnis

Satz 9.5.

SAT ist NP-vollständig.

Beweis:

a) SAT $\in NP$: Angabe einer NTM M mit SAT$= L(M)$.

Kodierung einer Eingabe α von SAT als $\underline{\alpha}$ über $\Sigma = \{0, 1, x, \bar{x}, \wedge, \vee\}$:

- Literal $l = x_i$, $l = \bar{x}_i \Rightarrow \underline{l} = x\underline{i}$ bzw. $\underline{l} = \bar{x}\underline{i}$ mit $\underline{i}$ = Dualzahldarstellung von $i \in \mathbb{N}$.
- Klausel $c = l_1 \vee \cdots \vee l_m \Rightarrow \underline{c} = \underline{l_1} \vee \cdots \vee \underline{l_m}$
- Formel $\alpha = c_1 \wedge \cdots \wedge c_n \Rightarrow \underline{\alpha} = \underline{c_1} \wedge \cdots \wedge \underline{c_n}$ (+)

Sei $x \in \Sigma^*$

1) M testet deterministisch, ob x die Form $\underline{\alpha}$ in (+) hat.
2) M rät nicht-deterministisch eine Belegung φ für die x_i in α.
3) M berechnet deterministisch $\varphi(\alpha)$.
4) M akzeptiert x gdw $x = \underline{\alpha}$ und $\varphi(\alpha) = 1$.

$\Rightarrow$ M arbeitet nicht-deterministisch in Polynomzeit. Also SAT $\in NP$.

Beweis zu Satz 9.5 (1. Fortsetzung)

b) $L \leq_{\text{pol}}$ SAT für alle $L \in NP$:

Zu $L \in NP$ gibt es eine NTM M und Polynom $p(n)$ mit
$x \in L$ gdw es gibt zur Eingabe x eine akzeptierende Rechnung in M der Länge $\leq p(|x|)$. (*)

Konstruktion von $\alpha : \Sigma^* \to \Sigma^*$ mit

α ist in Polynomzeit berechenbar.
$\alpha(x)$ ist Formel in KNF für alle $x \in \Sigma^*$.
$\alpha(x) \in$ SAT gdw $x \in L$.

Dann ist $L \leq_{\text{pol}}$ SAT gezeigt.

Sei $M = (Q, \Sigma, q_1, \delta, F)$ die NTM wie oben mit (o.B.d.A.)

- M benutzt Bandalphabet $\Gamma \supseteq \Sigma$.
- $F = \{q_e\}$, also nur **ein** akzeptierender Zustand.
- $b = a_1$ ist das Leerzeichen.

Wir verändern M durch $\delta(q_e, a) = \{(q_e, a, 0)\}$. M stoppt also nicht, die akzeptierenden Konfigurationen reproduzieren sich.

Sei $R = \{(q, a, q', a', \mu) \mid (q', a', \mu) \in \delta(q, a),$
$q, q' \in Q, a, a' \in \Gamma, \mu \in \{-1, 0, 1\}\}$

Die Elemente von R sind fest von 1 bis m durchnumeriert. R ist die Beschreibung von δ als Menge. Dabei bedeutet $\mu = -1(1)$ eine Verlegung des Arbeitsfeldes nach links (rechts).

Beweis zu Satz 9.5 (2. Fortsetzung)

Sei $x \in \Sigma^*$ gegeben, $n = |x|$. Aufzubauen ist $\alpha(x)$.

$\alpha(x)$ enthält folgende **Variablen**

$z_{t,k}$	für	$1 \leq t \leq p(n), 1 \leq k \leq s$ mit $s = \|Q\|$
$a_{t,i,j}$	für	$1 \leq t, \lvert i \rvert \leq p(n), 1 \leq j \leq v$ mit $v = \|\Gamma\|$
$s_{t,i}$	für	$1 \leq t, \lvert i \rvert \leq p(n)$
$b_{t,l}$	für	$1 \leq t \leq p(n), 1 \leq l \leq m$

Intention

Die Felder sind numeriert: $-p(n) \leq i \leq p(n)$

$z_{t,k} = 1$	gdw	nach t Schritten ist q_k aktueller Zustand
$a_{t,i,j} = 1$	gdw	nach t Schritten steht $a_j \in \Gamma$ im Feld i des Bandes
$s_{t,i} = 1$	gdw	nach t Schritten ist Feld i das Arbeitsfeld
$b_{t,l} = 1$	gdw	im t-ten Rechenschritt wird das l-te Element aus R benutzt

$\alpha(x)$ beschreibt alle möglichen Rechnungen von M bei Eingabe x.

Beweis zu Satz 9.5 (3. Fortsetzung)

Aufbau von $\alpha(x)$

$\alpha(x) = \alpha_A \wedge \alpha_R \wedge \alpha_K \wedge \alpha_{akz}$
mit

α_A beschreibt die Anfangsbedingung:
Startzustand q_1, Bandinhalt $b^{p(n)} x b^{p(n)-n}$.

α_R beschreibt die Randbedingungen:
Für alle t gilt: Genau ein Feld i ist das AFeld, jedes Feld i enthält genau ein $a_j \in \Gamma$, genau ein Element aus R wird für den nächsten Rechenschritt benutzt.

α_K beschreibt den Konfigurationen-Übergang gemäß R:
Zustand, AFeld und Bandinhalt ändern sich gemäß dem l-ten Element von R (das l mit $b_{t,l} = 1$).

α_{akz} beschreibt das Akzeptieren:
Zur Zeit $t = p(n)$ ist ein $q \in F$ aktuell.

Beweis zu Satz 9.5 (4. Fortsetzung)

Angabe der Komponenten von $\alpha(x)$

Zu α_A: Sei $x = a_{j_1} a_{j_2} \cdots a_{j_n}$ mit $a_{j_i} \in \Sigma$

$$\begin{aligned}\alpha_A = \; & z_{1,1} \wedge s_{1,1} \wedge a_{1,1,j_1} \wedge \cdots \wedge \cdots \wedge a_{1,n,j_n} \\ & \wedge a_{1,n+1,1} \wedge \cdots \wedge a_{1,p(n),1} \wedge a_{1,0,1} \wedge a_{1,-1,1} \wedge \\ & \cdots \wedge a_{1,-p(n),1}\end{aligned}$$

Zu α_R:

$$\begin{aligned}\alpha_R = \bigwedge_{1 \le t \le p(n)} (\; & \text{GEine}\,(z_{t,1}, \ldots, z_{t,s}) \\ & \wedge \text{GEine}\,(s_{t,-p(n)}, \ldots, s_{t,0}, \ldots, s_{t,p(n)}) \\ & \wedge \bigwedge_{-p(n) \le i \le p(n)} \text{GEine}\,(a_{t,i,1}, \ldots, a_{t,i,v}) \\ & \wedge \text{GEine}\,(b_{t,1}, \ldots, b_{t,m}) \;)\end{aligned}$$

Dabei steht GEine $(x_1, \ldots, x_N)$ für „genau ein x_i“:

$$\text{GEine}\,(x_1, \ldots, x_N) = (x_1 \vee x_2 \vee \cdots \vee x_n) \wedge \bigwedge_{1 \le i \le N} \; \bigwedge_{i+1 \le j \le N} (\bar{x}_i \vee \bar{x}_j)$$

Beweis zu Satz 9.5 (5. Fortsetzung)

Zu α_{akz}**:** Sei $q_s = q_e$ der akzeptierende Zustand

$$\alpha_{akz} = z_{p(n),s}$$

Zu α_K**:** Wir setzen

$\alpha_K = U(1) \wedge U(2) \wedge \cdots \wedge U(p(n))$

Dabei beschreibt $U(t)$ den Konfigurationen-Übergang zur Zeit $t \to t+1$.

Sei zur Zeit t

- Feld i_0 das AFeld, also $s_{t,i_0} = 1$,
- a_{j_i} der Inhalt von Feld i, also $a_{t,i,j_i} = 1$, $\hat{j} = j_{i_0}$,
- q_k der aktuelle Zustand, also $z_{t,k} = 1$,
- $(q_{k'}, a_{j'}, \mu) \in \delta(q_k, a_{\hat{j}})$ der ausgeführte Befehl von M.

Sei $(q_k, a_{\hat{j}}, q_{k'}, a_{j'}, \mu)$ das l-te Element von R.

Dann ist zur Zeit $t+1$

- Feld $i_1 = i_0 + \mu$ das AFeld,
- a_{j_i} der Inhalt von Feld i für $i \neq i_0$,
- $a_{j'}$ der Inhalt von Feld i_0,
- $q_{k'}$ der aktuelle Zustand.

Beweis zu Satz 9.5 (6. Fortsetzung)

Dies wird ausgedrückt durch

$$\begin{aligned} U(t) = \bigwedge_{-p(n)\le i\le p(n)} \{ & \bigwedge_{1\le j\le v} ((\bar{s}_{t,i} \wedge a_{t,i,j}) \to a_{t+1,i,j}) \wedge \\ & \bigwedge_{1\le l\le m} [\, ((s_{t,i} \wedge b_{t,l}) \to z_{t,k}) \wedge \\ & \quad ((s_{t,i} \wedge b_{t,l}) \to a_{t,i,j_i}) \wedge \\ & \quad ((s_{t,i} \wedge b_{t,l}) \to z_{t+1,k'}) \wedge \\ & \quad ((s_{t,i} \wedge b_{t,l}) \to a_{t+1,i,j'}) \wedge \\ & \quad ((s_{t,i} \wedge b_{t,l}) \to s_{t+1,i+\mu}) \,] \} \end{aligned}$$

Dabei steht

$$x \wedge y \to z \text{ für } \bar{x} \vee \bar{y} \vee z$$

Ergebnis (Siehe auch Mehlhorn: Effiziente Algorithmen)

$\alpha(x)$ ist in KNF.

$\alpha(x)$ ist in Polynomzeit aus x berechenbar.

$x \in L$ gdw es gibt akzeptierende Rechnung der Länge $\le p(|x|)$
gdw $\alpha(x)$ erfüllbar.
gdw $\alpha(x) \in$ SAT

Komplexität von SAT(3)

Satz 9.6.

SAT(3) ist NP-vollständig.

Beweis: Wir zeigen SAT $\leq_{\text{pol}}$ SAT(3).

a) Sei $c = y_1 \vee \cdots \vee y_n$ eine Klausel über den $x_1, x_2, \ldots$

$z_1, \ldots, z_{n-1}$ neue Variablen.

Setze $\hat{c} = (y_1 \vee \neg z_1) \wedge (z_1 \vee y_2 \vee \neg z_2) \wedge \cdots \wedge (z_{n-1} \vee y_n)$.

Sei φ Belegung der x_i.

Setze $\hat{\varphi}$ Belegung mit $\hat{\varphi}(x_i) = \varphi(x_i)$,

$$\hat{\varphi}(z_i) = \begin{cases} 0 & \varphi(y_1) = \cdots = \varphi(y_i) = 0 \\ 1 & \text{sonst} \end{cases}$$

Dann $\hat{c}$ in KNF vom Grad ≤ 3.
$\varphi(c) = 1 \text{ gdw } \hat{\varphi}(\hat{c}) = 1$.

