Vorlesungsmitschrift zur Vorlesung Höhere Algorithmik

Naja v. Schmude
www.najavonschmude.de

WS 2008/2009*

*Letztes Update: 18. Oktober 2010

Inhaltsverzeichnis

Vorlesung 1 und 2, 13/15. Oktober 2008 **5**
1 Algorithmen, Effizienz und Berechnungsmodelle 5
1.1 Fünf Beispiele für Sortieralgorithmen 5

Vorlesung 3, 20. Oktober 2008 **10**
1.2 Berechnungsmodelle 11

Vorlesung 4, 24. Oktober 2008 **13**
2 Suchen und Sortieren 14
2.1 Bemerkungen zum Sortieren 14

Vorlesung 5, 27. Oktober 2008 **15**
2.2 Das Auswahlproblem (Selection, order statistics) 16

Vorlesung 6, 31. Oktober 2008 **18**
2.3 Binärsuche 19

Vorlesung 7, 3. November 2008 **21**

Vorlesung 8, 7. November 2008 **24**
3 Datenstrukturen 24
3.1 Wörterbuch 24

Vorlesung 9, 10. November 2008 **27**
3.2 Optimale binäre Suchbäume 28

Vorlesung 10, 14. November 2008 **29**

Vorlesung 11, 17. November 2008 **32**
3.3 Das Vereinige-Finde-Problem (Union-Find) 32

Vorlesung 12, 21. November 2008 **34**

Vorlesung 13, 24. November 2008 **37**
4 Graphenalgorithmen 37
4.1 Definitionen 37

4.2 Traversierung von Graphen 38

Vorlesung 14, 28. November 2008 **39**
4.3 Minimal spannende Bäume 39
4.4 Wegeprobleme in Graphen 40

Vorlesung 15, 1. Dezember 2008 **42**

Vorlesung 16 und 17, 5/8. Dezember 2008 **44**
4.5 Flüsse in Netzen 44

Vorlesung 18, 12. Dezember 2008 **47**

Vorlesung 19, 15. Dezember 2008 **50**
4.6 Bipartites Matching 50
5 String-Matching 51
5.1 Algorithmus von Knuth-Morris-Pratt 51

Vorlesung 20, 5. Januar 2009 **53**
5.2 Suffixbäume 54

Vorlesung 21, 9. Januar 2009 **56**
6 NP-Vollständigkeit 56
6.1 Beispiele 57

Vorlesung 22, 12. Januar 2009 **58**
6.2 Die Klassen P und NP 58
6.3 Polynomzeit Reduktionen 59

Vorlesung 23, 16. Januar 2009 **60**
6.4 Schaltkreis-Erfüllbarkeit 60

Vorlesung 24, 19. Januar 2009 **64**
6.5 Weitere NP-vollständige Probleme 64

Vorlesung 25, 23. Januar 2009 **65**

Vorlesung 26, 26. Januar 2009 **67**

Vorlesung 27, 30. Januar 2009 **70**
6.6 Komplexitätsklasse Co-NP 70
6.7 Jenseits von NP 70
6.8 Jenseits von PSPACE 71

Vorlesung 28, 2. Februar 2009 **73**
7 Approximationsalgorithmen 73
7.1 Approximationsalgorithmen für Δ-TSP 74

Vorlesung 29, 6. Februar 2009 **77**
7.2 Nichtapproximierbarkeit des allgemeinen TSP 78

Vorlesung 30, 9. Februar 2009 **80**
7.3 PTAS für SUBSET-SUM 80

Vorlesung 1 und 2, 13/15. Oktober 2008

1 Algorithmen, Effizienz und Berechnungsmodelle

Definition 1 (Algorithmus). *Ein Algorithmus ist eine Methode zu Lösung eines Problems. Man kann ihn auch als mathematische Formalisierung verstehen.*

Bei mehreren Algorithmen für das selbe Problem, kann man sich noch Gedanken über die *Effizienz* machen, hier einfach mal als Verbrauch von Ressourcen zu sehen. So kann man schon beim *Entwurf* von Algorithmen die Effizienz mit zu betrachten.

1.1 Fünf Beispiele für Sortieralgorithmen

Gegeben sie eine Folge S von Elementen $a_1, a_2, \ldots a_n$ eines linear geordneten Universums. Der Algorithmus soll jetzt die Folge S aufsteigend geordnet berechnen, d.h. man will eine Permutation $i_1, i_2, \ldots i_n$ finden, so dass die Folge sortiert ist, d.h. $a_{i_1} \leq a_{i_2} \leq \ldots a_{i_n}$.
Jetzt zu den Beispielen.

Algorithmus 1

1. Erzeuge systematisch alle Permutationen der Folge.
2. Für jede Permutation, prüfe ob sie sortiert ist.
3. Falls ja, gebe sie aus.

Der Algorithmus ist auch unter *Bogosort* bekannt.
Es gibt $n!$ Permutationen und für jede benötigt man $n-1$ Vergleiche. Im schlechtesten Fall benötigt man $n!(n-1)$ Vergleiche. Im Mittel sind es dann noch $\frac{n!}{2}$ ($\Omega(n!)$), was immer noch exponentiellem Wachstum entspricht.

Algorithmus 2

1. Durchlaufe S und finde das Minimum.
2. Gib das Minimum aus.

3. Behandle die Restfolge auf die gleiche Art, bis alles abgearbeitet ist.

Das ist Sortieren durch Auswahl bzw. *Selection-Sort*.
Zum Finden des Minimums benötigt man im ersten Durchlauf $n-1$ Vergleiche, danach nur noch $n-2$, dann $n-3$ etc. Beim $(n-1)$-ten Durchlauf nur noch 1 Vergleich. Es ergibt sich also $\frac{n(n-1)}{2}$ für die Anzahl der Vergleiche insgesamt. Die Laufzeit ist also $\Theta(n^2)$.

Algorithmus 3

1. Falls die Folge aus nur einem Element besteht, gib es aus.
2. Ansonsten teilen wir S in $S_1 = a_1, a_2, \ldots a_{\frac{n}{2}}$ und $S_2 = a_{\frac{n}{2}+1}, \ldots a_n$
3. Sortiere S_1 und S_2 rekursiv.
4. Mische die sortierten Teilfolgen zu einer sortierten Gesamtfolge.

Das ist natürlich *Mergesort*.
Sei C_n die Anzahl der Vergleiche bei einer Folge von n Elementen. Wir wissen

$$\begin{aligned} C(1) &= 0 \\ C(n) &\leq 2C(\frac{n}{2}) + (n-1) \end{aligned}$$

Wir nehmen dabei an, dass n eine Zweierpotenz ist. Wir können das jetzt entwickeln.

$$\begin{aligned} C(n) &\leq 2C(\frac{n}{2}) + n \\ &\leq 2 * (2C(\frac{n}{4}) + \frac{n}{2}) + n = 4C(\frac{n}{4}) + n + n \\ &\leq 2^k C(\frac{n}{2^k}) + kn \end{aligned}$$

Falls $k = \log n$ dann folgt

$$\begin{aligned} C(n) &\leq 2^k C(0) + n \log n \\ &\leq n \log n \end{aligned}$$

Es gilt also $\Theta(n \log n)$.

Algorithmus 4

1. Falls S nur aus einem Element besteht, gib es aus.
2. Sonst wähle zufällig einen Index $j \in 1 \ldots n$.
3. Durchlaufe S und teile es in S_1 mit den Elementen, die $< a_j$, S_2 mit allen Elementen $= a_j$ und S_3, die alle größer a_j sind.

4. Man sortiert dann rekursiv S_1 und S_3
5. Gib S_1, S_2 und S_3 in dieser Reihenfolge aus.

Das ist natürlich *Quicksort*.
Es ist auch ein randomisierte Algorithmus, daher hängt die Laufzeit vom Zufall ab. Daher betrachtet man die erwartete Anzahl der Vergleiche. Zunächst aber erstmal der schlechteste Fall. Dieser tritt ein, wenn das ausgewählte Element (Pivotelement) das kleinste bzw. größte ist. Dann macht man $n-1$ Vergleiche zum Aufspalten im Schritt 2. Es folgt, dass beim Aufteilen aller anderen Elemente $S_1 = \emptyset$, $S_2 = \{a_j\}$ und S_3 hat alle anderen. Rekursiv geht es dann so weiter. Wir haben hier also $n-1+n-2+\ldots+1 = \frac{n(n-1)}{2}$, was $\Theta(n^2)$ entspricht. Jetzt zum Erwartungswert $T(n)$. Wir wissen

$$
\begin{aligned}
T(1) &= 0 \\
T(n) &= \sum_{k=1}^{n} \underbrace{\frac{1}{n}}_{\text{Wkeit, dass Pivot das k kleinste}} (T(k-1) + T(n-k)) + \underbrace{n-1}_{\text{zum Aufspalten}} \\
T(n) &= \frac{2}{n} \sum_{k=0}^{n-1} T(k) + n - 1 \quad | \quad * n \\
nT(n) &= 2 \sum_{k=0}^{n-1} T(k) + n(n-1)
\end{aligned}
$$

Für $n-1$ gilt dann

$$
(n-1)T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + (n-1)(n-2)
$$

Nun subtrahiert man die Zeile mit $nT(n)$ mit der von $(n-1)T(n-1)$ und erhält

$$
\begin{aligned}
nT(n) - (n-1)T(n-1) &= 2T(n-1) + 2(n-1) \\
nT(n) &= (n+1)T(n-1) + 2(n-1)
\end{aligned}
$$

Nun können wir die Formel entwickeln.

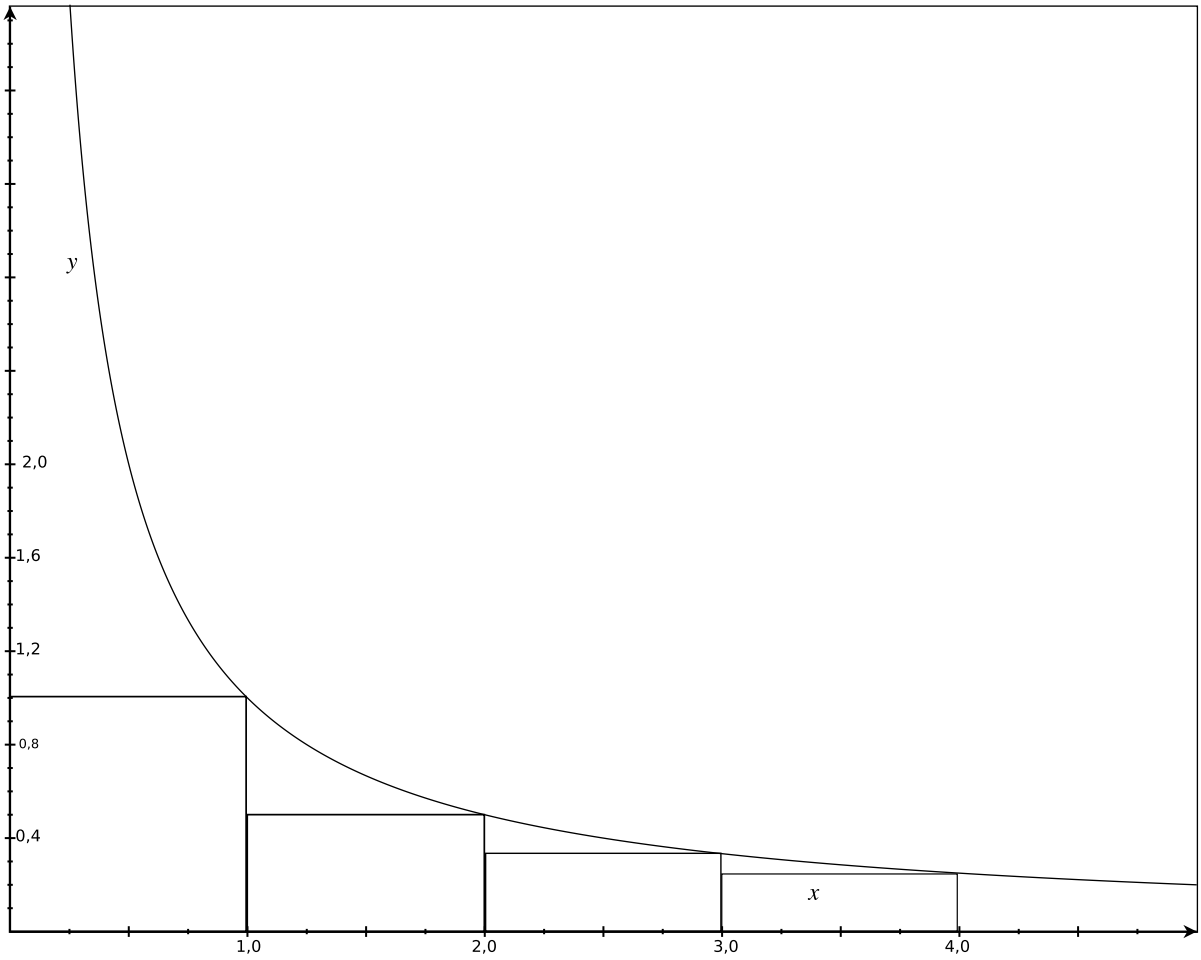
$$
\begin{aligned}
T(n) &= \frac{n+1}{n} T(n-1) + 2\frac{n-1}{n} \\
T(n) &\leq \frac{n+1}{n} T(n-1) + 2 \\
T(n) &\leq \frac{n+1}{n} \left(\frac{n}{n-1} T(n-2) + 2 \right) + 2 \\
&= \frac{n+1}{n-1} T(n-2) + 2\frac{n+1}{n} + 2\frac{n+1}{n+1}
\end{aligned}
$$

$$
\begin{aligned}
T(n) &\leq \frac{n+1}{n-1}\left(\frac{n-1}{n-2}T(n-3)+2\right)+2\frac{n+1}{n}+2\frac{n+1}{n+1}\\
T(n) &\leq \frac{n+1}{n-2}T(n-3)+2\frac{n+1}{n-1}+2\frac{n+1}{n}+2\frac{n+1}{n+1}\\
&\vdots\\
T(n) &\leq \frac{n+1}{2}T(1)+2\frac{n+1}{3}+2\frac{n+1}{4}+\ldots+2\frac{n+1}{n+1}\\
&= 2(n+1)*\left(\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n+1}\right)\\
&= H_{n+1}-\frac{3}{2}
\end{aligned}
$$

mit

$$H_n = 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}$$

Man kann die harmonische Zahl graphisch abschätzen, indem man die Flächen aufmalt und diese durch die Fläche unter einer Funktion abschätzt.



Die Funktion in dem Fall ist $f(x) = \frac{1}{x}$. Es gilt dann

$$\begin{aligned} H_n &\leq \int_1^n \frac{1}{x} dx + 1 = \ln x|_1^n + 1 = \ln n + 1 \\ H_n &\geq \int_0^n \frac{1}{x+1} dx = \ln(x+1)|_0^n = \ln(x+1) \\ \ln(n+1) &\leq H_n \leq \ln n + 1 \end{aligned}$$

Damit gilt

$$\begin{aligned} T(n) &\leq 2(n+1)(\ln(n+1) + 1 - \frac{3}{2}) \\ &= 2n \ln n + \text{niedrigere Terme} \\ &\approx 1,38 n \log n \\ &= \Theta(n \log n) \end{aligned}$$

Die Anzahl der Vergleiche ist zwar größer als bei Mergesort, allerdings ist Quicksort dennoch der bessere.

Algorithmus 5

Und jetzt der letzte Algorithmus. Wiederhole die Schritte k mal:

1. Ziehe zufällig zwei Elemente a_i und a_j aus S mit $i < j$.
2. Falls sie in falscher Reihenfolge sind, also $a_i > a_j$, werden sie vertauscht.

Es stellt sich die Frage, wie groß man k wählen muss, damit mit hoher Wahrscheinlichkeit die Folge anschließend sortiert ist. Es interessiert uns also nach wie vielen Schritten im Mittel die Folge sortiert ist.
Eine Inversion liegt vor, wenn $i < j$ mit $a[i] > a[j]$. Der Algorithmus reduziert die Zahl der Inversionen in einem Schritt um mindestens 1. Die Frage, wie groß man das k wählen muss, damit die Folge sortiert ist, ist äquivalent zum *Coupon Collector's Problem*. Das Problem beschäftigt sich damit, wie viele Proben aus Menge $\{c_1, \ldots, c_n\}$ im Mittel notwendig sind, damit jedes c_i ausgewählt wurde. Es gilt im Mittel $n \ln n$.
In der Vorlesung werden wir uns mit der Analyse nicht weiter beschäftigen.

Vorlesung 3, 20. Oktober 2008

Noch zu den Algorithmen ein paar Zahlen zu den unterschiedlichen Laufzeiten.

Vergleich der benötigten Anzahl der Vergleiche

n	$n \log n$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$n!$
10	33	45	$3,6 * 10^6$
20	86	190	$2,4 * 10^{18}$
50	282	1125	$3,0 * 10^{64}$

Größe der Probleme, die eine Maschine mit 10^9 Vergleichen pro Sekunde lösen kann

	$n \log n$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$n!$
1 Sekunde	$4 * 10^7$	$4 * 10^4$	13
1 Stunde	$1 * 10^{11}$	$2,6 * 10^6$	16

Welche Lehren können wir nun aus den Beispielen ziehen?

- Die Analyse von Algorithmen ist sehr wichtig.
- Die Effizient von Algorithmen ist schon in etwa ohne Implementierung, sogar bei umgangssprachlicher Formulierung, berechenbar. Das ist die *Analyse* von Algorithmen.
- Quicksort und Mergesort beruhen beide auf dem Prinzip *Teile und Herrsche* (divide & conquer). Das heißt, teile das Grundproblem in kleinere Probleme, löse diese rekursiv und kombiniere anschließend die Lösungen der Teilprobleme zu einer Gesamtlösung. Bei der Analyse von solchen Algorithmen treten *Rekursionsgleichungen* für die Laufzeit o.ä. auf, die gelöst werden müssen.
- Es gibt Algorithmen, die den *Zufall* zu Hilfe nehmen. Diese Algorithmen nennt man randomisierte bzw. probabilistische Algorithmen. Der Zufall kann unterschiedliche Folgen habe. Bei Quicksort ist das Ergebnis in jedem Fall richtig, nur die Laufzeit hängt vom Zufall ab. Bei dem letzten Algorithmus hängt jedoch das Ergebnis vom Zufall ab, die Laufzeit wird durch k vorherbestimmt. Die Algorithmen, bei der die Laufzeit mit hoher Wahrscheinlichkeit effizient ist, nennt man *Las Vegas-Algorithmen*, die bei denen das Ergebnis mit hoher Wahrscheinlichkeit richtig ist, nennt man *Monte-Carlo-Algorithmen*.