Komplexität von SAT(3) (Forts.)

b) Sei $\alpha = c_1 \wedge \cdots \wedge c_n$ in KNF über den $x_1, x_2, \ldots$
Setze $f(\alpha) = \hat{c}_1 \wedge \cdots \wedge \hat{c}_n$.

Dann $f(\alpha)$ ist in KNF vom Grad ≤ 3.

α erfüllbar gdw $f(\alpha)$ erfüllbar (nach a)).

$\alpha \in$ SAT gdw $f(\alpha) \in$ SAT(3).

$f(\alpha)$ in Polynomzeit aus α berechenbar.

Also SAT $\leq_{\text{pol}}$ SAT(3).

Es gilt natürlich SAT(3) $\in NP$.

Komplexität von SAT(2)

Beachte

Nach Satz 9.6 ist SAT(k) NP-vollständig für alle $k \geq 3$.
Gilt dies auch für $k = 2$? NEIN (falls $P \neq NP$).

Satz 9.7.

Es gilt SAT(2) $\in P$.

Beweis:

Sei α in KNF mit max. 2 Literalen pro Klausel über $V = \{x_1, \ldots, x_n\}$. Sei $L = \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$.
Für $u \in L$ sei $\bar{u} = \bar{x}_i$, falls $u = x_i$, und $\bar{u} = x_i$, falls $u = \bar{x}_i$.

a) Man kann leicht aus α ein β in KNF berechnen mit
   - α ist erfüllbar gdw β ist erfüllbar.
   - β hat genau 2 Literale pro Klausel oder $\beta = 1$ oder $\beta = 0$.

b) Es habe also α genau 2 Literale pro Klausel.
   Sei $u \Rightarrow v$ gdw $\bar{u} \vee v$ ist Klausel in α.
   Wir zeigen

   α ist unerfüllbar $\mathrm{gdw}\ \exists x \in V : x \overset{+}{\Rightarrow} \bar{x} \overset{+}{\Rightarrow} x$

Beachte

- $\bar{u} \vee v$ ist logisch äquivalent zu $u \to v$ und $\to$ ist transitiv.
- $(x \to \bar{x}) \wedge (\bar{x} \to x)$ ist unerfüllbar.
- $u \vee v$ in α erzeugt $\bar{u} \Rightarrow v$ und $\bar{v} \Rightarrow u$.

Beweis zu Satz 9.7 (1. Forts.)

Es gilt

1. $\varphi(\alpha) = 1, u \overset{k}{\Rightarrow} v$ impliziert $\varphi(u \to v) = 1$
   (Induktion nach k).
2. $\varphi(\alpha) = 1, u \overset{+}{\Rightarrow} v \overset{+}{\Rightarrow} u$ impliziert $\varphi(u) = \varphi(v)$ (nach (1)).
3. Gilt $u \overset{k}{\Rightarrow} v$, so gilt auch $\bar{v} \overset{k}{\Rightarrow} \bar{u}$
   (Induktion nach k).
4. α ist erfüllbar gdw
   es gibt kein $x \in V$ mit $x \overset{+}{\Rightarrow} \bar{x}$ und $\bar{x} \overset{+}{\Rightarrow} x$ (*)

„⇒“: Sonst: Es gibt Belegung φ mit $\varphi(\alpha) = 1$ und nach (2) gilt $\varphi(x) = \varphi(\bar{x})$ ↯

„⇐“: Konstruktion von φ mit $\varphi(\alpha) = 1$:

Sei $[u] = \{v \mid u \overset{*}{\Rightarrow} v \overset{*}{\Rightarrow} u\}$ Äquivalenzklasse von u.

Definiere Graph $G_\alpha = (V_\alpha, E)$ mit

- $V_\alpha = \{[u] \mid u \in L\}$
- $([u], [v]) \in E$ gdw $u' \Rightarrow v', u' \in [u],$
  $v' \in [v], [u] \neq [v]$.

Beweis zu Satz 9.7 (2. Forts.)

G_α ist azyklisch.

Definiere die **Tiefe** von $[u]$ in G_α als die Länge des längsten gerichteten Weges zu $[u]$.

Definiere φ ausgehend von der leeren Belegung „top down" in G_α:

Solange es noch Knoten $[v]$ gibt mit $\varphi(v)$ nicht definiert:
Wähle unter diesen einen Knoten $[u]$ minimaler Tiefe in G_α.
Setze $\varphi(u') = 0$ für alle $u' \in [u]$, also $\varphi(\bar{u'}) = 1$.

Es gilt:

Aus $v \in [u]$ folgt $\bar{v} \in [\bar{u}]$	nach (3)
Es existiert kein $v \in V$ mit $\bar{v} \in [v]$	nach (*)

Daher belegt das Verfahren jede Variable eindeutig.

Weiterhin gilt:

$\varphi(u \to v) = 1$ für alle $([u],[v]) \in E$	(**)
$\varphi(u \to v) = 1$ für alle $u \overset{+}{\Rightarrow} v$	nach Def. von G_α
$\varphi(\alpha) = 1$	nach Def. von $\Rightarrow$

Beweis zu Satz 9.7 (3. Forts.)

Wir zeigen (**):

Sonst gibt es einen Knoten $[u]$ minimaler Tiefe, für den (**) nicht gilt. Dann gilt $([u],[v]) \in E$ mit $\varphi(u \to v) = 0$, also $\varphi(u) = 1$, $\varphi(v) = 0$.

Es gilt nach (3) auch $([\bar{v}],[\bar{u}]) \in E$ und auch $\varphi(\bar{v} \to \bar{u}) = 0$.

Also gilt Tiefe$([u]) \leq$ Tiefe$([\bar{v}]) <$ Tiefe$([\bar{u}])$.

Also wird bei der Definition von φ gesetzt $\varphi(u) = 0$ $\lightning$.

Es ist $\Rightarrow$ in Polynomzeit aus α berechenbar.
Dann ist nach Abschnitt 3.2 auch $\overset{+}{\Rightarrow}$ in Polynomzeit berechenbar.

Nach (4) ist also in Polynomzeit entscheidbar, ob α erfüllbar ist.
Also gilt SAT(2) $\in P$.

Beispiele zu SAT(2)

Zum Säubern von α

Sei $\alpha = x_1 \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge \beta$ mit β ist x_1, x_2-frei.

Dann gilt: α erfüllbar gdw $1 \wedge (0 \vee \bar{x}_2) \wedge \beta)$ erfüllbar
gdw $\bar{x}_2 \wedge \beta$ erfüllbar
gdw β erfüllbar.

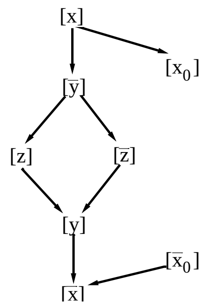
Zur Berechnung von φ aus G_α

$\alpha = (\bar{x} \vee \bar{y}) \wedge (y \vee y_0) \wedge (\bar{y}_0 \vee \bar{y}) \wedge (y \vee z) \wedge (y \vee \bar{z}) \wedge (\bar{x} \vee x_0)$

Es ergibt sich

$[y] = \{y, \bar{y}_0\}$, $[\bar{y}] = \{\bar{y}, y_0\}$, $[u] = \{u\}$ sonst

G_α :



also
$\varphi(x) = 0$
$\varphi(x_0) = 0$
$\varphi(\bar{y}) = 0$
$\varphi(z) = 0$

u	$\varphi(u)$
x	0
x_0	0
y	1
z	0

9.6 Einige NP-vollständige Probleme

Bemerkung

Bisher : nur Komplexität von Sprachen $L \subseteq \Sigma^*$ betrachtet:

- $P = \{L \subseteq \Sigma^* \mid L$ wird von einer DTM in Polynomzeit akzeptiert$\}$.
- $NP = \{L \subseteq \Sigma^* \mid L$ wird von einer NTM in Polynomzeit akzeptiert$\}$.
- $L_1 \leq_{\text{pol}} L_2$ gdw es gibt $f : \Sigma^* \to \Sigma^*$ mit
  - f in Polynomzeit auf einer DTM berechenbar.
  - $x \in L_1$ gdw $f(x) \in L_2$ für alle $x \in \Sigma^*$.
- L heißt **NP-hart**, falls $L' \leq_{\text{pol}} L$ für alle $L' \in NP$.
- L heißt **NP-vollständig**, falls $L \in NP$ und NP-hart ist.

Sinn der Betrachtung von Sprachen und der TM als Rechenmodell:

Mathematisch präzise Fundierung der Theorie.

Ziel jetzt

Übertragung der Theorie auf natürliche Probleme z. B. SAT, TSP, RSP, U-HAMILTON, . . .

NP-Theorie für natürliche Probleme

Sei

- Q_i Problem (Prädikat) auf M_i, $i = 1, 2$.
- $Q_1 \leq_{\text{pol}} Q_2$ gdw es gibt $f : M_1 \to M_2$ mit
  - f in Polynomzeit auf einer RAM berechenbar.
  - $Q_1(x)$ gdw $Q_2(f(x))$ für alle $x \in M_1$.
- $P = \{Q \mid Q$ ist auf det. RAM in Polynomzeit entscheidbar$\}$.
- $NP = \{Q \mid Q$ ist auf nicht-det. RAM in Polynomzeit entscheidbar$\}$.
- Q heißt NP-hart, falls $Q' \leq_{\text{pol}} Q$ für alle $Q' \in NP$.
- Q heißt NP-vollständig, falls $Q \in NP$ und Q NP-hart ist.

NP-Theorie für natürliche Probleme (Forts.)

Beachte

Der Begriff „in Polynomzeit berechenbar“ hängt nicht vom Maschinen-Modell ab, z. B. TM oder RAM.
„Praktisch berechenbar“ heißt „in Polynomzeit berechenbar“.

Beweis-Technik: Reduktion

Q NP-hart, $Q \leq Q' \Rightarrow Q'$ NP-hart.

Q NP-vollständig, $Q \leq Q'$, $Q' \in$ NP $\Rightarrow Q'$ NP-vollständig.

Beachte

SAT, SAT(3) sind NP-vollständig.

9.6.1 Das Problem CLIQUE

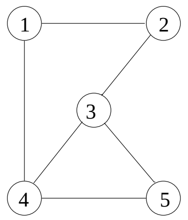
Problemstellung

Gegeben: ein ungerichteter Graph $G = (V, E)$, $k \in \mathbb{N}$.
$V' \subseteq V$ heißt eine Clique, falls $(u, v) \in E$ für alle $u, v \in V'$ mit $u \neq v$.

Frage: Gibt es eine Clique V' mit $\|V'\| = k$?

Beispiel

G:



$V_1 = \{1, 2\}$ ist eine Clique.
$V_2 = \{3, 4, 5\}$ ist eine Clique.
Es gibt keine Clique V' mit $\|V'\| \geq 4$.

Satz 9.8.

CLIQUE ist NP-vollständig.

Beweis:

a) CLIQUE $\in NP$:
Ein Algorithmus für CLIQUE:
1) Rate $V' \subseteq V$ mit $\|V'\| = k$.
2) Zeige, dass V' eine Clique ist.

b) Wir zeigen SAT(3) $\leq_{\text{pol}}$ CLIQUE.

SAT(3) $\leq_{\text{pol}}$ CLIQUE

Gesucht

f: Formeln in KNF vom Grad $\leq 3 \rightarrow$ Graphen $\times \mathbb{N}$ mit
$\alpha \in$ SAT(3) gdw $f(\alpha) = (G, k) \in$ CLIQUE.
f in Polynomzeit berechenbar.

Sei

$\alpha = c_1 \wedge \cdots \wedge c_m \qquad c_i = z_{i,1} \vee \cdots \vee z_{i,r_i} \qquad 1 \leq r_i \leq 3$

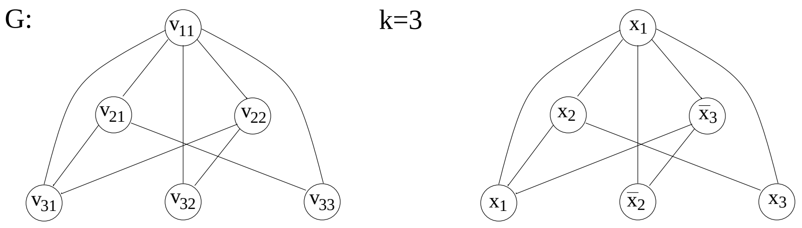
Setze

$f(\alpha) = (G, k)$ mit $G = (V, E)$, $k = m$.
$V = \{v_{i,j} \mid 1 \leq i \leq m,\ 1 \leq j \leq r_i\}$.
$E = \{(v_{i,j}, v_{p,q}) \mid i \neq p, z_{i,j} \neq \bar{z}_{p,q}\}$.

SAT(3) $\leq_{\text{pol}}$ CLIQUE (1. Forts.)

Beispiel

$\alpha = x_1 \wedge (x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$



a) $V' = \{v_{11}, v_{21}, v_{31}\}$ ist Clique der Größe $k = 3$.
$\varphi'(x_1) = \varphi'(x_2) = 1$ ist Belegung mit $\varphi'(\alpha) = 1$.

b) $\varphi''(x_1) = \varphi''(\bar{x}_3) = \varphi''(\bar{x}_2) = 1$ ist Belegung mit $\varphi''(x) = 1$.
$V'' = \{v_{11}, v_{22}, v_{32}\}$ ist Clique der Größe $k = 3$.

Also: Die erfüllenden Belegungen und die Cliquen der Größe 3 entsprechen einander.

SAT(3) $\leq_{\text{pol}}$ CLIQUE (2. Forts.)

Es gilt

a) $f(x)$ ist in Polynomzeit aus α berechenbar.
b) $\alpha \in$ SAT(3) gdw $f(\alpha) \in$ CLIQUE.

denn

$\alpha \in$ SAT(3) gdw α erfüllbar

$$\text{gdw} \bigvee_{\varphi} \bigvee_{i_1} \cdots \bigvee_{i_m} \bigwedge_{1 \leq j \leq m} \varphi(z_{j,i_j}) = 1,$$

$$\text{gdw} \bigvee_{i_1} \cdots \bigvee_{i_m} \bigwedge_{\substack{1 \leq j,l \leq m \\ j \neq l}} z_{j,i_j} \text{ und } z_{l,i_l} \text{ paarweise nicht komplementär,}$$

$$\text{gdw} \bigvee_{i_1} \cdots \bigvee_{i_m} \bigwedge_{\substack{1 \leq j,l \leq m \\ j \neq l}} v_{j,i_j} \text{ und } v_{l,i_l} \text{ paarweise verbunden in } E,$$

$$\text{gdw} \bigvee_{i_1} \cdots \bigvee_{i_m} V' = \{v_{j,i_j} \mid 1 \leq j \leq m\} \text{ ist Clique,}$$

gdw $(G, m) \in$ CLIQUE.

9.6.2 Das Problem RSP*

Problemstellung

Ist ein Rucksack ganz füllbar?

Eingabe: $a_1, a_2, \ldots, a_k, b \in \mathbb{N}_+$

Frage: Gibt es $I \subseteq \{1, \ldots, k\}$ mit $\sum_{i \in I} a_i = b$?

Satz 9.9.

RSP* ist NP-vollständig.

Beweis:

a) RSP* $\in NP$.

b) Wir zeigen SAT(3) $\leq_{\text{pol}}$ RSP*.

Gesucht: f: Formeln in KNF vom Grad $\leq 3 \to$ Folgen über $\mathbb{N}_+$ mit
$\alpha \in$ SAT(3) gdw $f(\alpha) \in$ RSP*.
f in Polynomzeit berechenbar.

Sei

α Formel über $V = \{x_1, \ldots, x_n\}$.
$\alpha = c_1 \wedge \cdots \wedge c_m$.

Also α hat m Klauseln, enthält n Variablen.

Beweis zu SAT(3) $\leq_{\text{pol}}$ RSP*

Berechne aus α Zahlen $z = z_1 \cdots z_m z_{m+1} \cdots z_{m+n}$ in Dezimaldarstellung:

$$b, v_1, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_n, d_1, \ldots, d_m, e_1, \ldots, e_m$$

b: $\quad b_j = \begin{cases} 4 & 1 \leq j \leq m \\ 1 & m < j \leq m+n \end{cases}$

v_i: $\quad v_{i_j} = \begin{cases} 1 & 1 \leq j \leq m, x_i \text{ kommt in } c_j \text{ vor} \\ 1 & j = m+i \\ 0 & \text{sonst} \end{cases}$

$\bar{v}_i$: wie v_i nur mit $\bar{x}_i$ statt x_i

d_i: $\quad d_{i_j} = \begin{cases} 1 & j = i \\ 0 & \text{sonst} \end{cases}$

e_i: $\quad e_{i_j} = \begin{cases} 2 & j = i \\ 0 & \text{sonst} \end{cases}$

Beweis zu SAT(3) $\leq_{\text{pol}}$ RSP* (Forts.)

Setze

$$f(\alpha) = (v_1, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_n, d_1, \ldots, d_m, e_1, \ldots, e_m, b)$$
$$= (a_1, \ldots, a_{2n+2m}, b) \in \mathbb{N}^{2n+2m+1}$$

Zu zeigen

$f(\alpha)$ ist in Polynomzeit aus α berechenbar.
$\alpha \in$ SAT(3) gdw $f(\alpha) \in$ RSP*.

Setze zu $I \subseteq \{1, 2, \ldots, 2(n+m)\}$

$$S(I) = \sum_{i \in I} a_i$$

Beispiel zu SAT(3) $\leq_{\text{pol}}$ RSP*

Sei $m = 3, n = 5$.
$\alpha = (x_1 \vee \bar{x}_3 \vee x_5) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_2 \vee \bar{x}_5)$.

Dies liefert (in Dezimaldarstellung)

$b =$	4 4 4	1 1 1 1 1			
$v_1 =$	1 0 0	1 0 0 0 0	$\bar{v}_1 =$	0 1 0	1 0 0 0 0
$v_2 =$	0 0 0	0 1 0 0 0	$\bar{v}_2 =$	0 0 1	0 1 0 0 0
$v_3 =$	0 0 0	0 0 1 0 0	$\bar{v}_3 =$	1 0 0	0 0 1 0 0
$v_4 =$	0 1 0	0 0 0 1 0	$\bar{v}_4 =$	0 0 0	0 0 0 1 0
$v_5 =$	1 1 0	0 0 0 0 1	$\bar{v}_5 =$	0 0 1	0 0 0 0 1
$d_1 =$	1 0 0	0 0 0 0 0	$e_1 =$	2 0 0	0 0 0 0 0
$d_2 =$	0 1 0	0 0 0 0 0	$e_2 =$	0 2 0	0 0 0 0 0
$d_3 =$	0 0 1	0 0 0 0 0	$e_3 =$	0 0 2	0 0 0 0 0

Beachte

a) Gilt $S(I) = b$, so enthält $S(I)$ zu jedem $1 \leq i \leq n$ entweder v_i oder $\bar{v}_i$.
Setze entsprechend $\varphi(x_i) = 1$ bzw. $\varphi(\bar{x}_i) = 1$.
Dann gilt $\varphi(\alpha) = 1$, also $\alpha \in$ SAT(3).

b) Sei $\varphi(x) = 1$. Wähle I so, dass $S(I)$ enthält
- v_i, falls $\varphi(x_i) = 1$, und $\bar{v}_i$, falls $\varphi(\bar{x}_i) = 1$,
- einige d_j, e_j, so dass $S(I) = b$.

Dann gilt $S(I) = b$, also $f(\alpha) \in$ RSP*.

Beispiel zu SAT(3) $\leq_{\text{pol}}$ RSP* (Forts.)

a) Sei $I = \{1, 3, 5, 7, 9, 12, 13, 14, 15, 16\}$,
d. h. es werden $v_1, v_3, v_5, \bar{v}_2, \bar{v}_4, d_2, d_3, e_1, e_2, e_3$ ausgewählt.

Dann gilt: $S(I) = b$.

I liefert $\varphi(x_1) = \varphi(x_3) = \varphi(x_5) = \varphi(\bar{x}_2) = \varphi(\bar{x}_4) = 1$
mit $\varphi(\alpha) = 1$.

b) Sei $\varphi(\bar{x}_1) = \varphi(\bar{x}_2) = \varphi(\bar{x}_3) = \varphi(x_4) = \varphi(x_5) = 1$.

Dann gilt: $\varphi(\alpha) = 1$.

φ liefert $I = \{6, 7, 8, 4, 5, 12, 13, 14, 16\}$
mit $S(I) = b$.
Es werden also $\bar{v}_1, \bar{v}_2, \bar{v}_3, v_4, v_5, d_2, d_3, e_1, e_3$ ausgewählt.

Beweis zu SAT(3) $\leq_{\text{pol}}$ RSP* (1. Forts.)

Sei

$$
\begin{aligned}
f(\alpha) &= (v_1, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_n, d_1, \ldots, d_m, e_1, \ldots, e_m, b) \\
&= (a_1, \ldots, a_{2n+2m}, b)
\end{aligned}
$$

Behauptung

$$\alpha \in \text{SAT(3)} \ \text{ gdw } \ f(\alpha) \in \text{RSP}^*.$$

„$\Leftarrow$“: Sei $I \subseteq \{1, \ldots, 2(n+m)\}$ und $b = S(I) = \sum_{i \in I} a_i$.

$S(I)$ enthält als Summanden

- zu jedem i ein v_i oder $\bar{v}_i$, aber nicht beide,
- zu jedem j ein $\hat{v}_i \in \{v_i, \bar{v}_i\}$ mit $\hat{v}_{i_j} = 1$.