1.2 Berechnungsmodelle

Berechnungsmodelle sind mathematische Modelle für Rechner, um Begriffe wie berechenbar, Algorithmus, Laufzeit, Platzbedarf usw. exakt definieren zu können.

Turingmaschine

Eins dieser Modelle ist die *Turingmaschine*, die 1936 von Turing erfunden wurde.

Laufzeit bei Eingabe w ist einfach als Anzahl der Schritte anzunehmen, bis die Turingmaschine hält.

Platzbedarf wäre bei Eingabe w die Zahl der Zellen des Bandes, die beschrieben werden.

Registermaschine, (RAM random access machine)

Die Registermaschine besteht aus unendlich vielen Registern, welche beliebige ganze Zahlen enthalten können. Ein *Programm* ist eine unendliche Folge von Befehlen der Form

```
A := B op C
A := B
```

wobei A ein Register ist und B, C ebenfalls oder Konstanten $\in \mathbb{Z}$. Operatoren können $+, -, *, /$ sein.

```
goto L
```

L ist eine Zeile des Programms, zu dem durch $goto$ gesprungen wird.

```
GGZ B,L (goto L if B > 0)
GLZ B,L (goto L if B < 0)
GZ B,L (goto L if B = 0)
HALT
```

Ein Zustand einer Registermaschine besteht demnach aus der Nummer der momentan ausgeführten Zeile und dem Inhalt der Register, den man durch eine Funktion $c : \mathbb{N}_0 \to \mathbb{Z}$ beschreiben kann. Die Menge der Zustände bezeichnet man als Z. Jedem Befehl lässt sich nun eine Funktion $Z \to Z$ zuordnen.

Es handelt sich um eine Berechnung einer Funktion $f : \mathbb{Z}^* \to \mathbb{Z}^*$ auf einer Registermaschine M, falls $x_0 \dots x_n$ in den Registern $0, \dots, n$ abgelegt sind, dann läuft M bis zum $HALT$ Befehl und dann befindet sich eine Folge $y_0 \dots y_m$ in den Registern $0, \dots, m$ wobei $f(x_0 \dots x_n) = y_0 \dots y_m$ ist.

Man kann jetzt mehrere Laufzeit- und Platzbedarf-Definitionen aufstellen. Die erste ist das *Einheitskostenmaß (EKM)*.

Laufzeit bei Eingabe $x \in \mathbb{Z}^*$ ist die Anzahl der Schritte der Registermaschine bei der Eingabe, bis sie hält.

Platzbedarf bei der Eingabe $x \in \mathbb{Z}^*$ wäre die Anzahl der benutzten Register.

Diese Definition entspricht der Zeit und dem Platz auf realen Rechnern. Allerdings sind dort nur beschränkt große Zahlen darstellbar, auf der RAM sind beliebig große benutzbar. Deshalb nimmt man das *logarithmische Kostenmaß (LKM)*.

Laufzeit eines Befehls ist die Summe der Länge $L(x)$ der Binärdarstellung der benutzten Zahlen. Z.B. $R_1 = R_2 + (R_3)$ mit $R_2 = 100$, $R_3 = 50$ und $R_1 = 1000$ würde $L(100) + L(50) + L(1000) = 8 + 7 + 11 = 26$ die Laufzeit.

Platzbedarf für ein Register ist L(Größe der enthaltenen Zahl) und wir addieren die Kosten aller Befehle, also den gesamten Platzbedarf und nehmen das Maximum der Summe aller Registerkosten über alle Schritte.

Man nimmt Laufzeit und Platzbedarf meist nicht als Funktionen $t(x), s(x)$ jeder einzelnen Eingabe, sondern ordnet jede Eingabe x eine Größe $g(x) \in \mathbb{N}$ zu, z.B. die Gesamtzahl der Bits um x darzustellen o.ä.. Die Laufzeit ist dann im schlechtesten Fall eine Funktion

$$T(n) = \max_{x, g(x)=n} t(x).$$

Der Platzbedarf funktioniert hier analog.
Die Laufzeit im Mittel benötigt die Annahme einer Wahrscheinlichkeitsverteilung aller Eingaben der Größe n. Dann ist $T(n) =$ Erwartungswert von $t(x)$ bezüglich dieser Verteilung.

Vorlesung 4, 24. Oktober 2008

Bei randomisierten Algorithmen haben wir eine erwartete Laufzeit von

$T(n) = \max_{x,g(x)=n}$ Erwartungswert der Laufzeit bei Eingabe x über alle möglichen Zufallsentscheidungen

Deterministische Algorithmen mit guter Laufzeit im Mittel, können schlechte Eingaben besitzen. Bei randomisierte Algorithmen ist das zu vermeiden, da man hier den Erwartungswert über die Zufallsentscheidungen bildet.

Beispiel Beim randomisierten Quicksort wird das Pivotelement zufällig gewählt. Der Erwartungswert für die Anzahl der Vergleiche ist $O(n \log n)$ für jede Eingabefolge der Länge n. Implementiert man Quicksort hingegen deterministisch, d.h. wir wählen das Pivotelement z.B. immer als erstes Element, dann gibt es schlechte Eingaben. Wenn die Folge nämlich schon sortiert ist, dann ist die Laufzeit bei $O(n^2)$.

Wie vergleichen sich nun Laufzeiten bzw. Speicherverbrauch auf den verschiedenen Berechnungsmodellen?

- Die Laufzeit auf einer RAM mit LKM kann exponentiell langsamer sein als eine RAM mit EKM.
- Die Laufzeiten von Turingmaschinen im Vergleich zu RAMs mit LKM sind polynomiell verwandt. D.h. es gibt Polynome p, q, so dass was man auf einer RAM in Zeit $T(n)$ berechnen kann, auf einer Turingmaschine mit Zeit $O(p(T(n))$ läuft. Umgekehrt gilt für eine Laufzeit von $T(n)$ auf einer Turingmaschine $O(q(T(n))$ für die RAM. Z.B $p(k) = k^5$.

Zum leichteren Verständnis beschreiben wir Algorithmen in einem Mischmasch aus Konstrukten höherer Programmiersprachen (`for`, `while`, `goto`, Rekursionen etc.) und Umgangssprache (*Pseudocode*) oder auch nur umgangssprachlich.
Trotz dieser unexakten Beschreibung kann man sie gut analysieren mit RAM-Laufzeit im EKM, zumindest asymptotisch bei $n \to \infty$ in Form von O.
Für den Platzbedarf bei Rekursionen gilt, dass man auf dem Laufzeitstack Platz braucht von der Größe der Schachtelungstiefe der Rekursionen multipliziert mit dem benötigten Platz für nur einen Rekursionsaufruf.

Typische Funktionen bei der Anaylse von Algorithmen

$\log\log n$	
$\log n$	logarithmisch
$\sqrt{n}$	
n	linear
$n\log n$	
n^2	quadratisch
n^3	kubisch
n^4	biquadratisch
2^n	exponentiell
$n!$	
2^{n^2}	exponentiell
2^{2^n}	doppelt exponentiell

Zum Einordnen von $n!$ kann man diese Zahl durch die *Stirlingsche Formel* abschätzen.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{\Theta}{12n}} \quad \text{für } 0 < \Theta < 1$$

2 Suchen und Sortieren

2.1 Bemerkungen zum Sortieren

Wie verhält sich die Anzahl der Vergleiche vs. der RAM-Laufzeit im EKM?

	Anzahl d. Vergleiche	RAM-Laufzeit
Selectionsort	$\approx \frac{1}{2}n^2$	$2,5n^2$
Mergesort	$\approx n\log n$	$12n\log n$
Quicksort	$\approx 1,38n\log n$	$9n\log n$

Die Frage ist, ob man immer $\Omega(n\log n)$ benötigt?

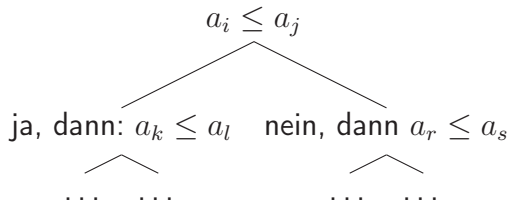
Beispiel Wenn man z.B. 1000 Nullen und Einsen sortieren will, kommt man auf lineare Laufzeit, man kann nämlich einfach alle Nullen und Einsen zählen und dann sortiert ausgeben.

Der Grund für die gute Laufzeit ist hier, dass man ein endliches Universum $U = \{0, 1\}$ hat, aus dem die Daten stammen. Verallgemeinerungen hiervon sind hinlänglich als *Bucketsort* und *Radixsort* bekannt, die beide auch lineare Laufzeit haben. Die vorhergehenden Algorithmen funktionieren für jedes Universum, auf dem eine lineare Ordnung definiert ist. Diese beruhen nur auf Vergleichen zwischen Elementen, daher nennt man sie auch *vergleichsbasierte Algorithmen*.

Vorlesung 5, 27. Oktober 2008

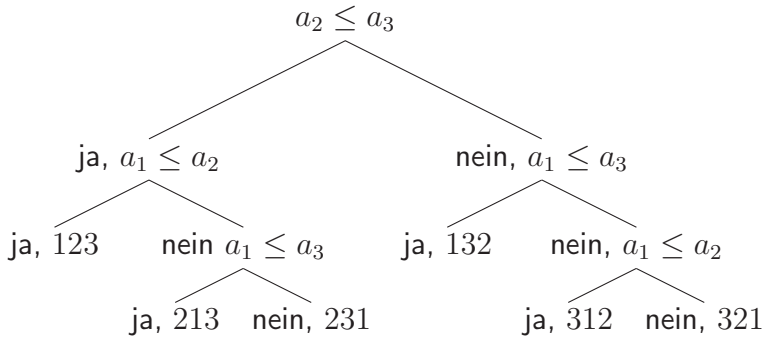
Beispiel für die untere Schranke um das Minimum zu bestimmen. Hier braucht man $\geq n-1$ Vergleiche. Denn jedes Element der Folge außer dem Maximum muss aus mind. einem Vergleich als das kleinere hervorgehen, sonst könnte es auch das Maximum sein.

Vergleichsbasierte Algorithmen lassen sich durch Vergleichsbäume für eine feste Eingabegröße darstellen. Jede Ebene entspricht dabei einem Vergleich.



Verzweigungen im Flussdiagramm für Eingabegröße n.

Beispiel Mergesort mit $n = 3$ bei a_1, a_2, a_3.



An den Blättern haben wir die zur sortierten Folge gehörige Permutation. Wir nehmen an, dass alle Elemente der Eingabefolge verschieden sind. Es gilt dann, dass unterschiedliche Permutationen zu unterschiedlichen Blättern führen. Für eine Eingabefolge der Länge n haben wir also mindestens $n!$ Blätter. Der Vergleichsbaum ist ein Binärbaum und hat deswegen eine Höhe $\geq \log n! = n * \log n - O(n)$ nach der Sterlingschen Formel. Die Höhe ist dabei die Anzahl der Vergleiche im schlechtesten Fall.

Satz 1. *Zum Sortieren einer Folge der Länge n benötigt jeder vergleichsbasierte Algorithmus mindestens $n \log n - O(n)$ Vergleiche nach dem Vergleichsbaummodell.*

2.2 Das Auswahlproblem (Selection, order statistics)

Eine Folge der Länge m sei gegeben. Das Minimum und Maximum können wir schon in $O(n)$ lösen. Wie ist das mit dem Median, also dem Element was bei sortierte Folge an Stelle $\lfloor \frac{n}{2} \rfloor$ liegt oder allgemein, wie kann man bei $k \leq n$ das k-te Element finden?

Ein Schema eines Algorithmus $SELECT(k, S)$ könnte so aussehen:

1. falls $|S| \leq n_0$, bestimme k-tes Element per brute force. sonst
2. wähle ein $a \in S$
3. teile S in S_1 mit Elementen $< a$, S_2 mit Elementen $= a$ und S_3 mit Elementen $> a$.
4. falls $k \leq |S_1|$ gebe $SELECT(k, S_1)$ zurück.
5. falls $k \leq |S_1| + |S_2|$ gebe a zurück
6. gebe $SELECT(k - (|S_1| + |S_2|), S_3)$ zurück.

Wie bei Quicksort gibt es auch hier wieder unterschiedliche Möglichkeiten das Pivotelement a auszuwählen. Bei *randomized selection* wählt man a zufällig gleichverteilt.

Analyse im schlechtesten Fall, wenn alle Elemente verschieden sind, haben wir, wie bei Quicksort $\Theta(n^2)$. Im Mittel gilt

$$T(1) = b$$

$$T(n) = \frac{1}{n} \left(\underbrace{\sum_{i=1}^{k-1} T(n-i)}_{i<k} + \underbrace{\sum_{i=k+1}^{n} T(i-1)}_{i>k, Z.4} \right) + \underbrace{cn}_{Z.1-3}$$

Der erste Teil der Summe ist $T(n-1) + T(n-2) + \ldots + T(n-k+1)$, der zweite Teil ist $T(n-1) + T(n-2) + \ldots + T(k)$.
Wir nehmen an, dass $k \geq \lfloor \frac{n}{2} \rfloor$. Damit lässt sich folgende Abschätzung machen:

$$T(n) \leq \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} T(i) + cn$$

Die Behauptung ist jetzt, dass $T(n) = O(n)$, d.h. $\exists d > 0 : T(n) \leq d * n$. Wir führen jetzt einen Induktionsbeweis über n.

Induktionsanfang $n = 1$ stimmt, wenn $d \geq b$ gewählt wird.

Induktionsschritt Da es für $\leq n-1$ gilt, soll nun folgen, dass es auch für n gilt.

$$T(n) \leq \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} T(i) + cn$$

$$T(n) \leq \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} di + cn \quad \text{nach Induktionsvoraussetzung}$$

Die Konstante d wird vor die Summe gezogen und dann wird sie durch die altbekannte Gaußformel ersetzt. Wir bilden erst die Summe von 1 bis $n-1$ und ziehen dann die Summe von 1 bis $\lfloor \frac{n}{2} \rfloor$ wieder ab.

$$\begin{aligned}
T(n) &\leq \frac{2}{n} d \left(\frac{n(n-1)}{2} - \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor - 1 \right)}{2} \right) + cn \\
&\leq \frac{2}{n} d \left(\frac{n(n-1)}{2} - \frac{\left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} - 2 \right)}{2} \right) + cn \quad \text{mit } \lfloor \frac{n}{2} \rfloor \geq \frac{n}{2} - 1 \\
&= \frac{2}{n} d \left(\frac{3}{8} n^2 + \frac{1}{4} n - 1 \right) + cn \\
&\leq d \left(\frac{3}{4} n + \frac{1}{2} \right) + cn
\end{aligned}$$

Jetzt müssen wir zeigen, dass das $\leq dn$ für geeignet gewähltes d ist.

$$\begin{aligned}
\frac{3}{4} n + \frac{1}{2} &\leq 0,8n \text{ für } n \geq 10 \text{ damit} \\
T(n) &\leq 0.8dn + cn \\
&\leq dn \text{ falls} \\
0,8dn + cn &\leq dn \\
cn &\leq 0,2dn \\
d &\geq 5c
\end{aligned}$$

Jetzt fehlt uns noch alles, was kleiner als 10 ist. Also wählen wir $d \geq \max\{T(1), \ldots, T(10), 5c\}$.

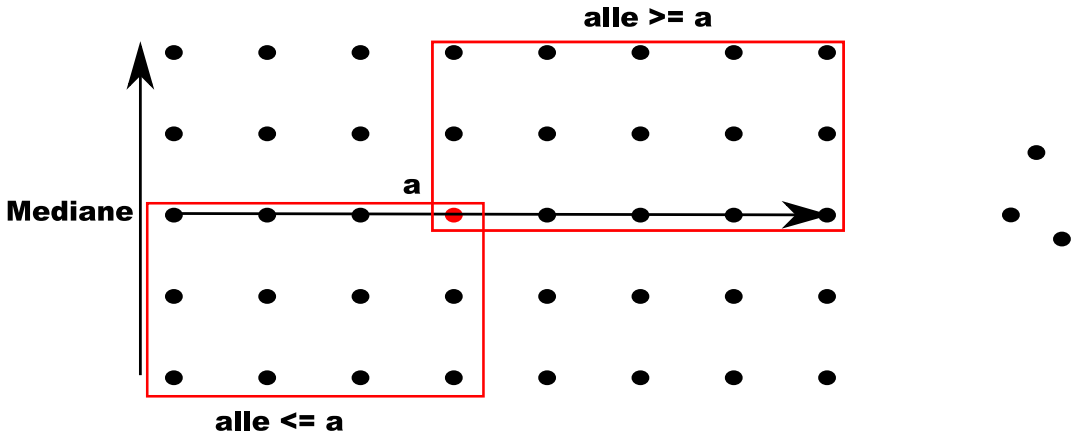
Die erwartete Laufzeit ist also tatsächlich in $\Theta(n)$.

Vorlesung 6, 31. Oktober 2008

Jetzt gucken wir uns einen Auswahlalgorithmus an, der das Pivotelement a deterministisch auswählt, so dass eine lineare Laufzeit erreicht wird.

1. falls $|S| \leq n_0$ bestimme per brute-force das k-te Element. Sonst
2. a) Teile S in $\lfloor \frac{n}{5} \rfloor$ Blöcke der Länge 5 (ignoriere die $0-4$ übrigbleibenden Elemente)
   b) Bestimme von jedem Block den Median
   c) Aus der Folge dieser Mediane, bestimme wieder den Median. Das wird dann unser a durch rekursiven Aufruf von $SELECT$.
3. weiter wie beim anderen $SELECT$.

Analyse Die Schritte 2a) und 2b) haben Kosten von $O(n)$. Für 2c) gilt $T(\lfloor \frac{n}{5} \rfloor)$ wobei $T(n)$ die Laufzeit ist. Wie groß werden nun die Folgen $S_1, \ldots, S_3$ maximal? Das können wir uns veranschaulichen, wobei die Pfeile die Richtung der Sortierung angeben:



Alle in A sind $\geq a$: $S_1 \subset S \backslash A$ und alle in B sind $\leq a$: $S_3 \subset S \backslash B$.

$$|A| \geq 3 * \frac{1}{2} \cdot \left\lfloor \frac{n}{5} \right\rfloor \geq 3 \cdot \left\lfloor \frac{n}{10} \right\rfloor,$$

damit ist $|S_1| \leq |S| - |A| \leq n - 3 \cdot \lfloor \frac{n}{10} \rfloor$. Analog gilt für $|S_3| \leq n - 3 \cdot \lfloor \frac{n}{10} \rfloor$.
Wir schätzen jetzt $n - 3 \cdot \lfloor \frac{n}{10} \rfloor$ ab. Es gilt $n - 3 \cdot \lfloor \frac{n}{10} \rfloor \leq 0,7n + 3 \leq 0,75n$ falls $0,05n > 3$ also $n \geq 60$.