Setze

$$\varphi(x_i) = \begin{cases} 1 & v_i \text{ in der Summe } S(I) \\ 0 & \bar{v}_i \text{ in der Summe } S(I) \end{cases}$$

Dann gilt $\varphi(c_j) = 1$ für alle j, also $\varphi(\alpha) = 1$.

Beweis zu SAT(3) $\leq_{\text{pol}}$ RSP* (2. Forts.)

„$\Rightarrow$“: Sei φ Belegung der x_i mit $\varphi(\alpha) = 1$.

Setze $I \subseteq \{1, \ldots, 2(n+m)\}$, so dass $S(I)$ enthält

- v_i, falls $\varphi(x_i) = 1$,
- $\bar{v}_i$, falls $\varphi(x_i) = 0$.

(jetzt sind die ersten m Ziffern in der Summe > 0
und die letzten n Ziffern $= 1$)

- d_j und/oder e_j, so dass $S(I) = b = 4 \ldots 4\, 1 \ldots 1$

Dann gilt $f(\alpha) \in$ RSP*.

9.6.3 Das Problem PART

Problemstellung Partition

Eingabe: $a_1, \ldots, a_k \in \mathbb{N}_+$

Frage: Gibt es $I \subseteq \{1, \ldots, k\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? (+)

Satz 9.10.

PART ist NP-vollständig.

Beweis:

a) PART $\in$ NP.

b) Wir zeigen RSP* $\leq_{\text{pol}}$ PART.

Gesucht: $f : \mathbb{N}_+^* \to \mathbb{N}_+^*$ mit

$x = (a_1, \ldots, a_k, b) \in$ RSP* gdw $f(x) \in$ PART (1)

f ist in Polynomzeit berechenbar. (2)

Setze

$f(x) = (a_1, \ldots, a_k, a_{k+1}, a_{k+2})$ mit

$$a_{k+1} = M - b + 1, \quad a_{k+2} = b + 1, \quad M = \sum_{i=1}^{k} a_i$$

Das Problem PART (Forts.)

Es gilt (2). Wir zeigen (1):

„$\Rightarrow$“: Ist $\sum_{i \in I'} a_i = b$, so gilt (+) mit $I = I' \cup \{k+1\}$,

denn $\sum_{i \in I} a_i = b + (M - b + 1) = M + 1$

$$= (M - b) + (b + 1) = \sum_{i \notin I} a_i$$

„$\Leftarrow$“: Gilt (+), so gilt

(1) $k+1 \in I$, $k+2 \notin I$ oder
(2) $k+1 \notin I$, $k+2 \in I$,

denn $a_{k+1} + a_{k+2} = (M - b + 1) + (b + 1) > M$.

Setze - $I' = I - \{k+1\}$ im Fall (1),
- $I' = \bar{I} - \{k+1\}$ im Fall (2).

Es ist

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i = \frac{1}{2}(M + (M - b + 1) + (b + 1)) = M + 1$$

und daher

$$\sum_{i \in I'} a_i = (M + 1) - (M - b + 1) = b.$$

Beispiel zu RSP* $\leq_{\mathsf{pol}}$ PART (Forts.)

Sei $x = (1, 2, 3, 4, 5, 7)$, also $f(x) = (1, 2, 3, 4, 5, 9, 8)$.

- Lösung $3 + 4 = 7$ zu RSP$^*(x)$ liefert
  Lösung $3 + 4 + 9 = 1 + 2 + 5 + 8$ zu PART$(f(x))$.

- Lösung $2 + 5 + 9 = 1 + 3 + 4 + 8$ zu PART$(f(x))$ liefert
  Lösung $2 + 5 = 7$ zu RSP$^*(x)$.

9.6.4 Das Problem RSP

Problemstellung Rucksackproblem

Eingabe: (n, p, w, K, M) mit $n, K, M \in \mathbb{N}_+$, $p, w \in \mathbb{N}_+^*$.

Frage: Gibt es $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ mit

$$\sum_{i=1}^{n} w_i x_i \leq K \text{ und } \sum_{i=1}^{n} p_i x_i \geq M? \qquad (+)$$

Satz 9.11.

RSP ist NP-vollständig.

Beweis:

a) RSP $\in$ NP.

b) Wir zeigen PART $\leq_{\text{pol}}$ RSP.

Gesucht: $f : \mathbb{N}_+^* \to$ Eingaben für RSP mit

$$e = (a_1, \ldots, a_k) \in \text{PART gdw } f(e) \in \text{RSP} \qquad (1)$$

f ist in Polynomzeit berechenbar. (2)

Setze

$$f(e) = (k, p, w, K, M) \text{ mit}$$

$$\begin{array}{lll} p_i = w_i = 1, & K = 1, M = 2 & \text{falls } l \text{ ungerade} \\ p_i = w_i = a_i, & K = M = l/2 & \text{falls } l \text{ gerade} \end{array}$$

$$l = \sum_{i=1}^{k} a_i$$

Beweis zu Satz 9.11 (Forts.)

Es gilt (2).

Wir zeigen (1):

- Ist l ungerade, so gilt weder $e \in$ PART noch $f(e) \in$ RSP.
- Ist l gerade, so gilt
  I ist Lösung zu e gdw I definiert Lösung x zu $f(e)$
  also gilt $e \in$ PART gdw $f(e) \in$ RSP.

Beachte die Entsprechung von

$I \subseteq \{1, \ldots, k\}$	Lösung zu PART
$x = (x_1, \ldots, x_k)$	Lösung zu RSP

gemäß

$$x_i = \begin{cases} 1 & i \in I \\ 0 & i \notin I \end{cases} \qquad \text{bzw.} \qquad I = \{i \mid x_i = 1\}$$

9.6.5 Das Problem G-HAMILTON

Problemstellung Hamilton-Kreis im gerichteten Graph

Eingabe: ein gerichteter Graph $G = (V, E)$.

Frage: Gibt es einen Hamilton-Kreis in G?

Satz 9.12.

G-HAMILTON ist NP-vollständig.

ohne Beweis.

9.6.6 Das Problem U-HAMILTON

Problemstellung Hamilton-Kreis im ungerichteten Graph

Eingabe: ein ungerichteter Graph $G = (V, E)$.

Frage: Gibt es in G einen Hamilton-Kreis?

Satz 9.13.

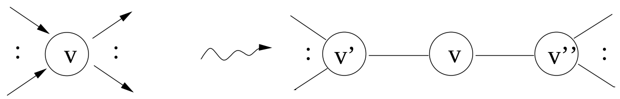
U-HAMILTON ist NP-vollständig.

Beweis:

a) U-HAMILTON $\in$ NP.
b) Wir zeigen G-HAMILTON $\leq_{\text{pol}}$ U-HAMILTON.

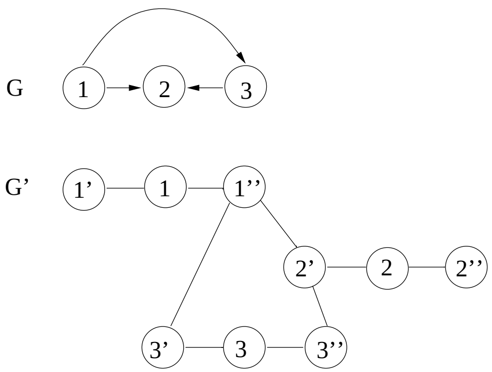
Setze f : gerichtete Graphen $\to$ ungerichtete Graphen.

$$G \mapsto f(G) = G'$$

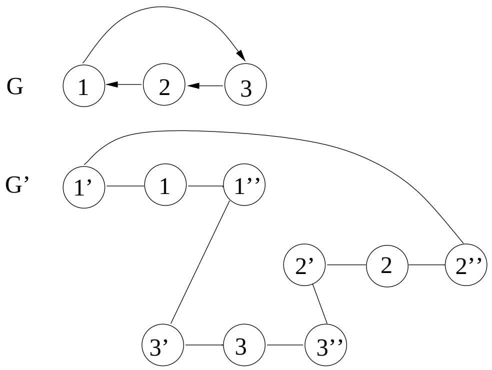


Das Problem U-HAMILTON (Forts.)

Beispiele



G und G' haben keinen H−Kreis



G und G' haben einen H−Kreis

Beweis zu Satz 9.13 (Forts.)

1) f ist in Polynomzeit berechenbar.

2) Wir zeigen

$$G \in \text{G-HAMILTON gdw } G' = f(G) \in \text{U-HAMILTON}.$$

„$\Rightarrow$“: Jeder H-Kreis in G entspricht einem H-Kreis in G':

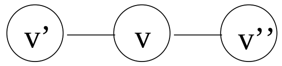
Ist $(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ ein H-Kreis in G, so ist $(v'_{i_1}, v_{i_1}, v''_{i_1}, \ldots, v'_{i_n}, v_{i_n}, v''_{i_n})$ ein H-Kreis in G'.

„$\Leftarrow$“: Sei K' ein H-Kreis in G'.
Dann besucht K' jeden v-Knoten genau ein Mal, etwa in der Reihenfolge $(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$.

Dann hat K' die Gestalt

$$(v'_{i_1}, v_{i_1}, v''_{i_1}, \ldots, v'_{i_n}, v_{i_n}, v''_{i_n}),$$

da jeder v-Knoten mit genau zwei Knoten verbunden ist.

Also: Zieht man 

zu einem Knoten v zusammen, so entsteht ein Kreis K in G (der die Richtungen der Kanten respektiert).

9.6.7 Das Problem TSP

Problemstellung Kürzeste Rundreise in Graphen

Eingabe: ein gerichteter Graph $G = (V, E)$ mit Distanzmatrix D, $K \in \mathbb{N}$.
Frage: Gibt es eine Rundreise der Länge $\leq K$?

Satz 9.14.

TSP ist NP-vollständig.

Beweis:

a) TSP $\in$ NP.

b) Wir zeigen U-HAMILTON $\leq_{\text{pol}}$ TSP.

Sei $G = (V, E)$ ein ungerichteter Graph mit $\|V\| = n$. Dann sei $G' = (V, E')$ mit $E' = \{1, \ldots, n\} \times \{1, \ldots, n\}$ und $D = (d_{ij})$ mit

$$d_{ij} = \begin{cases} 1 & (i,j) \in E \\ 2 & (i,j) \notin E \end{cases}$$

Setze f : Graphen $\rightarrow$ Graphen mit Distanzmatrix.
$f(G) = (G', D, K)$ mit $K = n$.

Dann gilt
1) $G \in$ U-HAMILTON gdw $f(G) \in$ TSP.
2) $f(G)$ ist aus G in Polynomzeit berechenbar.

9.6.8 Das Problem G-FÄRB

Problemstellung Graphenfärbung

Eingabe: ein ungerichteter Graph $G = (V, E)$, $m \in \mathbb{N}_+$.
Frage: Ist G m-färbbar?

Satz 9.15.

G-FÄRB ist NP-vollständig.