Die Größen von S_1 und S_3 sind also maximal dreiviertel so groß wie die Ursprungsgröße von S. Damit haben wir für die Laufzeit folgendes:

$$T(n) \leq \underbrace{T\left(\left\lfloor \frac{3}{4}n \right\rfloor\right)}_{\text{Aufruf in Zeile 5 oder 6}} + \underbrace{T\left(\left\lfloor \frac{n}{5} \right\rfloor\right)}_{\text{Aufruf in Schritt 2c)}} + \underbrace{cn}_{\text{andere Schritte}} \quad \text{für } n \geq 60$$

Lemma 1. *Falls für eine Funktion $T : \mathbb{N} \to \mathbb{N}$ gilt $T(n) \leq bn$ für alle $n \leq n_0$ wobei n_0 eine Konstante ist und $T(n) \leq T(\lfloor \alpha n \rfloor) + T(\lfloor \beta n \rfloor) + cn$ für alle $n \geq n_0$ wobei $\alpha, \beta > 0$ Konstanten mit $\alpha + \beta < 1$, dann ist $T(n) = O(n)$.*

Das kann man natürlich auch auf noch mehr Summanden erweitern, soweit die Konstanten in Summe < 1 sind.

Beweis des Lemmas Wir führen einen Beweis per Induktion. Wir zeigen $\forall n : T(n) \leq dn$, wenn d geeignet gewählt wird.

Induktionsanfang $n < n_0$ dann ist die Behauptung richten, wenn $d \geq b$ gewählt wird.

Induktionsschritt Es gilt für $\leq n - 1$ so soll auch folgen, dass es für n gilt. $n \geq n_0$.

$$\begin{aligned} T(n) &\leq T(\lfloor \alpha n \rfloor) + T(\lfloor \beta n \rfloor) + cn \\ &\leq d \cdot \lfloor \alpha n \rfloor + d \cdot \lfloor \beta n \rfloor + cn \text{ nach Induktionsvoraussetzung} \\ &\leq d(\alpha + \beta)n + cn \end{aligned}$$

Wir wollen nun, dass $T(n) \leq dn$ herauskommt. Das ist der Fall, wenn $d(\alpha+\beta)+c \leq d$ ist. Also $d \geq \frac{c}{1-(\alpha+\beta)}$. Wir wählen also $d = \max\left(b, \frac{c}{1-(\alpha+\beta)}\right)$.

Zusammenfassung Der randomisierte Auswahlalgorithmus hat die Laufzeit $\Theta(n^2)$ im schlechtesten Falls und $\Theta(n)$ im Mittel. Der deterministische Algorithmus hat hingegen immer die Laufzeit von $\Theta(n)$.

2.3 Binärsuche

$Binsuch(l, r, a)$ durchsucht ein sortiertes Feld S von der Stelle $S[l]$ bis $S[r]$ nach einem Element a.

```
1 Binsuch(l,r,a)
2   if l <= r then
3     k = l+r/2
4     if(S[k] = a) then return k
5     else if (S[k] > a) then return Binsuch(l,k-1,a)
6     else return Binsuch(k+1,r,a)
7   else
8     return nicht gefunden
```

Analyse

$$T(1) = c$$
$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + d$$

Hat natürlich als Lösung $O(\log n)$.

Interpolationssuche

Besser ist die Interpolationssuche, die statt des mittleren Elements zu nehmen, das Element versucht zu berechnen, wo a eher zu erwarten ist, falls die Daten gleichverteilt sind. Diese Stelle kann man mit folgender Formel berechnen:

$$k = l - 1 + \left\lceil \frac{a - S[l-1]}{S[r+1] - S[l-1]} \cdot (r - l + 1) \right\rceil$$

Anmerkung: Gespeicherte Daten sind reele Zahlen $\in [0, 1]$ und wir setzten $S[0] = 0$ und $S[n+1] = 1$.
Die Analyse ist sehr schwer, man findet sie auch nur in den Originalarbeiten. Im schlechtesten Fall braucht man $\Theta(n)$. Im Mittel, wenn die Daten $S[1], \ldots, S[n]$ tatsächlich gleichverteilt sind aus $[0, 1]$, ergibt sich dann $O(\log \log n)$.

Vorlesung 7, 3. November 2008

Quadratische Binärsuche

Hier berechnen wir auch wieder den erwarteten Index k wie bei der Interpolationssuche, wo a erwartet wird. Falls $S[k] = a$ ist, sind wir fertig. Falls aber $S[k] > a$ ist und $m = r - l + 1$ sei, dann vergleichen wir a mit $S[k - (\lfloor\sqrt{m}\rfloor + 1)$, $S[k - 2(\lfloor\sqrt{m}\rfloor + 1)], \ldots$ bis wir ein Element finden, dass $\leq a$ ist. Dann durchsuchen wir dieses Segment $S[k - i(\lfloor\sqrt{m} + 1)], \ldots, S[k - (i-1)(\lfloor\sqrt{m}\rfloor + 1)]$ rekursiv. Für $S[k] < a$ suchen wir analog nach rechts.

Analyse Im schlechtesten Fall sind wir mal wieder nicht wirklich gut, es ist irgendwas wie $\Omega(\sqrt{n})$. Um das Mittel zu bestimmen, muss man sich überlegen, wie viele Sprünge (Vergleiche) im Mittel notwendig sind, bis das richtige Intervall der Länge $\lfloor\sqrt{m}\rfloor$ gefunden ist, das a enthält. Das ist unser Erwartungswert C
Sei P_i die Wahrscheinlichkeit, dass man mindestens i Sprünge machen muss. Dann ist

$$\begin{aligned} C &= \sum_{i=1}^{\infty} i \underbrace{(P_i - P_{i+1})}_{\text{Wkt für genau } i \text{ Sprünge}} \\ &= P_1 - P_2 + 2P_2 - P_3 + 3P_3 - 3P_4 + \ldots = P_1 + P_2 + P_3 + \ldots \\ &= \sum_{i=1}^{\infty} P_i \end{aligned}$$

Falls der Algorithmus an dieser Stelle mindestens i Vergleiche (und $i-1$ Sprünge) braucht, so weicht die Stelle, wo sich a befindet, mindestens um $(i-2)\sqrt{m}$ ab. Also ist $|rg(a) - k| \geq (i-2)\sqrt{m}$.
Die Tschebyschev-Ungleichung besagt nun, dass bei einer Zufallsvariable X und einem Wert μ und der Varianz σ^2 gilt

$$P(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2}$$

Die Varianz berechnet sich durch

$$\sigma^2 = E((X - E(x))^2)$$

Wir wenden das nun auf unser Problem an. Die Wahrscheinlichkeit, dass ein $S[j] \leq a$ ist, ist

$$P = \frac{a - S[l-1]}{S[r+1] - S[l-1]}.$$

Die Wahrscheinlichkeit, dass genau q der $S[j] \leq a$ sind ist

$$\underbrace{\binom{m}{q}}_{\text{Anz. Mögl. } q \text{ auszuwählen}} \cdot \underbrace{P^q}_{\text{Wkt, dass } q \text{ ausgew. alle} \leq a} \cdot \underbrace{(1-P)^{m-q}}_{\text{das } m-q \text{ nicht ausgew. alle} > a}$$

Der Erwarungswert der Binomialverteilung ist

$$\mu = E(X) = \sum_{q=0}^{m} q \binom{m}{q} P^q (1-P)^{m-q} = mP.$$

Die Varianz ist dann $\sigma^2 = mP(1-P)$.
Damit können wir das P_i mit Hilfe der Tschebyschev-Ungleichung abschätzen.

$$\begin{aligned}
P_i &\leq \frac{\sigma^2}{t^2} \leq \frac{P(1-P)m}{(i-2)^2 m} \text{ für } i \geq 3 \\
&\leq \frac{\frac{1}{4}}{(i-2)^2} \\
C &= \sum_{i=1}^{\infty} P_i \leq P_1 + P_2 + \sum_{i=3}^{\infty} \frac{\frac{1}{4}}{(i-2)^2} \\
&\leq 2 + \frac{1}{4} \sum_{i=1}^{\infty} \frac{1}{j^2} \\
&= 2 + \frac{1}{4} \cdot \frac{\pi^2}{6} = 2,41\ldots
\end{aligned}$$

Wir erwarten also, dass wir immer ca. 2,41 Sprünge machen müssen.
Sei $T(n)$ die erwartete Laufzeit der quadratischen Binärsuche. Dann gilt

$$\begin{aligned}
T(2) &= b \\
T(n) &= T(\lfloor \sqrt{n} \rfloor) + \underbrace{c}_{\text{konstante Anz. von Sprüngen}}
\end{aligned}$$

Wir nehmen nun an, dass $n = 2^{2^k}$. Dann ergibt sich für $\sqrt{n} = 2^{2^{k-1}}$.

$$\begin{aligned}
T(2^{2^k}) &\leq T(2^{2^{k-1}}) + c \\
&\leq T(2^{2^{k-2}}) + c + c \\
&\vdots \\
&\leq T(2^{2^{k-i}}) + ic
\end{aligned}$$

Für $i = k$ gilt dann $n = 2^{2^k} \Leftrightarrow k = \log\log n$.

$$\begin{aligned}
T(n) &\leq T(2) + kc \\
&\leq d + c * \log\log n
\end{aligned}$$

Die Laufzeit ist also in $O(\log\log n)$.

Zusammenfassung Ein sortiertes Feld mit n Elementen, das nach einem Element a durchsucht werden soll, kann per Binärsuche im schlechtesten Fall und im Mittel in $O(\log n)$ durchsucht werden. Die Interpolationssuche und quadratische Binärsuche benötigt im Mittel $O(\log \log n)$, wenn S zufällig gleichverteilt aus $[0, 1]$ ist.

Vorlesung 8, 7. November 2008

3 Datenstrukturen

Man hat im Allgemeinen eine Menge S von Daten des Universums U und eine Reihe von Operationen, die darauf ausgeführt werden sollen. Das nennt man dann einen *abstrakten Datentyp*. Die Datenstruktur ist nun eine Art, die Daten so abzuspeichern, dass die Operationen effizient ausführbar sind. Man betrachtet zum einen die Vorverarbeitungszeit (preprocessing) um aus der Menge die Datenstruktur aufzubauen, den Speicherplatz und dann die Zeit, um die Operationen auszuführen.

3.1 Wörterbuch

Ein Wörterbuch besteht aus einer Menge $S \subset U$ und den Operationen `Suche(a,S)`, was feststellt, ob $a \in S$ enthalten ist. In der Praxis will man natürlich noch mehr Informationen zu a enthalten. Dann gibt es noch `Einfüge(a,S)`, was bedeutet, dass $S := S \cup \{a\}$ bewirkt. `Streiche(a,S)` löscht a aus der Menge: $S := S \backslash \{a\}$.

... mit einem sortierten Array

Wir brauchen hierbei eine lineare Ordnung auf U. In dem Fall können wir in $O(\log n)$ suchen im schlechtesten Fall (Binärsuche). Das Einfügen und Löschen geht in $\Theta(n)$.

... mit Hashing (Streuspeicherung)

Man hat hier eine Hashfunktion $H : U \to \{1, \ldots, m\}$ mit $h(a)$ gibt die Stelle in einer Tabelle mit m Einträgen an, wo sich a befinden soll.
Bei den Operationen berechnen wir $h(a)$ und führen dann die Operation an der entsprechenden Stelle in der Tabelle aus.
Ein Problem dabei sind *Kollisionen*, da h im Allgemeinen nicht injektiv ist. Es gibt also $a, b \in U$ mit $a \neq b$ aber $h(a) = h(b)$. Als Standardlösung hat man dann eine Liste aller Elemente pro Tabelleneintrag, die den gleichen Hashwert haben. So braucht man dann nur den Hashwert berechnen und die entsprechende Liste nach dem Element durchsuchen. Es ist also ein Anliegen, diese Listen möglichst kurz zu halten, am besten mit einer konstanten Länge. Im Mittel ist das auch möglich. Und zwar falls die Elemente zufällig und gleichverteilt aus dem Universum entstammen und die Hashfunktion für jeden Wert i die gleiche Wahrscheinlichkeit hat. Unter dieser Annahme haben alle Operationen eine Laufzeit von $O(1)$.

Die Größe der Hashtabelle ist $m \geq cn$ für eine Konstante c. Sollte durch viel Einfügen die Ungleichung verletzt werden, legt man einfach eine größere Hashtabelle an und dann stimmt es wieder. Wenn man das geschickt macht, bleibt auch die Laufzeit $O(1)$.
Wichtig ist hier, dass man *keine* lineare Ordnung braucht!

... mit binären Suchbäumen

Wir haben hier wieder eine lineare Ordnung auf U. Die Elemente von S sind hier in die Knoten enthalten, so dass alle Elemente $\leq w$ im linken Teilbaum liegen, die Wurzel w ist und alle Elemente $> w$ im rechten Teilbaum zu finden sind. Die Blätter entsprechen hier dann erfolglosen Suchen.
Für das Suchen durchlaufen wir rekursiv T_r bzw. T_l. Beim Einfügen finden wir ein Blatt, dass wir dann in einen inneren Knoten mit dem Element verwandeln und zwei Blätter anhängen. Die Laufzeit der Operationen ist dann natürlich $O(h)$, wobei h die Höhe des Baumes ist. Die Höhe liegt also zwischen $\log n$ und n. Damit ist unsere Laufzeit höchstens $\Theta(n)$ und mindestens $O(\log n)$.
Die Frage ist nun, wie ist die Laufzeit im Mittel ist. Wir nehmen uns einen zufälligen binären Suchbaum mit n Elementen her, der Höhe $O(\log n)$ im Mittel. Zufällig heißt hier, dass ein ursprünglich leerer Baum, in den n Elemente eingefügt wurden, wobei jede Permutation von der aufsteigenden Ordnung gleich wahrscheinlich ist.
Um die Zeit im schlechtesten Fall zu verbessern und auch hier $O(\log n)$ zu erreichen benutzt man ausgeglichene bzw. balancierte Suchbäume.

... mit AVL-Bäumen

Ein AVL-Baum ist ein binärer Suchbaum, bei dem sich für jeden inneren Knoten die Höhen des rechten und linken Teilbaums um höchstens eins unterscheiden. Dadurch ist die Höhe eines solchen Baums $O(\log n)$. Als Beweisidee betrachten wir die minimale Anzahl der Knoten eines AVL-Baumes der Höhe h. Dabei ergibt sich eine rekursive Formel

$$n_h = n_{h-1} + n_{h-2} + 1$$

n_h ist also Größer als die $(h-1)$-te Fibonacci-Zahl, die ja bekannterweise exponentiell wächst. Also $n_k \geq c^k$ für ein $c > 1$. Damit ist $h \leq \log_c n$. Also ist die Höhe tatsächlich in $O(\log n)$.
Um diese Höheneigenschaft beizubehalten, benötigen wir bei Einfüge- und Streich-Operationen Operationen, die das Gleichgewicht wieder herstellen. Da gibt es die *Rotation* und die *Doppelrotation*, siehe ALP 3. Diese Operationen gehen in $O(\log n)$, denn eine Rebalancierungsoperation kostet $O(1)$ und wir müssen maximal den Suchpfad uns angucken. Damit gehen alle Operationen des Wörterbuchproblems in $O(\log n)$.

... mit gewichtsbalancierten binären Suchbäumen (BB$[\alpha]$-Bäume

Sei v ein inneren Knoten. Dann ist die Balance definiert durch

$$balance(v) = \frac{\text{Anzahl Blätter in } T_l}{\text{Anzahl Blätter in } T}$$

Es ist ein BB[α]-Baum, wenn für alle inneren Knoten v gilt, dass $balance(v) \in [\alpha, 1-\alpha]$ und $\alpha \in [0, \frac{1}{2}]$. Falls $\alpha \in [\frac{1}{4}, 1 - \frac{\sqrt{2}}{2}]$, dann lässt sich die BB[α]-Eigenschaft durch Rotation und Doppelrotation wieder herstellen.

Vorlesung 9, 10. November 2008

... mit (a,b)-Bäumen

$a, b \in \mathbb{N}$ mit $b \geq 2a - 1$ und $a > 2$. Jeder innere Knoten hat mindestens a Kinder, außer der Wurzel, die hat mindestens 2 Knoten. Jeder Knoten hat auch maximal b Kinder und alle Blätter haben die gleiche Tiefe.
Es folgt, dass für die Anzahl der Blätter n bei Höhe h gilt

$$2a^{h-1} \leq n \leq b^h$$

Dadurch können wir die Höhe durch Logarithmieren bestimmen

$$\begin{aligned} 1 + (h-1)\log a \leq \log n &\leq h \log b \\ \frac{\log n}{\log b} \leq h &\leq \frac{\log n}{\log a} + 1 \\ \log_b n \leq h &\leq \log an + 1 \end{aligned}$$

Damit ist die Höhe also in $\Theta(\log n)$.
In der Praxis hat man dann meist n in der Größenordnung von 10^6 und a, b sind bei mehreren Hundert, daraus ergibt sich dann eine Höhe von 3 oder 4.
Wie wird nun eine Datenmenge S in einem (a,b)-Baum dargestellt? Die Blätter enthalten von links nach rechts die Elemente von S aufsteigend sortiert. Jeder innere Knoten mit k Kinder enthält $k - 1$ Elemente von S als Schlüssel, so dass in dem i-ten Unterbaum die Elemente zwischen den Schlüsseln a_i und a_{i+1} liegen. Die Schlüssel sind immer die größten Elemente der k Kinder.

Beim Suchen durchlaufen wir den Baum bei der Wurzel beginnend und suchen jeweils den Teilbaum, in dem a befindet. Wir brauchen pro Knoten des Suchpfads konstante Zeit, dadurch ergibt sich eine Laufzeit zum Suchen von $O(\log n)$.
Beim Einfügen suchen wir nach a und fügen ein neues Blatt mit a an der richtigen Stelle ein. Falls der Vater $> b$ Kinder hat, muss er in zwei Knoten aufgespalten werden, die dann jeweils a Kinder haben. Das machen wir den Suchpfad entlang zu Wurzel, da sich in jeder Ebene was verschieben kann. Wir brauchen dafür wieder $O(\log n)$.
Beim Streichen suchen wir wieder a und löschen das Blatt. Falls nun der Vaterknoten zu wenig Kinder hat, so müssen entweder zwei Vaterknoten miteinander verschmolzen werden oder ein Kind wird abgegeben. Das kann auch auf der ganzen Länge des Suchpfads passieren. Somit ist auch hier wieder $O(\log n)$. (a,b)-Bäume nennt man B-Bäume, wenn $b = 2a-1$.