Beweis:

a) G-FÄRB $\in$ NP.

b) Wir zeigen SAT(3) $\leq_{\text{pol}}$ G-FÄRB.

Gesucht: f : Eingabe für SAT(3) $\rightarrow$ Graphen $\times \mathbb{N}_+$ mit
$\alpha \in$ SAT(3) gdw $f(\alpha) \in$ G-FÄRB.
f ist in Polynomzeit berechenbar.

Sei $\alpha = c_1 \wedge \cdots \wedge c_r$, c_j Klauseln vom Grad ≤ 3 über $V = \{x_1, \ldots, x_n\}$. Wir konstruieren aus α einen Graph G_α mit
α erfüllbar gdw G_α 3-färbbar.

Setze $f(\alpha) = (G_\alpha, 3)$. Dann gilt
$\alpha \in$ SAT(3) gdw $f(\alpha) \in$ G-FÄRB.

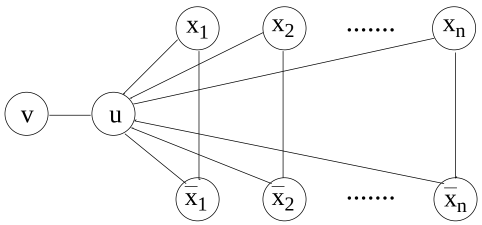
Sei $c_j = l_{1j} \vee l_{2j} \vee l_{3j}$, jede Klausel bestehe o.B.d.A. aus 3 Literalen.

Konstruktion von G_α

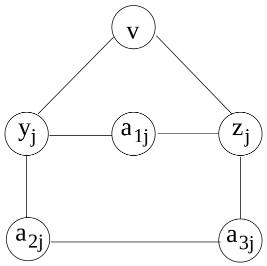
G_α besteht aus

- Skelett G_0
- Teilgraphen G_j für $j = 1, \ldots, r$

Zu G_0:



Zu G_j:



Anbindung von G_j an G_0

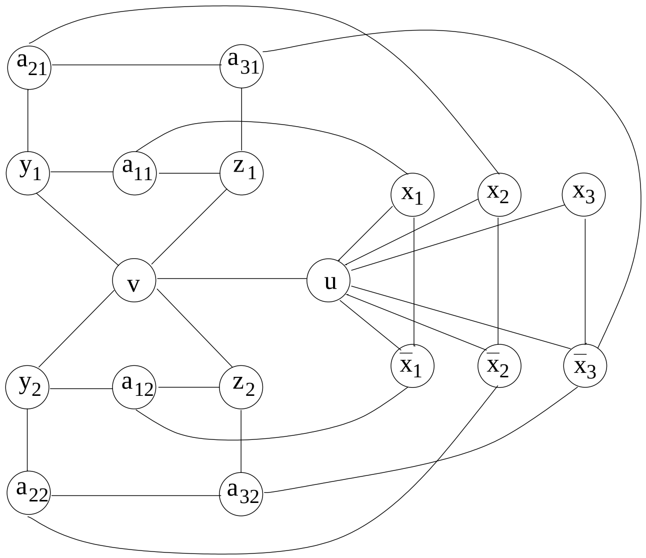
Es gibt in G_α die Kanten in G_0, den G_j und

$$(a_{1j}, l_{1j}), (a_{2j}, l_{2j}), (a_{3j}, l_{3j}) \quad j = 1, \ldots, r$$

Beispiel zu G_α

Sei $\alpha = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$.

Dies ergibt



a) Es gibt $\varphi(\alpha) = 1$ für $\varphi(x_1) = \varphi(x_2) = 1, \varphi(x_3) = 0$.
G_α ist 3-färbbar durch $\psi : V \to \{0, 1, 2\}$

$$\psi(x_i) = \varphi(x_i) \qquad \psi(\bar{x}_i) = \varphi(\bar{x}_i)$$
$$\psi(u) = 2 \qquad \psi(v) = 0$$
$$\psi(a_{ij}), \psi(y_j), \psi(z_j) \ \text{geeignet}$$

Beispiel zu G_α (Forts.)

b) Sei $\psi : V \to \{0,1,2\}$ 3-Färbung von G_α mit (o.B.d.A.) $\psi(u) = 2$, $\psi(v) = 0$.

Dann gilt $\psi(x_i), \psi(\bar{x}_i) \in \{0,1\}$, $\psi(\bar{x}_i) = 1 - \psi(x_i)$.
Setze $\varphi(x_i) = \psi(x_i)$. Dann gilt $\varphi(\alpha) = 1$.

Beweis zu „α erfüllbar $\Rightarrow \mathrm{G}_\alpha$ 3-färbbar"

Sei $\varphi(\alpha) = 1$, also $\varphi(c_j) = 1$ für alle $j = 1, \dots, r$.
Setze $\psi(x_i) = \varphi(x_i)$ $\psi(\bar{x}_i) = \varphi(\bar{x}_i)$
$\psi(u) = 2$ $\psi(v) = 0$
$\psi(a_{ij}), \psi(y_j), \psi(z_j)$ gemäß folgender Tabelle.

Sei $c_j = l_{1j} \vee l_{2j} \vee l_{3j}$, also $\varphi(l_{ij}) = 1$ für ein $1 \leq i \leq 3$.

l_{1j}	l_{2j}	l_{3j}	a_{1j}	a_{2j}	a_{3j}	y_j	z_j
0	0	1	1	1	0	2	2
0	1	0	1	0	1	2	2
0	1	1	2	0	2	1	1
1	0	0	0	1	2	2	1
1	0	1	0	1	0	2	2
1	1	0	0	0	1	2	2
1	1	1	0	0	2	2	1
φ – Werte			ψ – Werte				

Dann ist $\psi : V \to \{0,1,2\}$ eine 3-Färbung von G_α.

Beweis zu „G_α 3-färbbar $\Rightarrow \alpha$ erfüllbar“

Sei

$\psi : V \to \{0,1,2\}$ eine 3-Färbung von G_α mit (o.B.d.A.)
$\psi(u) = 2$, $\psi(v) = 0$.

Setze $\quad \varphi(x_i) = \psi(x_i)$.

Beachte $\quad \varphi(\bar{x}_i) = \psi(\bar{x}_i)$.

Wir zeigen $\varphi(c_j) = 1$ für alle j. Dann gilt $\varphi(\alpha) = 1$.

Annahme: $\varphi(c_j) = 0$, also $\varphi(l_{1j}) = \varphi(l_{2j}) = \varphi(l_{3j}) = 0$.

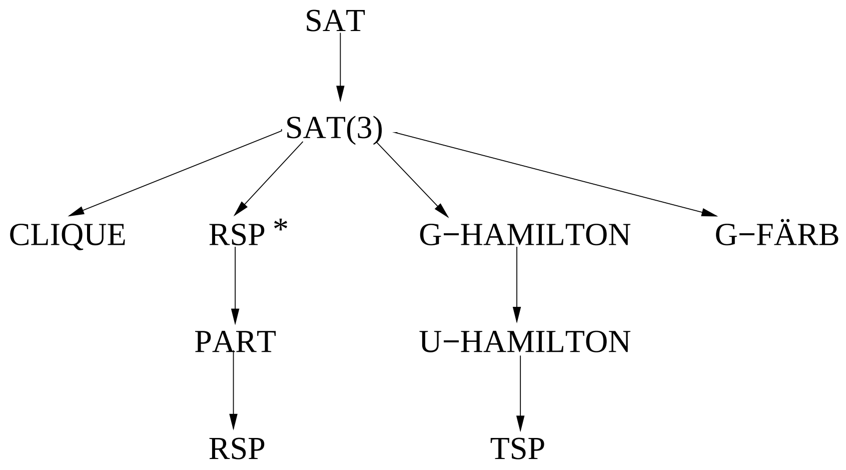
$\Rightarrow \quad \psi(a_{ij}) \in \{1,2\}$ wegen $(a_{ij}, l_{ij}) \in E$.
$\quad\quad \psi(y_j), \psi(z_j) \in \{1,2\}$ wegen $\psi(v) = 0$.
$\quad\quad \psi(a_{2j}) \neq \psi(a_{3j})$ wegen $(a_{2j}, a_{3j}) \in E$.

$\Rightarrow \quad \psi(a_{1j}) = 0$ wegen Gestalt von G_j.

$\Rightarrow \quad \lightning$

Ergebnis

Es ergibt sich jetzt folgendes Bild:



Dabei steht $\to$ für $\leq_{\text{pol}}$.

10 Polynomiale Approximation NP-harter Probleme

Motivation

Sei Q ein Optimierungsproblem mit

- E Menge der zulässigen Eingaben
- $L(e)$ Menge der zulässigen Lösungen zu $e \in E$
- $f(e, l)$ Zielfunktion, $e \in E$, $l \in L(e)$, $f(e, l) \in \mathbb{R}_+$
- opt$(e) =$ opt$\{f(e, l) \mid l \in L(e)\}$

Sei Q NP-hart.

Fragen

Kann man zu jedem $e \in E$ in Polynomzeit eine gute Näherungslösung berechnen?

Was heißt „gut“?

Wie genau kann man Q polynomial approximieren?

Ziel

Polynomialer Algorithmus $\mathcal{A}$, der den relativen Fehler beschränkt.

10.1 Approximationen und Approximationsschemata

Sei

- Q NP-hartes Optimierungsproblem,
- opt(e) das Optimum zur Eingabe $e \in E$,
- $\mathcal{A}$ polynomialer deterministischer Algorithmus, der zu jedem $e \in E$ den Wert $\mathcal{A}(e) \in \mathbb{R}$ liefert,
- $f : \mathbb{N} \to \mathbb{R}$.

Definition 10.1

$\mathcal{A}$ heißt eine **absolute Approximation** zu Q, falls es ein $c \in \mathbb{R}$ gibt, so dass für alle $e \in E$ gilt

$$|\text{opt}(e) - \mathcal{A}(e)| \leq c$$

Dann heißt Q **polynomial absolut approximierbar.**

Definition 10.2

$\mathcal{A}$ heißt eine f**-Approximation** zu Q, falls für alle $e \in E$ gilt

$$\frac{|\text{opt}(e) - \mathcal{A}(e)|}{\text{opt}(e)} \leq f(|e|)$$

Dann heißt Q polynomial f**-approximierbar**.

Ist $f(n) = c$, so heißt Q c-approximierbar.

Bemerkungen zur Approximierbarkeit

Sei $Q, \mathsf{opt}, \mathcal{A}$ wie oben.

$$RF_{\mathcal{A}}(e) = \frac{|\mathsf{opt}(e) - \mathcal{A}(e)|}{\mathsf{opt}(e)}$$ der relative Fehler von $\mathcal{A}$.

1. Q ist Maximierungsproblem

Es gilt $\mathsf{opt}(e) \geq \mathcal{A}(e)$

$$RF_{\mathcal{A}}(e) = 1 - \frac{\mathcal{A}(e)}{\mathsf{opt}(e)} \leq 1$$

$$RF_{\mathcal{A}}(e) \leq c \text{ gdw } \mathcal{A}(e) \leq (1+c)\mathsf{opt}(e)$$

Also Q ist 1-approximierbar.