Zugriffe auf Hintergrundspeicher dauern mehrere tausend mal länger als Hauptspeicherzugriffe. Daher nutzt man diese Bäume, um möglichst die Hintergrundspeicherzugriffe zu minimieren. Dazu wählt man a und b so groß, dass innere Knoten etwa die Größe eine Seite haben. So entsprechen 3-4 Schritten innerhalb des B-Baums auch 3-4 Hintergrundspeicherzugriffen.

... mit Rot-Schwarz-Bäume

Rot-Schwarz-Bäume sind binäre Suchbäume, bei denen die Knoten gefärbt sind, entweder rot oder schwarz. Dabei hat jeder rote Knoten zwei schwarze Kinder und die Anzahl der schwarzen Knoten auf jedem Pfad von der Wurzel zu einem Blatt ist gleich. Die Höhe ist also in $O(\log n)$. Für die Operationen kann man das so machen, dass wenn man rote Knoten mit ihren schwarzen Vätern verschmelzen lässt, dann führt das zu (2,4) Bäumen. Die Analyse dafür haben wir ja oben.

... mit Tries

Der Trie ist für Strings konzipiert und ist ein Suchbaum. Für jeden Anfangsbuchstaben der Wortmenge gibt es einen Teilbaum. In der zweiten Ebene kommen dann alle zweiten Buchstaben dran usw, so dass die Wörter auf dem Suchpfad liegen. Dabei markieren wir ein Wortende mit einem speziellen Blatt.
Das Suchen nach einem Wort $w = a_1 \dots a_k$ kostet $O(|w|)$ Zeit genauso wie das Einfügen und Streichen.

3.2 Optimale binäre Suchbäume

Wir haben hier eine statische Menge $S \subset U$ wobei U ein linear geordnetes Universum ist. Sei $S = \{a_1, a_2, \dots a_n\}$ mit $a_1 < a_2 < \dots < a_n$. Zusätzlich haben wir Zugriffswahrscheinlichkeiten $p_1, \dots, p_n$ mit p_i ist die Wahrscheinlichkeit für einen Zugriff auf a_i. Zusätzlich gibt es noch die Wahrscheinlichkeiten $q_0, \dots, q_n$ wobei q_i eine Zugriffswahrscheinlichkeit auf ein Element a mit $a_i < a < a_{i+1}$ ist, q_0 auf a mit $a < a_1$ und q_n auf a mit $a > a_n$. Gesucht sind nun binäre Suchbäume, die die erwartete Anzahl der Vergleiche minimieren.

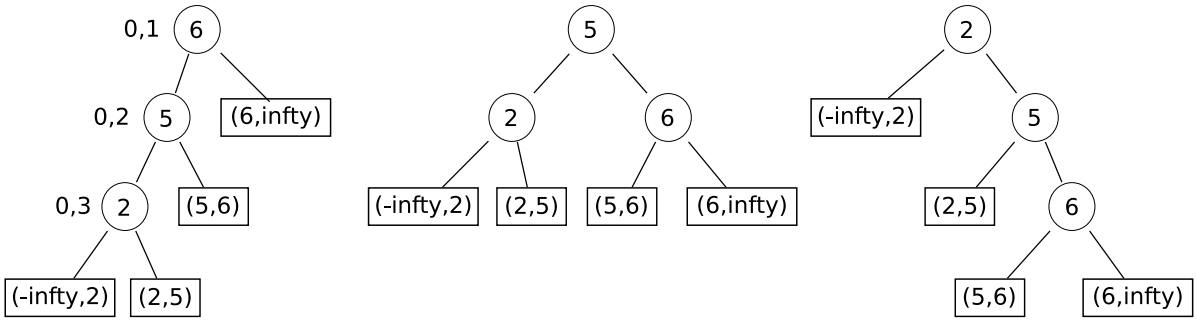
Vorlesung 10, 14. November 2008

Wir haben einen binären Suchbaum T. Dann ist die *gewichtete Weglänge* von T die erwartete Anzahl von Vergleichen bei einer Suche.

$$p_T = \sum_{i=1}^{n} p_i \cdot (Tiefe(i) + 1) + \sum_{j=0}^{n} q_j \cdot Tiefe'(j)$$

Wobei die erste Summe die erfolgreichen Suchen darstellt und die zweite die nicht erfolgreichen. $Tiefe(i)$ entspricht dabei der Tiefe des Knotens a_i und $Tiefe'(j)$ ist die Tiefe des Blatts (a_j, a_{j+1}).
Ein optimaler binärer Suchbaum ist nun ein binärer Suchbaum, der p_T minimiert.

Beispiel Sei $U = \mathbb{R}$ und $S = \{2, 5, 6\}$ mit $p_1 = 0,3, p_2 = 0,2, p_3 = 0,1$ und $q_0 = q_1 = q_2 = q = 3 = 0,1$. Es gibt ja nur drei mögliche Binärbäume.



Es ergeben sich dann folgende gewichtete Weglängen: $p_{T_1} = 2,3$, $p_{T_2} = 1,8$, $p_{T_3} = 1,9$. Optimal ist also T_2.

Die optimale gewichtete Weglänge ist dann maximal, wenn alle Wahrscheinlichkeiten gleich sind. Der optimale Suchbaum ist dann möglichst balanciert und es gilt $p_t = \log n + O(1)$. Sie ist minimal, wenn $p_1 = 1$ und alle übrigen Wahrscheinlichkeiten gleich 0. Dann ist $p_T = 1$. Das hat hier zu tun mit $(-\sum p_i \log p_i) + (-\sum q_j \log q_j)$. (Da steckt übrigens die Entropie drin)

Algorithmus

Als Algorithmus könnte man jetzt alle binären Suchbäume für S konstruieren, für jeden die mittlere Weglänge bestimmen und den besten herausnehmen. Es gibt etwa 4^n viele solcher binären Suchbäume, daher ist die Laufzeit also exponentiell.
Das ist uns natürlich zu langsam, wir gucken uns jetzt einen Algorithmus an, der polynomielle Laufzeit hat. Dazu machen wir ein paar Vorüberlegungen.

Betrachten wir für $i, j \in \{1, \ldots, n\}$ einen optimalen Suchbaum $T_{i,j}$ für $\{a_i, \ldots, a_j\}$ mit der Wahrscheinlichkeitsverteilung $\tilde{p_k} = \frac{p_k}{w_{i,j}}$ wobei $w_{i,j} = p_i + \ldots + p_j + q_{i-1} + \ldots + q_j$ und $k = i, \ldots, j$ das Gewicht von $T_{i,j}$ ist. Zusätzlich ist $\tilde{q_k} = \frac{q_k}{w_{i,j}}$ mit $k = i-1, \ldots, j$. Beides sind also bedingte Wahrscheinlichkeiten. Der Baum $T_{i,j}$ besteht wiederum aus dem linken Teilbaum $T_l = T_{i,m-1}$ und dem rechten Teilbaum $T_r = T_{m+1,j}$ wobei m die Wurzel von $T_{i,j}$ ist. Wie berechnet sich nun also $p_{T_{i,j}}$ aus p_{T_l} und p_{T_r}?

$$
\begin{aligned}
p_{T_{i,j}} &= \sum_{k=i}^{j} \tilde{p_k}(Tiefe(k)+1) + \sum_{k=i-1}^{j} \tilde{q_k} Tiefe'(k) \\
w_{i,j} p_{T_{i,j}} &= \sum_{k=i}^{j} p_k(Tiefe(k)+1) + \sum_{k=i-1}^{j} q_k Tiefe'(k) \\
&= \sum_{k=i}^{m-1} p_k(Tiefe(k)+1) + \sum_{k=m+1}^{j} p_k(Tiefe(k)+1) + p_m \\
&+ \sum_{k=i-1}^{m} q_k Tiefe'(k) + \sum_{k=m+1}^{j} q_k Tiefe'(k) \\
&= \sum_{k=i}^{m-1} p_k + p_m + \sum_{k=m+1}^{j} p_k + \sum_{k=i-1}^{m} q_k + \sum_{k=m+1}^{j} q_k \\
&\ldots \\
&= w_{i,j} + w_{i,m-1} \cdot p_{T_l} + w_{m+1,j} \cdot p_{T_r} \\
p_{T_{i,j}} &= 1 + \frac{w_{i,m-1} \cdot p_{T_l} + w_{m+1,j} \cdot p_{T_r}}{w_{i,j}}
\end{aligned}
$$

Anmerkung: Aufgrund von Herrn Alts vielen Tafelwischen ist die Rechnung in der Mitte sicherlich nicht korrekt und zudem unvollständig. Das letzte ist aber wieder korrekt.
Außerdem gilt: Wenn $T_{i,j}$ optimal, dann folgt T_l, T_r sind auch optimal. Als Variablen für den Algorithmus verwenden wir

$r_{i,j} :=$ Index der Wurzel von $T_{i,j}$
$c_{i,j} := w_{i,j} \cdot p_{T_{i,j}}$ Kosten von $T_{i,j}$
$w_{i,j}$ Wahrscheinlichkeit, dass $a \in [a_i, a_j]$

Es gilt

$$c_{i,j} = w_{i,j} + c_{i,m-1} + c_{m+1,j}$$
$$w_{i,j} = w_{i,m-1} + w_{m+1,j} + p_m$$

mit $m = r_{i,j}$

Algorithmus

```
1   for i= 0,...,n do
2     w(i+1,i) := q(i)
3     c(i+1,i) := 0
4   for k=0,...,n-1 do
5     for i=1,...,n-k do
6       j := i+k
7       bestimme m mit i <= m <= j, so dass c(i,m-1) + c(m+1,j)
            minimal ist
8       r(i,j) := m
9       w(i,j) := w(i,m-1) + w(m+1,j) + p(m)
10      c(i,j) := c(i,m-1) + c(m+1,j) + w(i,j)
```

Der Algorithmus folgt dem Paradigma der dynamischen Programmierung. Das heißt, von kleineren Teilproblemen ausgehend werden immer größere gelöst. Die größeren Lösungen werden dann aus kleineren zusammengebaut.
Wir machen das nun mal am Beispiel von oben. Und tragen jeweils $r_{i,j}$; $w_{i,j}$; $c_{i,j}$ ein.

$k\backslash i$	1	2	3	4
-1	$-\ ;\ 0,1\ ;\ 0$	$-\ ;\ 0,1\ ;\ 0$	$-\ ;\ 0,1\ ;\ 0$	$-\ ;\ 0,1\ ;\ 0$
0	$1\ ;\ 0,5\ ;\ 0,5$	$2\ ;\ 0,4\ ;\ 0,4$	$3\ ;\ 0,3\ ;\ 0,3$	
1	$1\ ;\ 0,8\ ;\ 1,2$	$2\ ;\ 0,6\ ;\ 0,9$		
2	$2\ ;\ 1,0\ ;\ 1,8$			

Jetzt muss man natürlich noch den Baum aus der Menge der Wurzeln rekonstruieren.

Vorlesung 11, 17. November 2008

Analyse

Speicherplatz Zunächst den Speicherplatz. für die $k = -1, \ldots, n-1$ brauchen wir also n Plätze, für $i = 1, \ldots, n-k$ ebenso.

$$\sum_{k=-1}^{n-1} \sum_{i=1}^{n-k} c = \sum_{k=-1}^{n-1} c(n-k) \sum_{k=1}^{n+1} cl = c\frac{(n+1)n}{2} = O(n^2)$$

Laufzeit Die Laufzeit in den Zeilen 1 und 2 ist natürlich $O(n)$.

$$\sum_{k=0}^{n-1} \sum_{i=1}^{n-k} (c_1 + c_2(j-i+1)) = \sum_{k=1}^{n-1} \sum_{i=1}^{n-k} (c_2 k + c_3) = \sum_{k=0}^{n-1} c_2(n-k)k + c_3(n-k)$$
$$= \sum_{k=0}^{n-1} (n-k)(c_2 k + c_3) = \sum_{k=0}^{n-1} c_2 nk - c_2 k^2 + c_3 n - c_3 k \leq O(n^3)$$

Wobei c_1 für die Zeilen 5,7,8 und 9 zuständig ist. c_2 ist für Zeile 6 zuständig.

Verbesserung der Laufzeit Wir können die Laufzeit noch verbessern. Dazu nehmen wir an, dass

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

Den Beweis hierfür führen wir nicht. Natürlich ist jetzt die Frage, wie viel man dadurch einsparen kann. Dazu guckt man sich genau Zeile 6 des Algorithmus an. Die Stücke, die in der k-ten Iteration für verschiedene Stücke überprüft werden, addieren sich insgesamt zu n. Also haben wir $O(n)$ in Zeile 6 für jedes k. Damit ist die gesamte Laufzeit nur noch $O(n^2)$.

3.3 Das Vereinige-Finde-Problem (Union-Find)

Abstrakter Datentyp

Wir haben eine unveränderliche Menge S mit oBdA $S = \{1, \ldots, n\}$. Zusätzlich haben wir eine Partition von S in disjunkte Teilmengen $S = S_1 \cup S_2 \cup \ldots \cup S_k$. Auf dem Datentyp sind folgende Operationen definiert:

- $UNION(S_i, S_j)$ ändere Partition, indem S_i mit S_j vereinigt wird
- $FINDE(k)$ liefert die Menge S_i, mit $k \in S_i$.

Meist werden die Mengen S_i durch ein Element repräsentiert, das sie enthalten.

Union-Find hat folgende Anwendungen.

1. Angenommen wir haben einen gegebener Graph $G = (V, E)$. Nun suchen wir die Zusammenhangskomponenten. Dies kann so geschehen, dass man für jede Kante $e = (u, v)$ $S_i = FINDE(u)$ und $S_j = FINDE(v)$ und dann führen wir $UNION(S_i, S_j)$ durch. Initialisiert wird das Ganze so, dass jeder Knoten sein eigens S_i bekommt.
2. minimale Spannbäume
3. In der Bildverarbeitung kann man die Segmentierung so machen.

Datenstruktur

Die übliche Datenstruktur sieht wie folgt aus. Für jede Menge S_i haben wir einen gerichteten Baum, wobei die Kinder auf ihre Väter verweisen. Die Wurzeln der jeweiligen Bäume sind dabei dann die Repräsentaten der Mengen.
Wie führen wir nun die Operationen durch?
$FINDE(a)$ liefert uns einen Verweis auf einen Knoten. Und zwar folgend wir hierbei von a ausgehende den Verweisen, bis die Wurzel gefunden wird. Die Laufzeit ist hier also $O(h)$ wobei h die Höhe des Baumes ist. Das ist natürlich tükisch, da die Höhe zwischen konstant und linear liegt.
Das $UNION(i, j)$ geht so, dass man den Baum von j einfach an den Baum von i ranhängt, in dem die Wurzel von j einen Verweis auf die Wurzel von i bekommt. Die Laufzeit ist hier also konstant.

Vorlesung 12, 21. November 2008

Damit das Finden besser geht, können wir versuchen den Baum in der Höhe zu balancieren, in dem man vor der $UNION$-Operatrion schaut, welcher Baum höher ist und den niedrigeren an den höheren ranhängt. Dazu muss natürlich immer die Höhe mitgespeichert werden.

Satz 2. *Mit Höhenbalancierung gilt, dass die Vereinigungsoperation in konstanter Zeit geht und das Finden in* $O(\log n)$.

Beweis Wir speichern die Höhe mit, dann geht die Aktualisierung beim Union in $O(1)$. Unsere Behauptung ist nun, dass jeder Baum mit k Knoten hat Höhe $\leq \log(k)$. Wir führen einen Beweis per Induktion über k.

Induktsionsanfang $k = 1$ daraus folgt $h = 0$. Ist natürlich wahr.

Induktsionsschritt Ein Baum mit $k^1, k^1 \leq 1$ Knoten hat Höhe $\leq \log k^1$. Ein Baum T mit k Knoten $T = T_1 \cup T_2$ mit k_1, k_2 die Anzahl der Knoten in T_1 bzw. T_2 und h_1 und h_2 sind die Höhen, mit $h_1 \geq h_2$. $k = k_1 + k_2$. Für die Höhe gilt dann $h = \max(h_1, h_2 + 1)$. Wir unterscheiden mehrere Fälle.

1. $h = h_1$. Daraus folgert sich dann $k = k_1 + k_2 \geq k_1 \geq 2^{h_1} = 2^h$.
2. $h = h_2 + 1$. Hier ergibt sich $h_1 = h_2$. $k = k_1 + k_2 \geq 2^{h_1} + 2^{h_2} = 2 \cdot 2^{h_2} = 2^{h_2+1} = 2^h$.

In beiden Fällen ist $k \geq 2^h$. Somit ist die Höhe hier also immer in $O(\log n)$. □

Das geht allerdings noch etwas besser, und zwar durch *Pfadkompression*.

Pfadkompression

Das Ziel der Pfadkompression ist es, eine Folge von Union- und Find-Operationen zu verbessern.
Das funktioniert so, dass man bei Finde-Operationen die Knoten auf dem Pfad aufsammelt und alle an die Wurzel hängt. Wir gucken uns nun die amortisierte Laufzeit an, dass sind die durchschnittlichen Kosten einer Operation in einer Folge.
$F, G : \mathbb{N} \to \mathbb{N}$ mit

$$F(0) = 1$$
$$F(i) = 2^{F(i-1)}$$
$$G(n) = \min\{k | F(k) \geq n\}$$

$G(n)$ wächst dabei extrem langsam und $F(n)$ ziemlich schnell.

i	$F(i)$	n	$G(n)$
0	1	1	0
1	2	2	1
2	4	$3, 4$	2
3	16	$5, \ldots, 16$	3
4	65536	$17, \ldots, 65536$	4
5	2^{65536}	$65537, \ldots, 2^{65536}$	5

Die Funktion $G(n)$ nennen wir $\log^*(n)$. Die Startsituation ist wieder $S_i = \{i\} \forall i = 1 \ldots n$. Die Folge von m $FIND$ Operationen und $n-1$ $UNION$-Operationen bezeichnen wir mit σ . Die Teilfolge σ' ist σ ohne $FIND$-Operationen. Als $Rang(v)$ definieren wir die Höhe von dem (Unter)baum T_v nach σ'. Daraus folgen ein paar Eigenschaften.

1. T_v hat $\geq 2^{Rang(v)}$ Knoten
2. Es gibt $\leq \frac{n}{2^r}$ Knoten mit Rang r
3. Alle Ränge sind $\leq \log n$
4. Falls bei σ Knoten w ein Nachkomme von v wird, folgt, dass $Rang(w) < Rang(v)$.

Für eine Ranggruppe r gilt $r \rightarrow$ Gruppe $G(r)$.
Jetzt analysieren wir die Laufzeit fürs Finden, die sich dadurch ja verbessern soll.

Die Kosten einer Finde-Operation hat als Kosten die Anzahl der Knoten auf dem Pfad. Wir betrachten jetzt einen Knoten v. Die Kosten von Knoten v weisen wir jetzt wie folgt zu:

1. Der Finde-Operation, falls v die Wurzel ist oder $G(Rang(Vater(v))) > G(Rang(v))$.
2. Knoten v sonst

Dieses Verfahren nennt man *accounting* oder auch *Buchhalter-Analyse*.

Kosten von Typ 1: Jede Finde-Operation bekommt $\leq O(G(n))$ Kosten zugewiesen.
Kosten von Typ 2: Kosten von einem Knoten $\leq$ Anzahl der Ränge in $G(Rang(v)) = g$. $Rang(v) \leq F(g) - F(g-1) \leq F(g)$. $N(g)$ sei die Anzahl der Knoten in der Ranggruppe g.

$$N(g) \leq \sum^{F(g)} \frac{n}{2^r} = \frac{n}{2^{F(g-1)+1}} \sum_{r=0} \frac{1}{2^r}$$

Gesamtkosten von Typ 2 ist $\leq n \cdot G(n)$. Die Gesamtkosten von σ setzt sich aus den Kosten für Union- und Finde-Operation zusammen. $O(m \cdot G(n) + n \cdot G(n))$. Daraus folgt der Satz

Satz 3. *Ausgehend von der $S_i = \{i\}$ wurde eine Folge σ von m Finde- und beliebig vielen Vereinigungsoperationen ausgeführt. Die Gesamtlaufzeit von σ ist dann $O(m \log^* n + n \log^* n)$*

Das ist allerdings keine scharfe Schranke für die Laufzeit. Eine noch bessere Analyse würde zu folgendem Ergebnis kommen, die wir hier aber nicht führen:

Satz 4. *Die Laufzeit von σ ist $O(m \cdot \alpha(m,n))$ mit $\alpha(m,n) = \min\{z \geq 1 | A(z, 4\left\lceil \frac{m}{n} \right\rceil > \log n\}$ wobei A eine Variante von einer Ackermann-Funktion ist.*

Die Ackermann-Funktion sieht so aus.

$$\begin{aligned} A(0,0) &= 0 \\ A(i,0) &= 1 \\ A(0,x) &= 2x \\ A(i+1,x) &= A(i, A(i-1, x-1)) \end{aligned}$$

Die Ackermann-Funktion wächst noch schneller als das obige $F(i)$.

$i \backslash x$	0	1	2	3	4	5
0	0	2	4	6	8	10
1	1	2	4	8	16	32
2	1	2	4	16	65536	2^{65536}
3	1	2	4	65536	2^{65536}	

Vorlesung 13, 24. November 2008

Vielen Dank an Gero, der mir seine Tafelfotos zu Verfügung gestellt hat.

4 Graphenalgorithmen

4.1 Definitionen

Definition 2 (Graph). *Ein Graph $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Menge $E \subset V \times V$ von Kanten.*
Bei einem gerichteten Graphen ist $E \subset \binom{V}{2}$, d.h. 2-elementige Teilmengen von V.

Definition 3 (Weg). *Ein Weg in einem Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, \ldots, v_k$ mit $(v_i, v_{i+1}) \in E$ für $i = 1, \ldots k-1$.*

Einen Weg nennen wir einfach, genau dann wenn $v_i \neq v_j$ für $i \neq j$. Die Länge des Weges ist die Anzahl der Kanten. Ein Weg heißt ein *Kreis*, genau dann wenn $v_k = v_1$. Einen Kreis nennen wir einfach, wenn außer $v_1 = v_k$ alle Knoten unterschiedlich sind, d.h. jeder andere Knoten nur einmal besucht wird. Ein Graph, der keinen Kreis enthält, ist *azyklisch*.
Ein ungerichteter Graph heißt *zusammenhängend*, genau dann wenn $\forall u, v \in V$ es ein Weg von u nach v existiert. Bei einem gerichteten Graphen spricht man dann von *stark zusammenhängend*. Eine *Zusammenhangskomponente* von G ist folglich ein maximaler Teilgraph $G' = (V', E')$ mit $V' \subset V$ und $E' \subset E$, der zusammenhängend ist. Maximal bedeutet dabei, dass keine Knoten oder Kanten hinzufügbar sind, so dass der Zusammenhang bleibt.
Der *Durchmesser* eines Graphen ergibt sich durch

$$\max_{u,v \in V} \left(\min_{w \text{ Weg von } u \text{ nach } v} \text{Länge von } w \right)$$

Darstellungsformen

Adjazenzmatrix Sei oBdA $V = \{1, \ldots, n\}$. Dann ist die Matrix A so definiert, dass $A_{i,j} = 1$, genau dann wenn $(i, j) \in E$ ist. Ansonsten enthält der Eintrag eine 0.
Der Platzbedarf liegt in $\Theta(n^2)$, mit $|V| = n$.

Adjazenzlisten Hier hat man n Listen, wobei die i-te Liste alle zum Knoten i adjazente Knoten j enthält, d.h. $(i, j) \in E$.
Der Platzbedarf liegt hier in $\Theta(m)$ mit $m = |E|$.

Inzidenzmatrix In dieser Matrix sind die Zeilen mit den Knoten und die Spalten mit den Kanten beschriftet. Hier enthält der Eintrag $I_{i,j}$ genau dann eine 1, wenn Knoten i mit Kante j inzident ist, d.h. i einer ihrer Endpunkte ist.
Der Platzbedarf liegt hier in $\Omega(nm)$.

4.2 Traversierung von Graphen

Mit der Traversierung meint man das systematische Besuchen aller Knoten, wobei die Kanten durchlaufen werden.
Im Allgemeinen unterscheidet man zwei Varianten, zum einen die *Breitensuche* und zum anderen die *Tiefensuche*. Bei der Breitensuche betrachtet man von einem Startknoten zunächst alle Nachbarn und von dort dann rekursiv weiter. Dabei ist ein Weg vom Startknoten zu einem Knoten v stets ein kürzester Weg. Bei der Tiefensuche ist man abenteuerlustig und probiert zunächst so weit wie möglich in den Graphen hinabzusteigen, und wenn es nicht mehr weitergeht rekursiv an weiter oben gelegenen Knoten vorzugehen. Die Laufzeit ist bei beidem $\Theta(n+m)$ in Adjazentlisten-Darstellung.

Topologisches Sortieren

Bei einem gerichteten azyklischen Graphen wollen wir die Knoten so auflisten, dass Knoten u erst erscheint, nachdem alle seine Vorgänger v (d.h. es existiert ein Weg von v nach u) aufgelistet wurden.
Die Vorgehensweise ist sehr einfach

```
1 solange noch nicht alle Knoten aufgelistet sind
2   wähle Knoten v mit Ingrad 0
3   Liste v auf
4   entferne v einschließlich ausgehender Kanten
```

Die Laufzeit von dem Spaß ist $\Theta(n+m)$.

Vorlesung 14, 28. November 2008

4.3 Minimal spannende Bäume

Gegeben sei ein (ungerichteter) Graph $G = (V, E)$ und eine Kostenfunktion $c : E \to \mathbb{R}$. Ein Baum, der G aufspannt, d.h. ein Teilgraph von G mit minimaler Kostensumme ist gesucht.

Lemma 2. *T sei ein Baum und e eine Kante, die nicht in T ist. Fügt man e zu T hinzu, so entsteht genau ein Kreis.*

Lemma 3. *Gegeben sei ein ungerichteter Graph $G = (V, E)$ und $c : E \to \mathbb{R}$ und ein spannender Wald $(V_1, T_1), \ldots (V_k, T_k)$. Sei $e \in E$ mit $e = (u, v)$ mit $u \in V_1$ und $v \notin V_1$ mit minimalen Kosten. Dann existiert ein spannender Baum, der e enthält und dessen Kosten minimal sind, die den ursprünglichen Wald enthalten.*

Aus diesem letzteren Lemma folgt, dass falls ein bereits spannender Wald konstruiert ist, auf dem Weg zu dem minimal spannenden Baum, so kann man die billigste Kante, die unterschiedliche Bäume des Waldes verbindet, hinzufügen und man ist immer noch auf dem Weg zum MST.

Beweis von Lemma 3 Angenommen, es sei nicht so. Sei also G^* ein spannender Baum minimaler Kosten, der T enthält, aber nicht e. Dann fügen wir e zu G^* hinzu und nach Lemma zwei entsteht dadurch genau ein Kreis, der e enthält. Zusätzlich enthält der Graph noch mind. eine weitere Kante e' zwischen V_1 und $V \backslash V_1$. Sei G^{**} der Baum der aus G^* entsteht, bei dem e' durch e ersetzt wird. Die Kosten für G^{**} sind die Kosten von G^* abzüglich der Kosten für e' plus der Kosten von e. Dadurch sind die Kosten von G^* größer als die von G^{**}. Damit ist G^{**} ein minimaler Baum, der T enthält.

Algorithmus von Kruskal Gegeben sein ein Graph $G = (V, E)$ und Kosten $c : E \to \mathbb{R}$. Der Algorithmus findet uns dann einen minimal spannenden Baum von G. T sind dabei die in den MST aufgenommenen Kanten. T ist zu beginn leer und $Q = E$, was die noch nicht betrachteten Kanten sind. $VS = \{\{V\}, \ldots \{V_k\}\}$ eine Partition der Knotenmenge.

```
1 Wiederhole |VS| > 1 do
2   e=(u,v) Kante mit minimalen Kosten aus Q.
3   entferne e aus Q
4   teste, ob sie in unterschiedlichen Mengen U,W von VS liegen
5   Falls ja:
6     vereinige U,W
7     füge e zu T hinzu
```

Für Q können wir eine Prioritätswarteschlange benutzten, z.B. ein Heap, dann können wir die Operation in logarithmischer Zeit durchführen. Für VS kann man Union-Find benutzen.

Was ist dann die Laufzeit? Die Schleife wird höchstens m mal ausgeführt, wobei m die Anzahl der Kanten ist. Eine Ausführung kostet dann $O(\log m)$. Insgesamt ist die Laufzeit also $O(m \log m)$. Maximal ergibt sich $O(n^2 \log n)$, wenn n die Anzahl der Knoten ist.

Das Prinzip des Algorithmus ist der *greedy-Ansatz* bei dem man sich gierig immer auf das billigste stürzt.

4.4 Wegeprobleme in Graphen

Gegeben sei ein gerichteter Graph $G = (V, E)$ und eine Kostenfunktion $c : E \to \mathbb{R}$
Typische Probleme sind hier kürzeste Wege zwischen zwei Knoten u, v, bzw. für einen Knoten alle kürzesten Wege (single source shortest path) oder auch generell für alle Paare von Knoten alle kürzesten Wege finden (all pairs shortest path).
Für single source gibt es den Algorithmus von Dijkstra.

Algorithmus von Dijkstra Menge S, die alle Knoten enthält zu denen kürzeste Wege gefunden wurden. Das Array $D[v]$ mit $v \in V$ enthält die Länge der kürzesten Weges von s nach v mit Zwischenknoten in S.

```
1  S := {s}
2  D[s] := 0
3  für alle Knoten v != s
4    D[v] := c(s,v), falls (s,v) in E, unendlich sonst
5
6  while V \ S != leere Menge do
7    wähle Knoten w in V \ S mit minimalen D[w]
8    S := S vereinigt {w}
9    für alle u in V \ S mit (w,u) in E
10     D[u] := min(D[u], D[w] + c(w,u))
```

Korrektheit Zur Korrektheit zeigen wir folgende Invariante. Die Behauptung ist, dass zu jeder Zeit gilt, dass zu jedem Knoten $v \in S$ gilt, dass $D[v]$ die Länge des kürzesten Weges von s nach v ist.

Beweis Wir führen einen Beweis per Induktion über k, die Anzahl der Iterationen der while-Schleife.

Induktionsanfang Durch die Initialisierung in Zeile 1-3 ist die Behauptung sichergestellt.

Induktionsschritt Sei w der Knoten, der in der $k+1$-ten Iteration zu S hinzukommt. Angenommen die Behauptung ist falsch, d.h. also, dass $D[w] > d(w)$, wobei $d(w)$ die

eigentliche Länge des kürzesten Weges von s nach w ist. Sei S_k die Menge S nach k Iterationen. Sei π der kleinste Weg von s nach w. Die Stelle, wo dieser Weg zum ersten mal die Menge S_k verlässt, bezeichnen wir als Kante (u, v). Nach Induktionsvoraussetzung, ist $d(u) = D[u]$. $d(v) = d(u) + c(u, v) = D[u] + c(u, v) \geq D[v]$ wegen der letzten Zeile des Algorithmus, als u in S eingefügt wurde. $D[v]$ ist die Länge eines Weges von s nach v. Also muss $d(v) = D[v]$ sein. Jetzt wissen wir aber, dass $D[w] > d(w) \geq d(v) = D[v]$. Das ist ein Widerspruch, denn dann hätte in Zeile 7 des Algorithmus v statt w ausgewählt werden müssen.

Vorlesung 15, 1. Dezember 2008

Dijkstra bestimmt eigentlich nur die Länge der kürzesten Wege. Wenn man es ein wenig modifiziert, werden die Wege selbst aber auch bestimmt. Dabei muss man dann sich lediglich immer die Kante merken, die man hinzunimmt. Dadurch wird dann ein Baum aufgespannt. Wenn alle Kanten gleiches Gewicht haben, entspricht Dijkstra der Breitensuche.
Die Laufzeit ist $O((m+n)\log n)$ wobei $m = |E|$ und $n = |V|$.

Kürzeste Wege zwischen allen Knotenpaaren

OBdA kann man annehmen, dass $V = \{1, \ldots, n\}$. Die Ausgabe soll eine $n \times n$ Matrix sein, wobei der Eintrag i, j die Länge des kürzesten Weges von i nach j angibt.
Der Algorithmus von Dijkstra n mal für jeden Knoten als Startknoten durchgeführt, würde diese Matrix zeilenweise füllen. Die Laufzeit dessen wäre $O((n^2 + nm)\log n) = O(n^3 \log n)$.
Für dichte Graphen, deren Kantenanzahl $m = \Theta(n^2)$ ist, geht es besser durch den Algorithmus von *Floyd-Warshall*

Algorithmus von Floyd-Warshall Sei $P_{i,j}^{(k)}$ die Menge der Wege von Knoten i nach j mit Zwischenknoten $\in \{1, \ldots, k\}$.

$$P_{i,j}^{(0)} = \begin{cases} (i,j) & \text{falls } (i,j) \in E \\ \emptyset & \text{sonst} \end{cases}$$

$$d_{i,j}^{(k)} = \min_{\pi \in P_{i,j}^{(k)}} \text{Länge von } \pi$$

Also die Länge des kürzesten Weges von i nach j mit Zwischenknoten $\in \{1, \ldots, k\}$.

$$d_{i,j}^{(0)} = \begin{cases} c(i,j) & \text{falls } (i,j) \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

Die Eingabe ist ein Graph $G = (V, E)$ und $c : E \to \mathbb{R}_{>0}$

```
1 for i = 1,...,n do
2   for j = 1,...,n do
3     d_(i,j)^(0) setzten
4 for k = 1,...,n do
5   for i = 1,...,n do
```

```
6       for j = 1,...,n do
7         d_(i,j)^(k) = min( d_(i,j)^(k-1), d_(i,k)^(k-1) + d_(k,
              j)^(k-1))
```

Korrektheit Die Initialisierung geschieht in den Zeilen 1-3. $d_{i,j}^{(0)}$ ist der kürzeste Weg ohne Zwischenknoten. Der Weg bei $i = j$ hat keine Kanten, daher ist die Länge dort 0. Andernfalls gibt es die Kante, dann ist die Länge das Gewicht der Kante, also $c(i, j)$ oder es gibt sie nicht, dann ist das Gewicht unendlich.

In Schritt 7 gibt es für den kürzesten Weg von i nach j mit Zwischenknoten $\in \{1, \ldots, k\}$ zwei Möglichkeiten.

Fall a) Er enthält Knoten k nicht. Dann ist

$$d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$$

Fall b) Er enthält k, dann gibt es einen kürzesten Weg, auf dem k nur ein Mal vorkommt.

$$d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$$

Laufzeit Die Initialisierung hat eine Laufzeit von $\Theta(n^2)$. Für den Rest brauchen wir $\Theta(n^3)$. Wir betrachten noch den Speicherplatz. Naiv bräuchte man $\Theta(n^3)$. Wenn man das sich genauer anschaut, benötigt man nur zwei Matrizen, die für $k-1$ und die gerade zu berechnende für k. So kommt man also mit $\Theta(n^2)$ Speicherplatz aus.

Der Algorithmus ist vielseitig verwendbar. Wenn man für $d_{i,j}^{(k)}$ eine Bitmatrix verwendet und keine Kantengewichte und bei der Berechnung der letzten Zeile $\min$ durch $\vee$ und $+$ durch $\wedge$ ersetzt, kann man berechnen, ob es einen Weg von i nach j gibt. Man kann auch reguläre Sprachen von endlichen Automaten in reguläre Ausdrücke überführen. In dem die Zustände die Knoten bilden und $C : E \to \Sigma$. Dann ist $d_{i,j}^{(k)}$ die Menge aller Wörter, die ohne Zustand i nach Zustand j führen und Zwischenzustände $\in \{1, \ldots, k\}$ haben. Die Initialisierung ist hier

$$d_{i,j}^{(0)} = \begin{cases} a^* & \text{falls } i = j \text{ und man kommt von } i \text{ durch } a \text{ nach } i \\ \{\epsilon\} & \text{falls } i = j \text{ und } (i,i) \notin E \\ b & \text{falls } i \neq j, (i,j) \in E \\ \emptyset & \text{sonst} \end{cases}$$

Zeile 7 sieht dann wie folgt aus

$$d_{i,j}^{(k)} = d_{i,j}^{(k-1)} \cup d_{i,k}^{(k-1)} \left(d_{k,k}^{(k-1)}\right)^* d_{k,j}^{(k-1)}$$

Vorlesung 16 und 17, 5/8. Dezember 2008

4.5 Flüsse in Netzen

Definition 4 (Netz). *Ein Netz(werk) ist ein gerichteter Graph $G = (V, E)$ und eine Funktion $c : E \to \mathbb{R}_{\geq 0}$, wobei $c(e)$ die Kapazität der Kante e angibt. Und $s, t \in V$, die als Quelle und Ziel bezeichnet werden.*

Definition 5 (Fluss). *Gegebein sei ein Netz (G, c, s, t) und sei $G = (V, E)$. Ein Fluss auf dem Netzwerk ist eine Funktion $f : V \times V \to \mathbb{R}$ mit folgenden Eigenschaften.*

1. $f(u, v) \leq c(u, v)\ \forall u, v \in V$ *wobei* $c(u, v) = 0$ *falls* $(u, v) \notin E$.
2. $f(u, v) = -f(v, u)$
3. $\sum_{v \in V} f(u, v) = 0\ \forall u \in V \backslash \{s, t\}$

Ein Beispiel für den Fluss ist die Kirchhoffsche Regel der Elektrizität (Knotenregel). Der Gesamtwert des Flusses ist definiert als $\sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$.