Ziel Q ist c-appoximierbar für kleines c.

Bemerkungen zur Approximierbarkeit (Forts.)

2. Q ist Minimierungsproblem

Es gilt $\mathcal{A}(e) \geq \mathsf{opt}(e)$
$RF_{\mathcal{A}}(e)$ kann groß werden.

Ziel Q ist c-approximierbar.
z. B. Q 0,5-approximierbar durch $\mathcal{A}$
$\Rightarrow RF_{\mathcal{A}}(e)$ ist auf 50% beschränkt.

Frage

Lässt sich Q beliebig genau polynomial approximieren?

Approximationsschemata

Definition 10.3

Ein Algorithmus $\mathcal{A}$ mit Parameter ε heißt ein **polynomiales Approximationsschema** für Q, falls für jedes $\varepsilon > 0$ gilt

(i) $\mathcal{A}_\varepsilon$ ist ε-Approximation für Q.
(ii) $\mathcal{A}_\varepsilon$ hat polynomialen Zeitbedarf: $T_\varepsilon(n) \leq p(n)$.

$\mathcal{A}$ heißt **voll-polynomiales Approximationsschema**, falls für jedes $\varepsilon > 0$ (i) gilt und außerdem

(ii') $\mathcal{A}$ hat polynomialen Zeitbedarf in n und $\frac{1}{\varepsilon}$, d. h. $T_\mathcal{A}(n, \varepsilon) \leq p(n, \frac{1}{\varepsilon})$, p ein Polynom.

Beachte

a) Gibt es ein polynomiales Approximationsschema $\mathcal{A}$ zu Q, so ist Q ε-approximierbar für jedes $\varepsilon > 0$.
b) Gibt es ein voll-polynomiales Approximationsschema $\mathcal{A}$ zu Q, so ist die Rechenzeit $T_\mathcal{A}$ polynomial beschränkt in der Größe $|e|$ der Eingabe und $\frac{1}{\varepsilon}$, wobei ε die Genauigkeit der Approximation angibt.

Beispiele

Sei Q das Rucksackproblem. Dies ist ein Maximierungsproblem.

Wähle $\mathcal{A}$ so: Zur Eingabe $e = (n, p, w, K)$

- sortiert $\mathcal{A}$ so, dass $p_1/w_1 \geq \cdots \geq p_n/w_n$,
- füllt $\mathcal{A}$ den Rucksack in dieser Reihenfolge, wie es die Gewichtsgrenze K zulässt.

Es gilt

- $\mathcal{A}$ ist nicht absolute Approximation für Q.
- $\mathcal{A}$ ist c-Approximation für $c = 1$, nicht für $c < 1$.

  Denn: Für die Eingabe $e_i = (2, (2, i), (1, i), i) \;\; i \in \mathbb{N}$ liefert $\mathcal{A}(e_i) = 2$ und $\mathrm{opt}(e_i) = i$, also

$$\mathrm{opt}(e_i) - \mathcal{A}(e_i) = i - 2 \to \infty \text{ für } i \to \infty$$

$$\frac{\mathrm{opt}(e_i) - \mathcal{A}(e_i)}{\mathrm{opt}(e_i)} = \frac{i-2}{i} \leq 1$$

10.2 Absolut approximierbare Probleme

Es gibt wenige NP-harte Probleme, die sich polynomial absolut approximieren lassen. Eins davon ist PART-OPT.

Problemstellung PART-OPT

Gegeben: $n, l_1, \ldots, l_n,\ L \in \mathbb{N}$

Gesucht: $A, B \subseteq \{1, \ldots, n\}$ disjunkt mit $\sum\limits_{i \in A} l_i \leq L$, $\sum\limits_{i \in B} l_i \leq L$, $\|A\| + \|B\|$ maximal.

Anwendung

n Datensätze der Länge $l_1, \ldots, l_n$ sollen auf 2 Bändern der Länge L gespeichert werden. Frage: Wie viele der Datensätze kann man maximal speichern, ohne einen Datensatz zu teilen.

PART-OPT ist absolut approximierbar

Satz 10.1.

a) PART-OPT ist NP-hart.
b) Es gibt polynomialen Algorithmus $\mathcal{A}$ mit opt$(e) - \mathcal{A}(e) \leq 1$ für jede Eingabe e.

Beweis:

a) PART $\leq_{\text{pol}}$ PART-OPT:

Sei $f(l_1, \ldots, l_n) = (n, l_1, \ldots, l_n, [(\sum\limits_{i=1}^{n} l_i)/2])$. Dann gilt:

PART$(l_1, \ldots, l_n)$ gdw $\|$PART-OPT$(f(l_1, \ldots, l_n))\| \geq n$.

b) Algorithmus $\mathcal{A}$: Sei $e = (n, l_1, \ldots, l_n, L)$ eine Eingabe.

1) Sortiere die l_i so, dass $l_1 \leq l_2 \leq \cdots \leq l_n$.
2) Setze $q = \max\{j \mid \sum\limits_{i=1}^{j} l_i \leq L\}$; also $\sum\limits_{i=1}^{q+1} l_i > L$ (*)
   $A = \{1, \ldots, q\}$.
3) Setze $k = \max\{j \mid \sum\limits_{i=q+1}^{j} l_i \leq L\}$;
   $B = \{q+1, \ldots, k\}$.
4) Setze $\mathcal{A}(e) = k$.

PART-OPT ist absolut approximierbar (Forts.)

$\mathcal{A}$ ist polynomialer Algorithmus.

Zu zeigen $\quad \mathrm{opt}(e) - \mathcal{A}(e) \leq 1.$

Sei $\quad p = \max\{j \mid \sum_{i=1}^{j} l_i \leq 2L\}.$

Es gilt $\quad \sum_{i=q+1}^{p-1} l_i \leq \sum_{i=q+2}^{p} l_i = \sum_{i=1}^{p} l_i - \sum_{i=1}^{q+1} l_i \overset{(*)}{\leq} L,$

also $\quad \mathcal{A}(e) \geq p-1, \mathrm{opt}(e) \leq p,$

also $\quad \mathrm{opt}(e) - \mathcal{A}(e) \leq p - (p-1) = 1.$

Nicht absolut approximierbare Probleme

Beachte

Ist $P = NP$, so ist jedes $Q \in NP$ polynomial exakt lösbar, also auch polynomial approximierbar. Gibt es NP-vollständige Q, die nur dann polynomial absolut approximierbar sind, wenn $P = NP$ gilt?

Satz 10.2.

Ist $P \neq NP$, so ist RSP nicht absolut approximierbar.

Beweis: Sonst

Sei $\mathcal{A}$ Polynomzeitalgorithmus, $k \in \mathbb{N}$ mit
$\mathrm{opt}(e) - \mathcal{A}(e) \leq k$ für jede Eingabe $e = (n, p, w, K)$.

Betrachte folgenden Algorithmus $\mathcal{A}'(e)$:

1. Setze $p'_i = (k+1)p_i$, $e' = (n, p', w, K)$.
2. Sei $\mathrm{opt}' = \mathcal{A}(e')$.
3. Gib $\lceil \mathrm{opt}'/(k+1) \rceil$ als optimalen Profit zu e zurück.

$\mathcal{A}'$ ist ein Polynomzeitalgorithmus.

Es gilt $\quad \mathrm{opt}(e') - \mathcal{A}(e') \leq k$ und $\mathrm{opt}(e') = (k+1)\mathrm{opt}(e)$
also $\quad \mathrm{opt}(e') - k \leq \mathcal{A}(e') \leq \mathrm{opt}(e')$
also $\quad \mathrm{opt}(e) - k/(k+1) \leq \mathcal{A}(e')/(k+1) \leq \mathrm{opt}(e)$
also $\quad \mathrm{opt}(e) \leq \lceil \mathcal{A}(e')/(k+1) \rceil \leq \mathrm{opt}(e)$
also $\quad \mathcal{A}'(e) = \lceil \mathcal{A}(e')/(k+1) \rceil = \mathrm{opt}(e)$
also $\quad$ ist $\mathrm{opt}(e)$ in Polynomzeit berechenbar. $\lightning$

CLIQUE ist nicht polynomial absolut approximierbar

Satz 10.3.

Ist $P \neq NP$, so ist CLIQUE nicht absolut approximierbar.

Beweis-Idee:

Der Beweis läuft wie beim Rucksackproblem RSP.

Sei $e = G = (V, E)$ ein Graph,

also opt(e)=maximale Größe einer Clique in G.

Sei $\mathcal{A}$ Polynomzeitalgorithmus für CLIQUE, $k \in \mathbb{N}$, so dass opt$(e) - \mathcal{A}(e) \leq k$ für alle $e = G = (V, E)$.

Betrachte folgenden Algorithmus $\mathcal{A}'(e)$:

1. Sei $G^i = (V^i, E^i)$ Kopie von G, $i = 1, \ldots, k+1$:
   $V^i = \{v^i \mid v \in V\}$, $E^i = \{(u^i, v^i) \mid (u, v) \in E\}$

   Setze $e' = G' = (V', E')$ mit $V' = \bigcup_{i=1}^{k+1} V^i$ und

   $$E' = \bigcup_{i=1}^{k+1} E^i \cup \{(u^i, v^j) \mid i \neq j, u, v \in V\}$$
2. Sei opt$' = \mathcal{A}(e')$.
3. Gib $\lceil \text{opt}'/(k+1) \rceil$ als Größe einer maximalen Clique zu e zurück.

Dann ist $\mathcal{A}'$ ein Polynomzeitalgorithmus, der opt(e) berechnet. $\lightning$

10.3 c-approximierbare Probleme

Frage

Gibt es NP-vollständige Probleme, die für ein $c > 0$ polynomial c-approximierbar sind?

Problemstellung: Zeitpläne für unabhängige Prozesse.

Gegeben: n unabhängige Prozesse mit Zeitbedarf $l_1, \ldots, l_n$.
m Prozessoren bzw. Zeitschranke T.

Problem: SIT$_m$ (Scheduling Independent Tasks):
Verteile die Prozesse auf die m Prozessoren, so dass alle n Prozesse möglichst früh fertig sind.

Problem: BIN-PACKING:
Wieviele Prozessoren sind nötig, so dass alle n Prozesse zur Zeit T fertiggestellt werden können?

SIT$_m$ ist NP-vollständig

Problem SIT$_m$ = SIT$_{m,\text{opt}}$

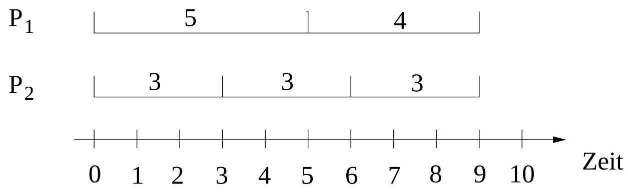
Gegeben: $n, l_1, \ldots, l_n, m$.