Das Ziel ist es nun, einen maximalen Fluss f^* zu finden, d.h. ein Fluss mit maximalem Wert. In vielen Anwendungen hat man mehrere Quellen $s_1, \ldots, s_k$ und mehrere Senken $t_1, \ldots, t_l$. Man kann diesen Fall allerdings auf den Fall mit nur einer Quelle und Senke zurückführen, in dem man einen neuen Knoten s generiert, der Kanten zu s_1 bis s_k besitzt und einen neuen Knoten t, auf den Kanten von t_1 bis t_l zeigen. Den Kapazität dieser neuen Kanten setzt man auf ∞. Wir nennen diese neue Quelle Superquelle und die Senke Supersenke.

Definition 6. *X, Y seien Mengen von Knoten. Wir definieren den Fluss von X nach Y als $f(X, Y) = \sum_{x \in X, y \in Y} f(x, y)$.*

Lemma 4. *Es gilt für jeden Fluss auf dem Netz (G, c, s, t) für alle $X, Y, Z \subset V$*

1. $f(X, X) = 0$
2. $f(X, Y) = -f(Y, X)$
3. $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ *falls* $X \cap Y = \emptyset$
4. $f(X, Y \cup Z) = f(X, Y) + f(X, Z)$ *falls* $Y \cap Z = \emptyset$

Definition 7 (Augmentierender Weg). *Eine Knotenfolge $s = v_0, v_1, \ldots, v_k = t$ mit $f(v_i, v_{i+1}) < c(v_i, v_{i+1})$ für $i = 0, \ldots, k-1$ in einem Netz nennt man augmentierender Weg.*

Aus der Existenz eines augmentierenden Weges folgt, dass der Fluss noch nicht maximal ist.

Algorithmus von Ford-Fulkerson

Die Idee ist erstmal, dass wir mit dem Nullfluss beginnen, d.h. $f(u, v) = 0 \; \forall u, v \in V$. Dann finden wir einen augmentierenden Weg und erhöhen den Fluss dann auf diesem Weg so stark wie möglich. Diesen Schritt iterieren wir dann so lange wie möglich. Danach haben wir dann den maximalen Fluss gefunden.
Um einen augmentierenden Weg zu finden, erhalten wir folgendes aufrecht: Das *Restnetz* bei gegebenen Fluss f ist (G_f, c_f, s, t) mit $G_f = (V, E_f)$. Falls man in G eine Kante hat $e = (u, v)$ mit $c(e) = b$ und $f(u, v) = a$, dann hat man in E_f zwei Kanten $e = (u, v)$ mit Kapazität $c_f(e) = b - a$ und $e' = (v, u)$ mit Kapazität

$$c_f(e') = \begin{cases} b - a & \text{falls } e' \in E \\ 0 & \text{sonst} \end{cases}$$

Etwas formaler kann man den Algorithmus so aufschreiben:

1. Initialisiere f mit 0
2. Solange ein augmentierender Weg P von s nach t im Restnetz G_f existiert, erhöhe für jede Kante in P den Fluss $c_f(P)$ (das ist die minimale Differenz $\{c(e) - f(e) | e \in P\}$).

Schritt zwei besteht aus zwei Teilen. Zum einen aus der Aktualisierung des Restnetzes und zum anderen aus dem Finden eines augmentierenden Weges. Die Frage ist nun, wie man einen augmentierenden Weg finden kann. Die Datenstruktur ist ein Graph $G' = (V, E')$ mit $E' = E \cup \{(u, v) | (u, v) \in E\}$.

Laufzeit $|V| = n$ und $|E| = m$. Für den ersten Schritt benötigen wir $O(m)$. In Schritt 2 kostet das Aktualisieren $O(m)$ und das Finden des augmentierenden Weges geht z.B. mit Tiefensuche, was auch in $O(m)$ geht. Insgesamt haben wir also eine Laufzeit von $O(m)$ pro Iteration. $O(m|f^*|)$ wobei $|f^*|$ der Wert des maximalen Flusses bei ganzzahligen Kapazitäten ist.
Bezüglich der Laufzeit kann man ein Beispiel konstruieren, wo der Algorithmus nicht terminiert.

Für ganzzahlige Kapazitäten bleibt der Algorithmus in der Menge der ganzen Zahlen und erhöht hier den Fluss in jeder Iteration um einen positiven Betrag, also um mindestens eins. Die Anzahl der Iterationen ist $\leq |f^*|$ wobei f^* der maximale Fluss ist. Wie schon erwähnt ist dann die Laufzeit $O(m|f^*|) = O(n^2|f^*|)$. Das ist nicht wirklich polynomielle Zeit, da f^* eventuell exponentiell in der Größe der Eingabe ist.

Korrektheit Gegeben ein Netzwerk (G, c, s, t) mit $G = (V, E)$ und Fluss $f : V \times V \to \mathbb{R}$. Wir definieren den Schnitt eines Netzwerks als eine Partition der Knotenmenge $V = S \cup T$ mit $s \in S$ und $t \in T$, wobei $S \cap T = \emptyset$. Sei S, T irgendein Schnitt des Netzwerks. Dann ist nach dem Lemma 4 $f(S, T) = |f|$. Und $f(S, T) = f(S, V) - F(S, S)f(S, V)$. Das ist das selbe wie $f(\{s\}, V) + f(S\backslash\{s\}, V) = f(\{s, \}, V) = |f|$

Korollar Der Wert jedes Flusses ist begrenzt durch die Kapazität von jedem Schnitt.

$\forall f \ |f| \leq c(S, T)$

Vorlesung 18, 12. Dezember 2008

Satz 5. *Sei $G = (V, E)$ ein Graph und (G, c, s, t) ein Netz und f ein Fluss im Netz. Dann sind folgende Aussagen äquivalent*

1. *f ist maximal*
2. *Das Restnetz G_f hat keine augementierenden Wege*
3. *Es existiert ein Schnitt (S, T) mit $|f| = c(S, T)$.*

Beweis

1 $\Rightarrow$ **2** Hätte das Resnetz noch einen augmentierenden Weg, so könnte man den Fluss entlang dieses Weges noch erhöhen.

2 $\Rightarrow$ **3** Wir definieren $S := \{v \in V | \exists$ eine Weg in G_f von s nach $v\}$ und $T := V \backslash S$. Dann ist $t \notin S$, da kein augmentierender Weg mehr existiert. Daraus folgt $t \in T$, d.h. (S, T) ist ein Schnitt. Für alle (u, v) mit $u \in S$ und $v \in T$ ist $f(u, v) = c(u, v)$. Damit gilt $f(S, T) = c(S, T)$.

3 $\Rightarrow$ **1** Sei f' ein beliebiger Fluss. Dann ist nach dem Korollar $|f'| \leq c(S, T) = |f|$ also $|f|$ ist maximal.

Bei Ford-Fulkerson haben wir gesehen, dass die Laufzeit nicht wirklich in polynomieller Zeit liegt, da es vom Fluss abhängt und der auch exponentiell sein kann. Besser geht das mit dem *Algorithmus von Edmonds-Karp*.

Algorithmus von Edmonds-Karp

Es funktioniert genauso wie der Ford-Fulkerson-Algorithmus, zum Finden eines augmentierenden Weges wird lediglich Breitensuche statt Tiefensuche verwendet.

Lemma 5. *Beim Algorithmus von Edmonds-Karp gilt für alle Knoten $v \in V \backslash \{s, t\}$ folgendes: Während des Ablaufs des Algorithmus ist $\delta_f(s, v)$ die Anzahl der Kanten auf dem kürzesten Weges von s nach v im Restnetz monoton wachsend.*

Beweis Angenommen es ist falsch für irgendein $v \in V \backslash \{s,t\}$. Dann existiert ein Fluss f und Fluss f' im nächsten Schritt des Algorithmus mit $\delta_{f'}(s,v) < \delta_f(s,v)$. Sei v der Knoten mit dieser Eigenschaft, wo $d_{f'}(s,v)$ minimal ist. Das heißt, für alle anderen Knoten u gilt $\delta_{f'}(s,u) < \delta_{f'}(s,v)$, dann folgt $\delta_f(s,u) \leq \delta_{f'}(s,u)$ (*). P' ist der kürzeste Weg von s nach v in $G_{f'}$. u ist der vorletzte Knoten auf P'. Mit der Eigenschaft (*) folgt $\delta_f(s,u) \leq \delta_{f'}(s,u)$ (**). Es gibt mehrere Fälle

Fall 1 $f(u,v) < c(u,v)$. Daraus folgt, dass (u,v) eine Kante in G_f ist und $\delta_f(s,v) \leq \delta_f(s,u) + 1$ weil es eine Kante von u nach v gibt. Weiterhin gilt:

$$\begin{aligned}\delta_f(s,v) &\leq \delta_f(s,u) + 1 \\ &\leq \delta_{f'}(s,u) + 1 \\ &= \delta_{f'}(s,v)\end{aligned}$$

Dies ist ein Widerspruch zur Annahme.

Fall 2 $f(u,v) = c(u,v)$. Dann ist $(u,v) \notin E_f$ aber $(u,v) \in E_{f'}$. Das heißt, der augmentierende Weg im Schritt von f nach f' muss die Kante (v,u) enthalten haben und es gilt, dass

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) - 1 \\ &\leq \delta_{f'}(s,u) - 1 \text{ nach (**)} \\ &= \delta_{f'}(s,v) - 2 \\ &< \delta_{f'}(s,v)\end{aligned}$$

Auch dies ist wieder ein Widerspruch.

□

Lemma 6. *Der Algorithmus von Edmons-Karp führt höchstens $O(nm)$ Augmentierungen durch, wobei $m = |E|$ und $n = |V|$.*

Beweis Eine Kante (u,v) heiße *kritisch* auf einem augmentierenden Weg p genau dann, wenn $c_f(u,v) = c_f(p)$, wobei $c_f(p) = \min_{e \in p} c(e)$. Das heißt (u,v) verschwindet bei einer Augmentierung aus dem Restnetz. Nun ist die Frage, wie oft eine Kante (u,v) kritisch sein kann während des Ablauf des Algorithmus? f ist der Fluss, wo (u,v) kritisch ist. Dann ist $\delta_f(s,v) = \delta_f(s,u) + 1$. f' sei der Fluss, bei dem (v,u) das nächste Mal auf einem augmentiereden Weg liegt. Dann gilt

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\geq \delta_f(s,v) + 1 \text{ nach vorhergehendem Lemma} \\ &= \delta_f(s,u) + 2\end{aligned}$$

Zwischen zwei Malen, wo (u,v) kritisch ist, muss (v,u) mindestens ein Mal auf dem augmentierenden Weg gelegen haben. Nur dadurch konnte (u,v) wieder entstehen. Der Abstand von s nach u ist im Restnetz um mindestens 2 gewachsen. Der Abstand kann höchstens $n-2$ überhaupt sein. Jede Kante ist höchstens $\frac{n-2}{2} = O(n)$ mal kritisch. Bei jeder Augmentatierung ist mind. eine Kante kritisch, wir haben also höchstens $m \cdot \frac{n-2}{2}$ Augmentierungen. □

Satz 6. *Der Algorithmus von Edmonds-Karp hat eine Laufzeit von $O(nm^2)$, nämlich $O(nm)$ Augmentierungen und pro Augmentierung $O(m)$ für die Breitensuche. Das ist das selbe wie $O(n^5)$.*

Es gibt mittlerweile schnellere Algorithmen, z.B. den von Goldmann und irgendwem, der eine Laufzeit von $O(nm \log \frac{n^2}{m})$. Ein andere hat eine Laufzeit $O(nm + n^2 \log m)$.

Vorlesung 19, 15. Dezember 2008

4.6 Bipartites Matching

Definition 8 (bipartit). *Ein ungerichteter Graph $G = (V, E)$ heißt bipartit, genau dann wenn es eine Partition $V = V_1 \cup V_2$ gibt, so dass alle Kanten einen Endpunkt in V_1 und einen in V_2 haben.*

Definition 9 (Matching). *Ein Matching in einem Graphen $G = (V, E)$ ist eine Teilmenge $M \subset E$, so dass keine zwei Kanten in M einen Endpunkt gemeinsam haben.*

Wir suchen ein größtest Matching, also eine Matching M mit $|M| \geq |M'|$ für alle Matchings M'. Ein Greedy-Ansatz funktioniert nicht. Da würde man es so machen, dass solange noch freie Kanten existieren, man diese hinzunimmt. Der Greedy-Ansatz liefert nur das *maximale* Matching, d.h. M kann nicht mehr durch zusätzliche Kanten erweitert werden und bleibt dabei ein Matching. Das ist demnach was anderes als das größte Matching!
Jedes größte Matching (maximum matching) ist maximal (maximal matching). Das gilt nicht umgekehrt.

Anwendungen Alle Probleme, wo Zuordnungen gesucht werden, finden hier ihre Anwendung. Z.B. wenn Aufgaben auf Personen aufgeteilt werden sollen.

Bipartites Matching durch Netzwerkfluss

Die Idee ist folgende: Wenn man einen bipartiten Graphen $G = (V, E)$ hat mit $V = V_1 \cup V_2$, dann fügt man noch eine Quelle s und ein Ziel t ein, so dass man einen gerichteten Graphen $G' = (V', E')$ mit $V' = V \cup \{s, t\}$ erhält, wobei $E' = \{(s, v)|v \in V_1\} \cup \{(v, t)|\ v \in V_2\} \cup \{(u, v)|\{u, v\} \in E\}$, also E enthält und zusätzlich noch gerichtete Kanten von s auf alle Knoten von V_1 und von allen Knoten aus V_2 auf t. Dann gibt man allen Kanten das Gewicht von 1 und sucht dann den maximalen Fluss.

Die Behauptung ist nun, dass es ein Matching in G mit $|M| = k$ gibt, genau dann wenn es einen ganzzahligen Fluss f in G' gibt mit $|f| = k$.

Beweis Wir zeigen Hin- und Rückrichtung.

$\Rightarrow$ Sei M das Matching und f schickt eine Einheit von s zu jedem Knoten in V_1, der zu einer Matchingkante inzident ist. Dann wird eine Einheit entlang jeder Matchingkante geschickt und dann natürlich auch noch eine Einheit zu t von jedem $v \in V_2$.

$\Leftarrow$ Sei f ein ganzzahliger Fluss der Größe k. Dann wählen wir $M = \{\{u, v\} | u \in V_1, v \in V_2, f(u, v) > 0\}$. Also $F(u, v) = 1$, da die Kapazität ganzzahlig und ≤ 1. Also muss jede andere von u ausgehende Kante Fluss 0 haben und auch jede andere nach v führende Kante Fluss 0 haben. Damit haben wir ein Matching der Größe k, da höchstens eine Kante von M zu jedem $u \in V_1$ und $v \in V_2$ inzident ist.

□

Korollar Die Kardinalität eines größten Matching in einem bipartiten Graphen G ist gleich dem Wert des maximalen Flusses des Netzwerk (G', c, s, t).

Satz 7. *Ein größtes Matching in einem bipartiten Graphen $G = (V, E)$ kann in Zeit $O(nm)$, wobei $n = |V|$ und $m = |E|$ ist, gefunden werden.*

Beweis Wir bestimmen den Netzwerkfluss in G' durch Ford-Fulkerson, was $O(m \cdot |f^*|)$ benötigt. In unserem Fall ist es $mk \leq m\frac{n}{2} = O(nm)$.

Es gibt mittlerweile bessere Algorithmen, z.B. den von Hopcroft/Kap, der eine Laufzeit von $O(\sqrt{n}m)$ hat.

Es gibt noch das gewichtete Matching, wo jede Kante ein Gewicht $\in \mathbb{R}$ hat und nun ein Matching mit maximalen Gesamtgewicht gesucht wird. Das geht in $O(nm \log n)$.

5 String-Matching

Das Problem ist folgendes. Gegeben sei ein Text $T \in \Sigma^*$ mit $|T| = n$ und ein Muster $P \in \Sigma^*$ mit $|P| = m$ und es sollen nun alle Vorkommen des Musters P im Text T gefunden werden. Also alle Zerlegungen $T = uPv$ mit $u, v \in \Sigma^*$ und alle Positionen von P.

Anwendungen gibt es viele. In der Textverarbeitung, bei Suchmaschinen, in der Bioninformatik etc.
Der naive Ansatz ist, dass man P an die erste Stelle von T anlegt und $P[1]$ mit $T[1]$ usw. vergleicht, und falls wir Übereinstimmungen haben für $i = 1 \ldots m$ dann gib 1 aus. Falls es keine Übereinstimmung gibt, dann schieben wir das Muster um einen Index weiter und vergleichen $P[i]$ mit $T[i + 1]$ usw. Die Laufzeit wäre hier durch die zwei verschachtelten Schleifen bestimmt. Die erste läuft über alle Positionen i von T und die zweite über Positionen von P. Die Laufzeit ist hier also $\Theta(nm)$ im schlechtesten Fall.

5.1 Algorithmus von Knuth-Morris-Pratt

Der Algorithmus hat eine lineare Laufzeit auch im schlechtesten Fall. P wird dabei dann um so viele Stellen weitergeschoben, bei denen sicher ist, dass es vorher kein Matching geben kann.

Definition 10 (Präfixfunktion). *Die Präfixfunktion* $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ *mit* $\pi(q) = \max\{k | k < q, P_k \sqsupset P_q\}$ *wobei* P_k *das Präfix der Länge* k *von* P *und* $\sqsupset$ *bedeutet Suffix von.*

Vorlesung 20, 5. Januar 2009

$\pi(q)$ hängt von P ab und kann durch Vorverarbeitung bestimmt werden.

Beispiel Es sei $P = ababaca$. Die Präfixfunktion ergibt sich hier dann als

q	P_q	$P_{\pi(q)}$	$\pi(q)$
1	a	ϵ	0
2	ab	ϵ	0
3	aba	a	1
4	abab	ab	2
5	ababa	aba	3
6	ababac	ϵ	0
7	ababaca	a	1

Algorithmus

Die Eingabe ist $P[1 \dots m]$ und $T[1 \dots n]$.

```
1  Berechne die Präfixfunktion von P
2  q := 0
3  for i=1 ... n do
4    while q > 0 && P[q+1] != T[i] do
5      q := Pi(q)
6    if P[q+1] = T[i] then
7      q := q+1
8    if q == m then
9      Match an Stelle i-m gefunden
10     q := Pi(q)
```

Laufzeit

Das bestimmen der Präfixfunktion geht in $O(m)$, wir zeigen das aber noch ausführlicher später. Alles nach der while-Schleife benötigt konstante Zeit pro Iteration der for-Schleife, also insgesamt $O(n)$. Interessant ist die while-Schleife: Sei $s = i - q - 1$ der Abstand vom Anfang vom P zum Anfang von T bei momentaner Stellung. In der while-Schleife wird nun bei jeder Iteration (außer der letzten) das s um mindestens eins erhöht, da q um mindestens eins größer wird. Da s immer $\leq n - m$ ist, wird die while-Schleife insgesamt höchstens

$n - m + n$ mal ausgeführt ($+n$ für letzte Iteration). Das ergibt also $O(n)$. Insgesamt ist die Laufzeit also $O(n + m)$.

Berechnung der Präfixfunktion

Wir verwenden einen Algorithmus, der dem zur Bestimmung des Stringmatchings ähnlich ist. Er sieht wie folgt aus:

```
1 Pi(1) := 0
2 for q = 2 ... m do
3   while k > 0 && P[k+1] != P[q]
4     k := Pi(k)
5   if P[k+1] = P[q] then
6     k := k+1
7   Pi(q) := k
```

Die Analyse funktioniert hier analog, die Laufzeit ist also $O(m)$.
Statt dem Knuth-Morris-Pratt verwendet man in der Praxis häufig den Algorithmus von Boyer-Moore, der zwar im schlechtesten Fall nicht linear ist, aber im Mittel dafür besser ist. Und auch nicht so kompliziert.

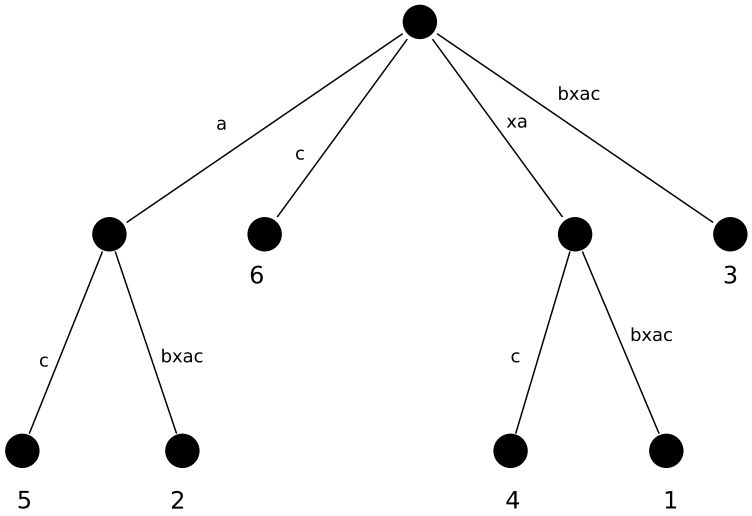
5.2 Suffixbäume

Wir haben einen Text T, der zu einer Datenstruktur vorverarbeitet werden kann. Natürlich gibt es auch wieder ein Muster P, das gesucht werden soll.

Interessant sind hier primär die Vorverarbeitungszeit, der Speicherbedarf der Datenstruktur und die Anfragezeit.

Definition 11. *Ein Suffixbaum zum Text T ist ein Baum mit einer Wurzel und n Blättern. Jeder innere Knoten außer der Wurzel hat mindestens zwei Kinder und jede Kante im Baum ist markiert mit einem nichtleeren Teilwort von T. Keine zwei Kanten, die aus einem Knoten kommen, haben das gleiche Anfangszeichen. Für ein Blatt b_i mit Nummer i gilt, dass von der Wurzel bis b_i der Weg insgesamt das Suffix $T[i \dots n]$ ergibt. Im Grunde ist es ein Trie für alle Suffizes von T.*

Beispiel T sei `xabxac`. Der Suffixbaum sieht dann wie folgt aus:



Vorverarbeitung und Speicherplatz

Zur Vorverarbeitung schaffen wir einen leeren Trie und fügen dann alle Suffizes von T nacheinander ein. Die Laufzeit hiervon ist dann $\Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$. Der Speicherplatzbedarf ist ebenfalls $\Theta(n^2)$.
Eine Variante der Suffixbäume hat eine Vorverarbeitungszeit in $O(n)$, ebenso den Platzbedarf.
Ganz genau funktioniert die Konstruktion so: Beginne mit der Wurzel, dann füge nacheinander Suffizes $T[1 \dots n], T[2 \dots n], \dots, T[n \dots n]$ in den Baum ein, und zwar wenn ein Suffix v eingefügt wird, teste ob ein Präfix von v mit einem Präfix von einer Beschriftung einer von der Wurzel ausgehenden Kante übereinstimmt, d.h. $v = v_1 v_2$. Falls ja, ändere diese Kante, so dass das übereinstimmende Präfix allein auf der Kante steht und das Restwort eine eigene Kante im Unterbaum bekommt.

Anfrage

Die Anfrage nach einem Muster P funktioniert so, dass man den Suffixbaum durchläuft und P sucht. Sie endet in einem Knoten v oder in einer Kante, die zu einem Knoten v führt (oder wir haben eine erfolglose Suche). Die Nummern aller Blätter des Teilbaums mit der Wurzel v sind die Positionen, wo P beginnt. Die Suchzeit ist $O(m + r)$, wobei r die Anzahl aller Vorkommen von P ist.

Vorlesung 21, 9. Januar 2009

Problem Ein Suffixbaum kann nicht konstruiert werden, wenn es einen Suffix gibt, der echter Präfix eines anderen Suffix ist. Z.B. bei $T = xabxab$. Das kann man aber ganz einfach beheben, in dem man am Ende seines Text T ein Terminierungssymbol anfügt, das nirgends im Text vorkommt.

Insgesamt liefert die Datenstruktur von Ukkonen einen alternativen Algorithmus für das String-Matching mit linearer Laufzeit. $O(n) + O(m + r)$.

Anwendungen von Suffixbäumen

Neben dem String-Matching finden Suffixbäume auch noch andere Anwendungen.

- Finden einer Menge $P_1, \ldots, P_l$ von Mustern in einem Text T. Die Vorverarbeitung benötigt hier natürlich auch $O(n)$, die Suchzeit ist $O(\sum_{i=1}^{l}(|P_i| + r_i))$.
- In einer Datenbank $T_1, \ldots, T_l$ von Texten sucht man ein Muster P. Das wird dann so gemacht, dass man P in $T_1\$T_2\$T_3\$\ldots\$T_l$ sucht. Die Vorverarbeitung dauert hier $O(\sum_{i=1}^{l}|T_i|)$ und das Suchen $O(m + r)$.
- Suche nach dem längsten gemeinsamen Teilwort in den Texten T_1 und T_2.
  Die Idee ist hier, dass man einen Suffixbuam für $T_1\$T_2$ konstruiert und innere Knoten mit einer 1 bzw. 2 markiert, falls ein Unterbaum mit Wurzel v einen Suffix von T_1 bzw. T_2 enthält. Dann wird der tiefste Knoten gesucht, der mit 1 und 2 markiert ist. Das ganze ist also möglich in $O(|T_1| + |T_2|)$.
- Anwendungen in der Bioinformatik. DNA-Moleküle sind folgen von vier Nukleotiden (A,T,C,G). Bei höheren Organismen besteht ein Genom aus ca. 100000 Nukleotiden. Jetzt will man die Folge bestimmen (Sequenzierung) aus verfügbaren Bruchstücken, die eine Länge von mehreren Hundert haben. Theoretisch heißt dass, bei gegebenen Wörtern $P_1, \ldots, P_l$ will man das kleinste gemeinsame Oberwort finden, das alle P_i enthält. Das Problem ist NP-schwer.

6 NP-Vollständigkeit

Alle Probleme bisher hatten Algorithmen, die in polynomieller Laufzeit liefen, also $O(n^k)$ für festes k.

6.1 Beispiele

PARTITION

Gegeben sei eine Folge $a_1, \ldots, a_n$ von ganzen Zahlen und die Frage ist nun, ob es eine Teilmenge $I \subset \{1, \ldots, n\}$ gibt, so dass $\sum_{i \in I} a_i = \sum_{j \notin I} a_j$. Dies ist ein *Entscheidungsproblem*, d.h. die Antwort ist entweder ja oder nein. Anscheinend ist dieses Problem schwer zu berechnen.
Eine Bruteforce Lösung sieht so aus, dass man für alle Teilmengen $I \subset \{1, \ldots, n\}$ prüft, ob die Bedingung gilt. Die Laufzeit ist hierbei $\Omega(2^n)$ also exponentiell. Man kennt bis jetzt keinen wesentlich besseren Algorithmus!
Dabei ist leicht zu verifizieren, d.h. effizient nachprüfbar, ob eine gegebene Lösung richtig oder falsch ist.

Vorlesung 22, 12. Januar 2009

TSP - Travelling salesperson problem

Gegeben sie ein vollständiger gerichteter Graph $G = (V, E)$, und eine Gewichtsfunktion $c : E \to \mathbb{N}_0$. Das Problem ist nun, dass eine kürzeste Rundreise in G gefunden werden soll, d.h. einen Kreis, der jeden Knoten genau einmal enthält. Dies ist kein Entscheindungsproblem, sondern ein *Optimierungsproblem*. Bei Optimierungsproblemen ist eine Lösung gesucht, die eine bestimmte Zielfunktion ("objective function") maximiert oder minimiert.
Bruteforce sieht hier so aus, dass man für alle Permutationen der Knoten die Länge der daraus resultierenden Rundreise berechnet. Über alle Längen muss dann das Minimum genommen werden. Die Laufzeit hiervon ist $\Omega(n!)$.

Aus einem Optimierungsproblem kann man ein Entscheidungsproblem definieren. Bei TSP z.B. könnte man so fragen, ob bei einem gegebenen gerichteten vollständigen Graphen $G = (V, E)$ und einem Wert $b \in \mathbb{N}$ es eine Rundreise gibt, deren Länge $\leq b$ ist.

6.2 Die Klassen P und NP

Wir betrachten hier Entscheidungsprobleme, deren Problemstellungen über einem endliche Alphabet codiert sind. Das Berechnungsmodell ist die Turingmaschine oder Registermaschine mit logarithmischem Kostenmaß. Wir wissen bereits, was im polynomieller Zeit auf einem dieser Modelle berechenbar ist, ist auch in polynomieller Zeit auf der anderen berechenbar.
Entscheidungsprobleme entsprechen formalen Sprachen. Das ist die Menge aller Problemstellungen mit Antwort ja.

Definition 12 (Komplexitätsklasse P). *Die Komplexitätsklasse P ist definiert als*

$$P = \{L | L \textit{ formale Sprache, } \exists \textit{ Algorithmus der } L \textit{ in polynomieller Zeit entscheidet}\}$$

Beispiele

- Sei $L = \{bin(n) | n \text{ durch } 3 \text{ teilbar } \}$
- Sei $L = \{w | w \text{ ist Codierung eines bipartiten Graphen mit perfektem Matching } \}$. Zur Erinnerung: Ein perfektes Matching ist bei n Knoten ein Matching der Größe $\frac{n}{2}$.

Definition 13 (Komplexitätsklasse NP). *NP ist die Klasse aller Entscheidungsprobleme L, die in polynomieller Zeit verifizierbar sind. D.h. es gibt einen Algorithmus A polynomieller Laufzeit und es gibt ein $k \in \mathbb{N}$, so dass*

$$L = \{w | \exists \text{ Wort } x \text{ mit } |x| \leq |w|^k, \text{ so dass } A(x, w) = 1\}$$

x heißt auch Zeuge für w.

Beobachtung $P \subseteq NP$. Denn wenn L in polynomieller Zeit entscheidbar ist, dann nimm $x = \epsilon$ für jede Eingabe w und $A(x, w)$ ist der Algorithmus, der x ignoriert und den Entscheidungsaltorithmus von L auf w anwendet.

Eine äquivalente Definition von NP sieht so aus, dass die Klasse die Entscheidungsprobleme enthält, die in polynomieller Zeit auf einer nichtdeterministischen Turingmaschine akzeptierbar sind.

6.3 Polynomzeit Reduktionen

Zunächst ein Beispiel. Das Problem SUBSET-SUM ist folgendes: Gegeben sind Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ und eine Zahl $b \in \mathbb{N}$. Jetzt ist die Frage, ob es eine Teilmenge I der Indizes $1, \ldots, n$ gibt, so dass $\sum_{i \in I} a_i = b$. Dieses Problem liegt in der Klasse NP.
Eine Beobachtung ist nun, sollte ein Algorithmus polynomieller Laufzeit für SUBSET-SUM existieren, so existiert auch einer für PARTITION. Denn: Eingabe $a_1, \ldots, a_n$ für Partition, dann gib $a_1, \ldots, a_n$ und $b = \frac{1}{2} \sum_{i=1}^{n} a_i$ an polynomiellen Algorithmus für SUBSET-SUM.

Definition 14 (Polynomzeit Reduktion). *$L_1, L_2 \in \Sigma^*$ seien Sprachen. L_1 heißt in polynomieller Zeit auf L_2 reduzierbar ($L_1 \leq_P L_2$) genau dann wenn eine in polynomieller Zeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ existiert, mit $w \in L_1$ genau dann wenn $f(w) \in L_2$.*

Beobachtung Ist $L_2 \in P$ und $L_1 \leq_P L_2$, dann ist auch $L_1 \in P$.

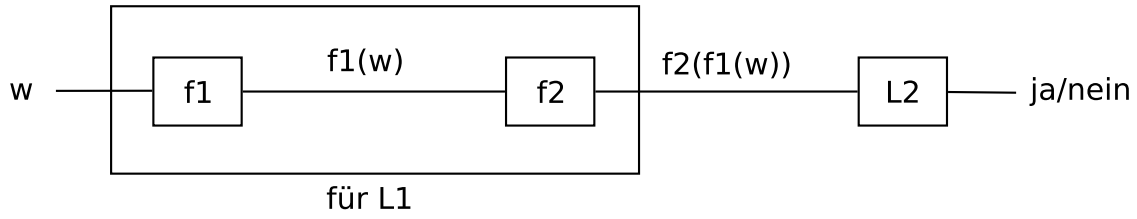
Vorlesung 23, 16. Januar 2009

Definition 15 (NP-schwer). *Eine Problem bzw. Sprache L heißt NP-schwer, genau dann wenn $L' \leq_P L$ für jedes $L' \in NP$.*

Definition 16 (NP-vollständig). *Ein Problem L heißt NP-vollständig, genau dann wenn L NP-schwer ist und $L \in NP$.*

Beobachtung Wenn es ein NP-schweres Problem L gibt mit $L \in P$, dann ist $P = NP$!

Wie zeigt man nun, dass Probleme NP-schwer sind? Dabei ist folgende Beobachtung nützlich: Falls L_1 NP-schwer und $L_1 \leq_P L_2$, dann ist auch L_2 NP-schwer, denn jedes beliebige Problem $L \in NP$ lässt sich auf L_1 reduzieren, und L_1 lässt sich auf L_2 reduzieren. Wenn $\leq_P$ transitiv ist, dann gilt das. Das kann man sich leicht veranschaulichen:



w ist dabei die Problemstellung für L.
f_1 ist in Zeit $q_1(n)$ berechenbar und f_2 in Zeit $q_2(n)$, wobei q_1, q_2 Polynome sind, dann ist $f = f_2 * f_1$ in Zeit $q = q_1(n) + q_2(n)$ berechenbar.

6.4 Schaltkreis-Erfüllbarkeit

Definition 17. *Ein Schaltkreis ist ein gerichteter azyklischer Graph, dessen Knoten von folgenden Typen sind:*

- *Und $\wedge$: Ingrad 2 und Ausgrad 1*
- *Oder $\vee$: Ingrad 2 und Ausgrad 1*
- *Negation $\neg$: Ingrad 1 und Ausgrad 1*
- *Eingang X: Ingrad 0 und Ausgrad 1*
- *Konstanten 0: Ingrad 0 und Ausgrad 1*

- *Verzweigung: Ingrad 1 und Ausgrad 2*
- *Ausgang A (gibt es nur einen): Ingrad 1 und Ausgrad 0*

Ein Schaltkreis S mit n Eingängen realisiert eine boolesche Funktion $f_S : B^n \to B$ mit $B = \{0, 1\}$.

Problem CSAT Gegeben sei ein Schaltkreis S und die Frage ist nun, ob es eine Belegung der Eingangsknoten mit Nullen und Einsen gibt, dass am Ausgang eine 1 erscheint, d.h. f_S ist nicht identisch 0.

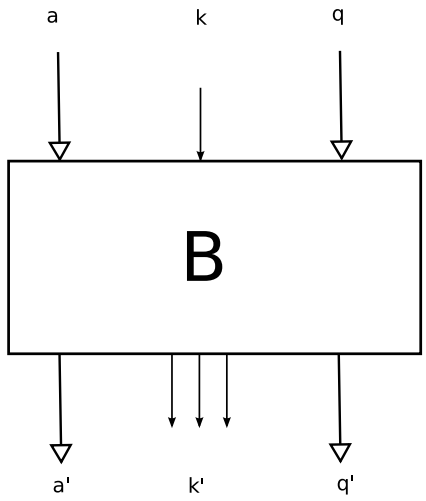
Satz 8. *CSAT ist NP-vollständig.*

Beweis

1. $CSAT \in NP$. Wir wählen als Zeugen für einen gegebenen Schaltkreis S eine Variablenbelegung $x \in \{0, 1\}^n$ mit n ist die Anzahl der Eingänge. Das Nachprüfen, ob S mit Eingabe x eine 1 liefert, ist in polynomieller Zeit möglich.
2. CSAT ist NP-schwer, d.h. Sei L ein beliebiges Problem in NP, nun ist zu zeigen, dass $L \leq_P CSAT$. A sei der zugehörige Verifikationsalorithmus, $k \in \mathbb{N}$.

   $$w \in L \Leftrightarrow \exists x, |x| \leq |w|^k \text{ und } A(x, w) = 1$$

   A ist in polynomieller Zeit auf einer 1-Band deterministischen Turingmaschine $M_A = (Q, \Sigma, \delta, q_0)$ berechenbar. Die Überführungsfunktion $\delta : Q \times \Sigma \to Q \times \Sigma \times \{R, L, 0\}$ wobei $\delta(q, a) = (q', a', R')$ heißt, wenn M_A im Zustand q das Zeichen a liest, geht es über in den Zustand q', überschreibt a mit a' und bewegt den Kopf in Richtung R'.
   Wir wollen für die Eingabe w nun einen Schaltkreis konstruieren, der erfüllbar ist, genau dann wenn $\exists x, |x| \leq |w|^k$ und M_A akzeptiert Eingabe x, w. Die Überführungsfunktion δ wollen wir in einem Baustein B konstanter Größe realisieren. Für jede Stelle i des Bandes von M_A $i = 1, 2, \ldots$ haben wir einen solchen Baustein.