Gesucht: minimales T, so dass die n Prozesse auf m Prozessoren zur Zeit T fertig werden können.

Beispiel

Sei $n = 5, m = 2, l = (5, 4, 3, 3, 3)$.

Zeitplan:



Also $T = 9$ ist optimal.

Satz 10.4.

Für $m \geq 2$ ist SIT$_m$ NP-vollständig.

Beweis für $m = 2$:

a) SIT$_2 \in NP$.

b) PART $\leq_{\text{pol}}$ SIT$_2$:
Sei $f(l_1, \ldots, l_n) = (n, l_1, \ldots, l_n)$. Dann gilt:
PART$(l_1, \ldots, l_n)$ gdw SIT$_2(f(l_1, \ldots, l_n)) \leq \left\lceil \frac{\sum_{i=1}^{n} l_i}{2} \right\rceil$

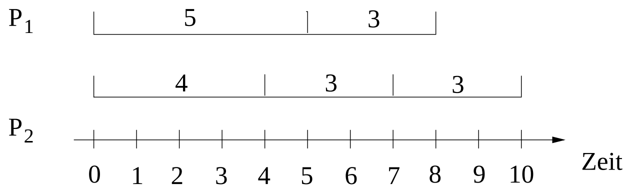
c-Approximation für SIT$_m$

Strategie LPT (longest processing time first)

1. Sortiere so, dass $l_1 \geq l_2 \geq \cdots \geq l_n$.
2. Zu jedem Zeitpunkt t starte freien Prozessor mit dem freien Prozess, der größten l-Wert hat.

Beispiel $n = 5, m = 2$, $l = (5, 4, 3, 3, 3)$

LPT liefert $T = 10$:



Satz 10.5.

LPT liefert für SIT$_m$ eine c_m-Approximation mit $c_m = \frac{1}{3} - \frac{1}{3m}$.

o. Beweis (siehe Horowitz+Sahni).

Im Beispiel $c_2 = \frac{1}{3} - \frac{1}{6} = \frac{1}{6}$

$$\frac{\text{LPT}(e) - \text{opt}(e)}{\text{opt}(e)} = \frac{10 - 9}{9} = \frac{1}{9} \leq c_2$$

BIN-PACKING ist NP-vollständig

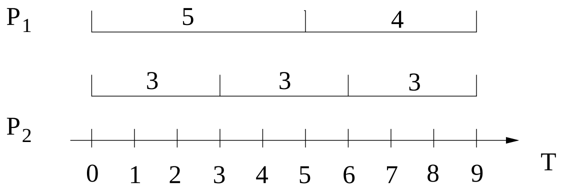
Problem BIN-PACKING

Gegeben: $n, l_1, \ldots, l_n, T$.

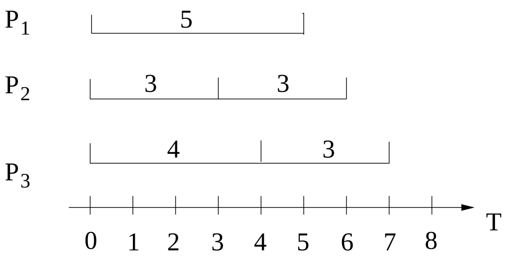
Gesucht: minimale Anzahl m der „bins“ P_i der Größe T, in die die $l_1, \ldots, l_n$ reinpassen.

Beispiel $n = 5, l = (5, 4, 3, 3, 3)$

a) $T = 9 \Rightarrow m = 2$ ist optimal:



b) $T = 8 \Rightarrow m = 3$ ist optimal:



BIN-PACKING ist NP-vollständig (Forts.)

Satz 10.6.

BIN-PACKING ist NP-vollständig.

Beweis:

a) BIN-PACKING $\in NP$.

b) PART $\leq_{\text{pol}}$ BIN-PACKING:

Sei $f(l_1, \ldots, l_n) = (n, l_1, \ldots, l_n, [(\sum_{i=1}^{n} l_i)/2])$. Dann gilt:

PART$(l_1, \ldots, l_n)$ gdw BIN-PACKING$(f(l_1, \ldots, l_n)) \leq 2$

Strategien für BIN-PACKING

Strategien

Gegeben sei $n, l_1, \ldots, l_n, T, P_1, P_2, \ldots$

First Fit (FF)

Für $j = 1, \ldots, n$: Sei i minimal mit l_j passt noch in P_i. Speichere Objekt j in P_i.

Best Fit (BF)

Für $j = 1, \ldots, n$: Bestimme i so, dass P_i am vollsten, aber l_j passt noch in P_i. Speichere Objekt j in P_i.

First Fit Decreasing (FFD)

Sortiere erst so, dass $l_1 \geq l_2 \geq \cdots \geq l_n$.
Verteile dann nach FF.

Best Fit Decreasing (BFD)

Sortiere erst so, dass $l_1 \geq l_2 \geq \cdots \geq l_n$.
Verteile dann nach BF.

Qualität der Strategien für BIN-PACKING

Satz 10.7.

Sei $\mathcal{A}$ entweder FF oder BF. Dann gilt für alle $e \in E$

$$\mathcal{A}(e) \leq \frac{17}{10}\text{opt}(e) + 2$$

Sei $\mathcal{A}$ entweder FFD oder BFD. Dann gilt für alle $e \in E$

$$\mathcal{A}(e) \leq \frac{11}{9}\text{opt}(e) + 4$$

o. Beweis (siehe SIAM J. on Comp. I (1974) 299-325).

Beachte

a) Für den relativen Fehler $\text{RF}_{\mathcal{A}} = (\mathcal{A}(e) - \text{opt}(e))/\text{opt}(e)$ gilt

$$\text{RF}_{\mathcal{A}}(e) \leq \frac{7}{10} + \frac{2}{\text{opt}(e)} \quad \text{bzw.} \quad \text{RF}_{\mathcal{A}}(e) \leq \frac{2}{9} + \frac{4}{\text{opt}(e)}$$

b) In beiden Fällen gilt für den Zeitbedarf

$$T_{\mathcal{A}}(n) \in O(n^2)$$

c-Approximierbarkeit von TSP

Satz 10.8.

Ist $P \neq NP$, so ist TSP für kein $c > 0$ polynomial c-approximierbar.

Beweis:

Annahme: $\mathcal{A}$ ist Polynomzeit-Algorithmus für TSP mit

$$\mathcal{A}(e) - \mathsf{opt}(e) \leq c \cdot \mathsf{opt}(e) \qquad (*)$$

Wir zeigen, dann ist U-HAMILTON in P, also $\lightning$

Sei $G = (V, E)$ Eingabe für U-HAMILTON,
$V = \{1, \dots, n\}$.

Setze $G' = (V, E, D)$ Eingabe für TSP mit

$$d(i,j) = \begin{cases} 1 & (i,j) \in E \\ 2 + \lceil c \cdot n \rceil & \text{sonst} \end{cases}$$

c-Approximierbarkeit von TSP (Forts.)

Es gilt: G' ist in Polynomzeit aus G berechenbar.

$$\begin{aligned} \text{U-HAMILTON}(G) &\Rightarrow \mathsf{opt}(G') = n \\ \neg\text{U-HAMILTON}(G) &\Rightarrow \mathsf{opt}(G') \geq n + 1 + \lceil c \cdot n \rceil \\ &\qquad\qquad > (1+c) \cdot n \end{aligned}$$

Also

$$\begin{aligned} \text{U-HAMILTON}(G) &\Rightarrow \mathcal{A}(G') \leq (1+c) \cdot n \text{ wegen } (*) \\ \neg\text{U-HAMILTON}(G) &\Rightarrow \mathcal{A}(G') \geq \mathsf{opt}(G') > (1+c) \cdot n \end{aligned}$$

Also U-HAMILTON(G) gdw $\mathcal{A}(G') \leq (1+c) \cdot n$

1-Approximation für ΔTSP

Beachte

Für G' aus dem letzten Beweis ist D symmetrisch, d. h. $d(i, j) = d(j, i)$. Also ist auch TSP nicht polynomial ε-approximierbar, wenn man es auf symmetrische Distanzfunktionen einschränkt.

Wir sagen: D erfüllt die **Dreiecksungleichung**, falls stets $d(i, j) \leq d(i, k) + d(k, j)$ gilt.

Problemstellung ΔTSP

Eingabe: $G = (V, E, D)$, D ist symmetrisch und erfüllt die Dreiecksungleichung

Gesucht: Länge der kürzesten Rundreise durch alle Knoten von G

Satz 10.9.

ΔTSP ist polynomial 1-approximierbar.

Beweis zu Satz 10.9

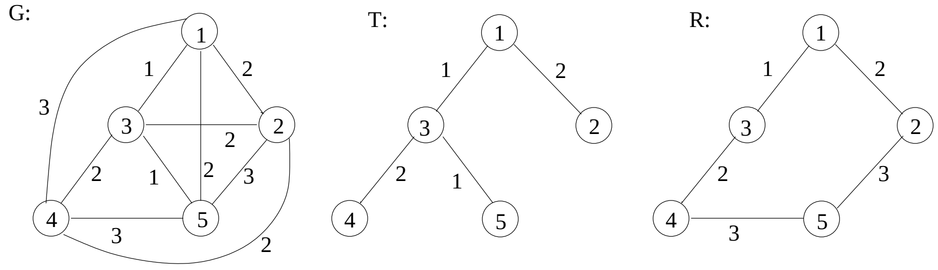
Sei

$G = (V, E, D)$ eine Eingabe für ΔTSP, $V = \{1, \ldots, n\}$.

Idee für $\mathcal{A}$ bei Eingabe G.

a) Erstelle billigsten aufspannenden Baum T für G.
b) Laufe einmal um T herum.
c) Bilde Rundreise durch G durch Benutzung von Abkürzungen.

Beispiel $n = 5$



Weg W um T herum
1–3–4–3–5–3–1–2–1

Rundreise R
1–3–4–5–2–1

Beweis zu Satz 10.9 (Forts.)

Sei $l(W) =$ Länge von W.
$l(R) =$ Länge von R.
$d(T) = \sum_{(i,j)\in T} d(i,j)$ Gewicht von T.

Beachte $l(R) \leq l(W)$ (wegen Δ-Ungleichung) z. B.

$$\begin{aligned} d(5,2) &\leq d(5,3) + d(3,2) \\ &\leq d(5,3) + d(3,1) + d(1,2) \end{aligned}$$

Es gilt
- T ist in Polynomzeit aus G berechenbar (nach Abschnitt 4.6).
- R ist in Polynomzeit aus G berechenbar.
- $d(T) \leq \text{opt}(G)$ (denn: Sei R_{opt} optimale Rundreise, R_{opt} ohne eine Kante ist aufspannender Baum für G, T ist billigster aufspannender Baum).
- $l(R) \leq 2d(T) = l(W)$ (wegen der Dreiecksungleichung).
- $\mathcal{A}(G) = l(R) \leq 2\text{opt}(G)$.

Also
- $\mathcal{A}$ ist Polynomzeit-Algorithmus.
- $\dfrac{\mathcal{A}(G) - \text{opt}(G)}{\text{opt}(G)} \leq 1$

10.4 Polynomiale Approximationsschemata

Wir geben polynomiale Approximationsschemata an für

- SIT Scheduling Independent Tasks.
- RSP Rucksackproblem.

Entwicklung von Algorithmen $\mathcal{A} = \mathcal{A}_k$ für diese Probleme mit

- $\text{RF}_{\mathcal{A}_k}(e) \in O(1/k)$.
- Zeitbedarf von $\mathcal{A}_k$ ist polynomial für jedes k.
- Zeitbedarf von $\mathcal{A}$ ist **nicht** polynomial in $1/k$.

Ein polynomiales Approximationsschema für SIT$_m$

Strategie

LPT$_k$ für SIT$_m$.

1. Sortiere so, dass $l_1 \geq l_2 \geq \cdots \geq l_n$.
2. Erstelle optimalen Plan für die längsten k Prozesse.
3. Verplane die restlichen Prozesse nach LPT.

Zeitbedarf $O(m^k)$.

Satz 10.10.

LPT$_k$ liefert für SIT$_m$ eine $c_{m,k}$-Approximation mit

$$c_{m,k} = \frac{m-1}{k+1}$$

o. Beweis (siehe Horowitz+Sahni).

Beachte

(LPT$_k$)$_{k\in\mathbb{N}}$ liefert ein polynomiales Approximationsschema für SIT$_m$:

- polynomialer Zeitbedarf für festes k.
- Polynomgrad der Zeitschranke $\to \infty$ für $k \to \infty$.
- $c_{m,k} \to 0$ für $k \to \infty$.

Ein polynomiales Approximationsschema für RSP

Idee für $\mathcal{A}_k$

Sei $e = (n, p, w, K)$ Eingabe für RSP.

Sortiere die Elemente $1, \ldots, n$ so, dass

$$p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$$

Löse für jedes $I \subseteq \{1, \ldots, n\}$ mit $\|I\| \leq k$ das Problem exakt und verteile die Restkapazität nach schnellem Näherungsalgorithmus $\mathcal{A}'$.

Algorithmus $\mathcal{A}_k(e)$

```
Sortiere, so dass p_1/w_1 ≥ p_2/w_2 ≥ ··· ≥ p_n/w_n
PROF ← 0
for all I ⊆ {1, ..., n}, ||I|| ≤ k do
    P_I ← Σ_{i∈I} p_i
    if Σ_{i∈I} w_i ≤ K
      then
           PROF ← max{PROF , P_I + A'(I, p, w, K)}
    fi
od
return PROF
```

Approximationsschema für RSP (Forts.)

Algorithmus $\mathcal{A}'(I, p, w, K)$

$S \leftarrow 0;\ T \leftarrow K - \sum_{i \in I} w_i$
for $i = 1$ **to** n **do**
  if $i \notin I$ **and** $w_i \leq T$
   then
     $S \leftarrow S + p_i;\ T \leftarrow T - w_i$
  fi
od
return S

Zeitbedarf

Sei $|e| = m$.

Es gibt $\binom{n}{\mu}$ Teilmengen $I \subseteq \{1, \ldots, n\}$ mit $||I|| = \mu$.

$$T_{\mathcal{A}_k}(m) \in O(n \log n) + O\left(\sum_{\mu=1}^{k} \binom{n}{\mu} \cdot n\right)$$
$$\subseteq O(n^{k+1}) \subseteq O(m^{k+1})$$

Also $\mathcal{A}_k$ hat für jedes k polynomialen Zeitbedarf.

Satz 10.11.

Für jedes k gilt

$$\frac{\text{opt}(e) - \mathcal{A}_k(e)}{\text{opt}(e)} \leq \frac{1}{k+1}$$

Beweis-Idee zu Satz 10.11

Sei $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ optimaler Lösungsvektor zu e und $I(x) = \{i \mid x_i = 1\}$, also opt$(e) = \sum_{i \in I(x)} p_i$.

Gilt $||I(x)|| \leq k$, so folgt $\mathcal{A}_k(e) =$ opt(e).

Sei $||I(x)|| > k$.

$I \subseteq I(x)$ bestehe aus den k Elementen von $I(x)$ mit den größten p-Werten.

Also $p_i \leq \text{opt}(e)/(k+1)$ für alle $i \in I(x) - I$
$\mathcal{A}_k(e) = P_I + S$ S aus $\mathcal{A}'$

Sei $j \in I(x) - I$ minimal, so dass j in $\mathcal{A}'$ nicht in den Rucksack aufgenommen wird. Beim Stopp von $\mathcal{A}'$ gilt

$$\sum_{i \in I(x)-I} p_i \leq S + p_j/w_j T \leq S + p_j \qquad \text{(Übung)}$$

$$\text{opt}(e) = P_I + \sum_{i \in I(x)-I} p_i \leq P_I + S + p_j$$

$$p_j \leq \frac{\text{opt}(e)}{k+1} \qquad \text{wegen } j \in I(x) - I$$

Also
$$\frac{\text{opt}(e) - \mathcal{A}_k(e)}{\text{opt}(e)} \leq \frac{p_j}{\text{opt}(e)} \leq \frac{1}{k+1}$$

10.5 Voll-polynomiale Approximationsschemata

Ziel

RSP hat ein voll-polynomiales Approximationsschema.

Idee

Sei e Eingabe für RSP, ε Genauigkeitsschranke.

Transformiere e in e' mit

- $\mathrm{opt}(e')$ in Polynomzeit berechenbar.
- $\dfrac{\mathrm{opt}(e) - \mathrm{opt}(e')}{\mathrm{opt}(e)} \leq \varepsilon$

Name der Technik

Rundung (engl. Rounding).

Andere Techniken

a) Interval Partitioning
Es wird nicht e transformiert, sondern zu e nicht der ganze Suchraum nach zulässigen Lösungen durchsucht.

b) Separation
Ansatz ähnlich wie bei a)

Siehe Horowitz+Sahni.

Algorithmen für RSP

Sei $e = (n, p, w, K), \varepsilon > 0$.

1. Wähle $UB = UB(e)$ mit $\mathrm{opt}(e) \geq UB$.
   Setze $\beta = \frac{UB \cdot \varepsilon}{n}$.
2. Setze $e' = (n, q, w, K)$ mit
$$q_i = p_i - \mathrm{rem}(p_i, \beta) = \left\lfloor \frac{p_i}{\beta} \right\rfloor \beta \qquad 1 \leq i \leq n$$
3. Berechne $\mathrm{opt}(e')$ nach der Strategie Dynamisches Programmieren und verwende Säuberungstechniken.
4. $\mathcal{A}_\varepsilon(e) \leftarrow \mathrm{opt}(e')$.

Bemerkung

a) Zur Wahl von UB:
$UB_1 = \max\{p_i \mid 1 \leq i \leq n, w_i \leq K\}$
$UB_2 = \sum_{i \in I} p_i$ mit
$I = \{i \mid x_i = 1\}, x = (x_1, \ldots, x_n)$ opt. Lösung zum Bruchteil-RSP.

b) Zur Berechnung von $\mathrm{opt}(e')$:
$f(i, g) = \mathrm{opt}(e_{i,g})$,
$e_{i,g} = (i, (q_1, \ldots, q_i), (w_1, \ldots, w_i), g)$ ist für festes i eine monotone Treppenfunktion. Berechne für $i = 1, \ldots, n$ die Folge $S^{(i)}$ der Sprungstellen von $f(i, g)$ (siehe Abschnitt 3.5).

Ergebnis

Satz 10.12.

Der Algorithmus $\mathcal{A}_\varepsilon$ lässt sich so implementieren, das gilt

- $\dfrac{\mathsf{opt}(e) - \mathcal{A}_\varepsilon(e)}{\mathsf{opt}(e)} \leq \varepsilon$
- $\mathcal{A}_\varepsilon$ hat Zeitbedarf $T_\varepsilon(n) \in O(n^3 \cdot 1/\varepsilon)$

Beweis: Siehe Horowitz + Sahni.
Die zentrale Idee ist, dass durch die „Rundung“ der p_i zu den q_i die Anzahl der Sprungstellen in den $S^{(i)}$ durch $O(n^3/\varepsilon)$ beschränkt ist.

Bemerkung

Es gibt noch effizientere voll-polynomiale Approximationsschemata für RSP, zum Beispiel Zeitbedarf.

$$T_\varepsilon(n) \in O(n(\log n + 1/\varepsilon^2))$$

Siehe z. B. Ibarra+Kim, J. ACM 22 (1975) 463-468.

Generelle Bemerkung

In diesem Kapitel wurde die Frage untersucht, welche NP-vollständigen Probleme sich wie gut polynomial approximieren lassen. Dabei wurde als Zeitbedarf der Zeitbedarf im schlechtesten Fall festgelegt.

Für praktische Anwendungen ist häufig die Frage wichtiger, ob es Algorithmen gibt, die im Mittel wenig Zeit benötigen und dennoch Näherungswerte mit abschätzbarer Abweichung vom Optimum liefern.

Hierzu gibt es zwei Ansätze:

a) Auf das spezielle Problem zugeschnittene Verfahren.
b) Generelle Verfahren, die den Zufall ausnutzen.

Zum Ansatz b) gehören Verfahren wie probabilistische Algorithmen, genetische Algorithmen und Simulated Annealing.

Für beide Ansätze wird auf Spezialvorlesungen bzw. auf die Literatur verwiesen.

Literatur

- Bentley, Jon: *Programming Pearls*.
  Addison-Wesley, 2000.
  Signatur: INF 410/237.
- Brassard, Gilles; Bratley, Paul: *Algorithmik: Theorie und Praxis*.
  Wolfram, 1993.
  Signatur: INF 335/130.
- Brassard, Gilles; Bratley, Paul: *Fundamentals of algorithmics*.
  Prentice-Hall International, 1996.
  Signatur: INF 335/133.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.:
  Introduction to algorithms.
  MIT Press, 1996.
  Signatur INF 335/097.
- Gary, Michael R.; Johnson, David S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*.
  W. H. Freeman and Company, 1979.
  Signatur: L inf 188, INF 370/004.
- Horowitz, Ellis; Sahni, Sartaj:
  Fundamentals of computer algorithms.
  Pitman, 1979.
  Signatur: L inf 120, INF 335/051.

Literatur (Forts.)

- Mehlhorn, Kurt: *Effiziente Algorithmen*.
  Teubner, 1977.
  Signatur: L inf 423.
- Rawlins, Gregory J. E.: *Compared to what? An introduction to the analysis of algorithms*.
  Computer Science Press, 1992.
  Signatur: INF 335/143.
- Schöning, Uwe: *Algorithmen - kurz gefasst*.
  Spektrum Akademischer Verlag, 1997.
  Signatur: L inf 60, INF 335/150.
- Sedgewick, Robert: *Algorithmen*.
  Addison-Wesley, 1992.
  Signatur: INF 335/117.