Die Belegung der Ausgänge des Bausteins ist wie folg definiert:

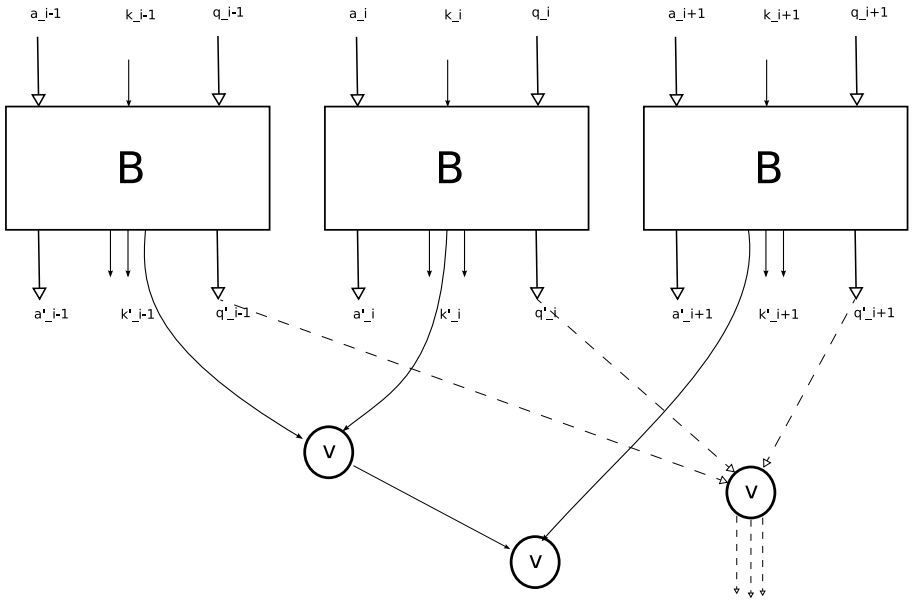
$$k_i = \begin{cases} 1 & \text{soll bedeuten, Kopf an der Stelle } i \\ 0 & \text{sonst} \end{cases}$$

$$\tilde{a_i} = \begin{cases} a_i & \text{falls } k_i = 0 \\ a_i' & \text{falls } k_i = 1 \end{cases}$$

$$\tilde{q_i} = \begin{cases} q_i' & \text{falls } k_i = 1 \\ 0 \ldots 0 & \text{sonst} \end{cases}$$

$$\tilde{k_i} = \begin{cases} 000 & \text{für } k_i = 0 \\ 100 & \text{für } k_i = 1 \text{ und } R_i' = L \\ 010 & \text{für } k_i = 1 \text{ und } R_i' = 0 \\ 001 & \text{für } k_i = 1 \text{ und } R_i' = R \end{cases}$$

Das k ist dabei ein Steuerbit.
Wir nummerieren die Zellen des Bandes von M_A von $1, \ldots, N$, wobei $N = P(n)$ und $n = |w|$; P ist ein Polynom, das die Laufzeit von M_A beschränkt. Nur diese Zellen werden von M_A benutzt. Um einen Schritt von M_A zu simulieren: N der Bausteine B nebeneinander im Schaltkreis S. Für jede Bandzelle $i = 1, \ldots, N$ von M_A und jeden Zeitschritt $t = 1, \ldots, N$ enthält der Schaltkreis S_w einen Baustein B. Es entsteht also ein $N \times N$ Gitter von Bauteilen, wobei die erste Spalte für die Zelle 1 steht, usw. und die erste Zeile für Zeitpunkt 1. Dabei packt man dann jede Zeile zu Blöcken zusammen, so dass man N Blöcke erhält.
Was wird nun in die oberste Zeile reingefüttert? Der Baustein in Spalte i hat dabei als Eingabe den Wert x_i, bis x zu ende ist. Die übrig bleibenden Eingänge gehen nicht nach außen, sondern sind fest verdrahtet und enthalten das w. Die Verknüpfung zwischen den Zeilen t zu Zeile $t + 1$ geht folgendermaßen:



Die Konstruktionszeit, um aus dem w S_w zu berechnen setzt sich folgendermaßen zusammen: Ein Baustein hat konstante Größe, d.h. ist unabhängig von der Länge von w. Es werden N^2 viele Bausteine benutzt und jeweils durch eine konstante Schaltung verdrahtet. Daraus folgt, dass die Zeit für die Konstruktion $O(N^2)$ im EKM ist. Im LKM ergibt sich dann sowas wie $O(N^2 * \log(Faktoren))$. N ist ja ein Polynom, daher ist es nach dem Quadrieren immer noch ein Polynom.
Noch zu zeigen bleibt, dass $w \in L$ genau dann, wenn $S_w \in CSAT$.

$\Rightarrow$ Wenn $w \in L$, dann existiert ein x mit $A(x, w) = 1$. Die Turingmaschine M_A liefert bei der Eingabe x, w also eine 1. Da S_w die Operationen von M_A simuliert, folgt, dass wenn x die Eingabe ist, S_w eine 1 liefert.

$\Leftarrow$ Sei S_w erfüllbar. Sei x die Eingabe, die es erfüllt. Damit liefert M_A auch 1 bei Eingabe von x, w.

Somit ist gezeigt, dass CSAT NP-vollständig ist.

Vorlesung 24, 19. Januar 2009

6.5 Weitere NP-vollständige Probleme

SAT

SAT dreht sich um die Erfüllbarkeit von aussagelogischen Formel. Wir haben also Variablen und Verknüpfungen $\wedge, \vee, \neg$ und wir nehmen an, dass die Formel in konjunktiver Normalform vorliegt.
Gegeben sei nun eine Formel α in konjunktiver Normalform und SAT fragt nun, ob α erfüllbar ist, d.h. gibt es eine Belegung der Variablen mit 0 oder 1, so dass sich α zu 1 auswertet.

Satz 9. *SAT ist NP-vollständig.*

Beweis Es ist wieder zu zeigen, dass erstens $SAT \in NP$ und zweitens SAT NP-schwer ist.

1. Als Zeuge x für die Erfüllbarkeit von α nehmen wir eine Variablenbelegung. Dann ist in polynomieller Zeit nachprüfbar, ob α erfüllt wird.

2. Wir zeigen, dass $CSAT \leq_P SAT$ gilt. Sei S die Fragestellung für CSAT, also ein Schaltkreis. Daraus ist in polynomieller Zeit eine Formel α_S konstruierbar mit $S \in CSAT \Leftrightarrow \alpha_S \in SAT$. Für jeden Knoten des Schaltkreises S konstruieren wir eine Klausel für α_S (außer Eingänge, Verzweigungen und Konstanten) und zwar hat man für jeden Knoten eine eigene Variable, ebenso für jeden Eingang. So dass für einen Knoten x und y, die z.B. durch ein $\vee$ verknüpft werden, der resultierende Knoten z benannt wird und $z \equiv x \vee y$ (meint Äquivalenz!). Das brauchen wir dann natürlich in KNF. Zusätzlich wird eine Klausel a benötigt, wobei a die Variable des Ausgangs ist, damit der Ausgang garantiert eine 1 hat.
   Es ist noch zu zeigen, dass die Formel α_S die Konjunktion aller Teilformeln erfüllbar ist, genau dann wenn S erfüllbar ist.
   $\Rightarrow$ Aus α_S ist erfüllbar, folgt, dass eine erfüllende Belegung ψ der Variablen existiert, so dass jede Teilformel erfüllt ist. Das garantiert, dass die Variablen zu den Knoten gesetzt werden entsprechend der Funktion des Knotens, also $\vee, \wedge$ oder $\neg$. Dies gilt auch für den Ausgang a. Da a auf 1 gesetzt werden muss (durch die eigene Klausel), ist also der Schaltkreis S erfüllbar mit der Eingabe, die durch die Belegung ψ der Variablen, die zu den Eingangsknoten gehören, zustande kommt.
   $\Leftarrow$ Das lassen wir hier weg, aber es ist natürlich richtig.

Vorlesung 25, 23. Januar 2009

3-SAT

Gegeben sei eine boolesche Formel α mit je drei Literalen pro Klausel. Die Frage ist nun, ob α erfüllbar ist.

Satz 10. *3-SAT ist NP-vollständig.*

Beweis Wir müssen wieder zeigen, dass $3SAT \in NP$ und dass $3SAT$ NP-schwer ist.
Sei also α eine Formel in KNF als Eingabe für SAT. Wir konstruieren in polynomieller Zeit eine Formel α_3 mit drei Literalen pro Klausel für 3-SAT, so dass α erfüllbar ist, genau dann wenn α_3 erfüllbar ist.
Wir konstruieren das wie folgt: Sei $C = x_1 \vee \ldots \vee x_n$ eine Klausel von α, wobei $x_1, \ldots, x_n$ Literale sind. Zu C konstruieren wir folgende Formel C_3, die nur drei Literale pro Klausel hat:

$$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n)$$

Die Behauptung ist nun, dass eine Belegung $\psi : \{x_1, \ldots, x_n\} \to \{0, 1\}$ C erfüllt, genau dann wenn ψ sich fortsetzen lässt zu $\psi' : \{x_1, \ldots, x_n, y_1, \ldots y_{n-3}\} \to \{0, 1\}$ das C_3 erfüllt.

$\Rightarrow$ ψ muss mindestens ein x_i auf 1 setzen. Wir setzen ψ fort zu ψ' mit

$$\psi'(y_j) = \begin{cases} 1 & \text{für } 1 \leq j \leq i-2 \\ 0 & \text{sonst} \end{cases}$$

$\psi'(y_j) = 0$ für $j = 1, \ldots, n-3$ falls $i \leq 2$

Daraus folgt, dass alle Klauseln von C_3 erfüllt sind.

$\Leftarrow$ Sei ψ' eine Belegung, die C_3 erfüllt. Zu zeigen ist, dass ψ' mindestens ein x_i auf 1 setzt. Damit erfüllt die Einschränkung ψ die Klausel C. Angenommen das ist falsch, d.h. $\psi'(x_i) = 0$ für $i = 1, \ldots, n$. Dann muss $\psi(y_{n-3}) = 0$ gelten damit letzte Klausel von C_3 erfüllt wird usw. Dadurch ist die erste Klausel nicht erfüllt, dass ist ein Widerspruch!

Damit ist α_3 die Konjunktion aller C_3' Belegung ψ' die α_3 erfüllt, genau dann wenn jedes C_3 erfüllt, genau dann wenn jedes C erfüllt (durch Einschränkung von ψ'), genau dann wenn die Einschränkung ψ' α erfüllt.

CLIQUE

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$. Die Frage ist nun, ob eine k-Clique in G existiert, d.h. eine Teilmenge $V' \subset V$ der Größe k mit $(u, v) \in E$ für alle $u, v \in V'$.

Satz 11. *CLIQUE ist NP-vollständig.*

Beweis

1. Wir zeigen, dass $CLIQUE \in NP$. Das gilt, denn als Zeuge nehmen wir einfach eine Teilmenge $V' \in V$ mit $|V'| = k$ und prüfen einfach nach, ob es eine Clique ist.

2. Um zu zeigen, dass es NP-schwer ist, führen wir eine Reduktion von $3SAT \leq_P CLIQUE$ durch. Gegeben sei eine Formel α in KNF mit drei Literalen pro Klausel. Wir konstruieren einen Graphen G_α, der eine k-Clique hat, genau dann wenn α erfüllbar ist. Dabei ist k dann die Anzahl der Klauseln von α.
   Die Konstruktion funktioniert wie folgt: $\alpha = C_1 \wedge \ldots \wedge C_k$ mit $C_i = (x_{i_1} \vee x_{i_2} \vee x_{i_3})$ als Klauseln. G_α hat $3k$ Knoten, für jedes Literal in jeder Klausel einen. Wir haben Kanten von x_{i_j} nach x_{l_m}, genau dann wenn $i \neq l$ verschiedene Klauseln sind und x_{i_j} ist nicht die Negation von x_{l_m}.
   Die Behauptung ist nun, dass G_α eine k-Clique besitzt, genau dann wenn α erfüllbar ist. Die Behauptung stimmt natürlich.

Vorlesung 26, 26. Januar 2009

Unabhängige Knotenmenge (UK)

Gegeben sie ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Die Frage ist nun, ob G eine unabhängige Knotenmenge V' der Größe k enthält, d.h. $\forall u, v \in V' : (u, v) \notin E$.

Satz 12. *UK ist NP-vollständig.*

Beweis

1. Dass $UK \in NP$ ist trivial, wir können ja uns einen Zeugen konstruieren und den leicht nachprüfen.
2. Um zu zeigen, dass UK NP-schwer ist, zeigen wir $CLIQUE \leq_P UK$. $G = (V, E)$ hat eine Clique der Größe k genau dann wenn der komplementäre Graph $G' = (V, \binom{V}{2} \backslash E)$ eine unabhänige Knotenmenge der Größe k hat.

Überdeckende Knotenmenge (ÜK, Vertex-Cover)

Gegeben sei wieder ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Die Frage ist hier, ob eine überdeckende Knotenmenge der Größe k existiert, d.h. $V' \subset V$ mit jede Kante hat mindestens eine Endpunkt in V'.

Satz 13. *ÜK ist NP-vollständig.*

Beweis

1. $UEK \in NP$ ist einleuchtend. Man kann nämlich bei einem gegebenen Zeugen das natürlich sehr schnell und einfach nachprüfen.
2. Wir zeigen, dass $UK \leq_P UEK$ um zu zeigen, dass ÜK NP-schwer ist. Angenommen wir haben einen Graphen $G = (V, E)$, der eine UK der Größe k V' hat. Genau dann hat G auch eine ÜK der Größe $n - k$, nämlich $V \backslash V'$.

SUBSET-SUM

Gegeben seien natürliche Zahlen $a_1, \ldots, a_n$ und ein $b \in \mathbb{N}$. Die Frage ist nun, $\exists I \subset \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = b$.

Satz 14. *SUBSET-SUM ist NP-vollständig.*

Beweis

1. Dass $SUBSET - SUM \in NP$ liegt, haben wir ja schon gezeigt.

2. Um zu beweisen, dass SUBSET-SUM NP-schwer ist, reduzieren wir ÜK auf SUBSET-SUM. Sei $G = (V, E)$ und $k \in \mathbb{N}$ eine Fragestellung für ÜK. Wir konstruieren daraus eine Fragestellung für SUBSET-SUM wie folgt: Wir schreiben uns für unseren Graphen die Inzidenzmatrix auf, und fügen als erste Spalte eine Spalte von Einsen hinzu. Darunter setzt man eine $|E| \times |E|$-Einheitsmatrix. Die erste, gerade hinzugefügte Einserspalte, füllen wir für diese Länge mit Nullen auf. Unter alles schreiben wir noch eine letzte Zeile, die ganz vorne k und sonst überall Zweien enthält. Die Zeilen interpretiert man nun als Viernäre Zahlen, das ist dann die Eingabe für SUBSET-SUM. Das b ist dann $b = \sum_{i=1}^{m-1} 2 * 4^i + k * 4^m$.
   Die Behauptung ist nun, dass SUBSET-SUM mit der oben konstruierten Eingabe eine Lösung hat, genau dann wenn $G = (V, E)$ eine überdeckende Knotenmenge der Größe k hat.

   $\Leftarrow$ Sei $V' = \{v_{i_1}, \ldots, v_{i_k}\}$ die überdeckende Knotenmenge. Wir wählen die Zahlen $a_{i_1}, \ldots a_{i_k}$. Die Spaltensummen ergeben bei Addition dieser Zeilen jeweils eins oder zwei, wegen der überdeckenden Knotenmenge. Wo immer eine eins herauskommt, da nehmen wir zusätzlich unten aus der Einheitsmatrix die entsprechende Zeile hinzu. Daraus folgt, dass alle Spaltensummen zwei sind, bis auf die erste Spalte, die hat natürlich Größe k. Damit ist die Gesamtsumme gleich b.

   $\Rightarrow$ Angenommen SUBSET-SUM hat eine Lösung. Dann müssen k Zeilen aus dem oberen Teil ausgewählt sein, damit an der ersten Stelle ein k steht. Für jede dieser Spalten muss in der Auswahl mindestens eine eins stehen, da nur eine 1 aus der Einheitsmatrix kommen kann und das Ergebnis 2 ist. Daraus folgt, dass die Auswahl der Zeilen dann der überdeckenden Knotenmenge von G entspricht.

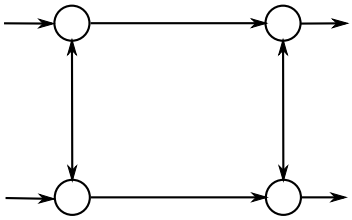
Gerichteter Hammiltonscher Kreis (GHK)

Gegeben ist ein gerichteter Graph $G = (V, E)$ und die Frage ist, ob ein gerichteter Hamiltonscher Kreis existiert, d.h. ein Kreis, der jeden Knoten genau einmal enthält.

Satz 15. *GHK ist NP-vollständig.*

1. $GHK \in NP$ ist wieder einfach. Als Zeuge nimmt man eine Folge von Knoten, die man dann einfach lang laufen kann um zu gucken, ob es eine Kreis ist, der alle Knoten enthält.

2. Für NP-schwer zeigen wir $UEK \leq_P GHK$. Sei $G = (V, E)$ ein ungerichteter Graph und k die Fragestellung für ÜK. Daraus wollen wir einen gerichteten Graphen G' machen, mit G hat eine überdeckende Knotenmenge der Größe k, genau dann wenn G' einen Hamilton Kreis der Größe k hat. Für jede Kante $(v_i, v_j) \in E$ konstruieren wir 6 Kanten mit 4 Knoten in G' wie folgt:



So ein Gebilde nenen wir dann ein Gadget.
Wir führen noch k Startknoten $s_1, \ldots, s_k$ und k Zielknoten $z_1, \ldots, z_k$ ein. Wir haben Kanten von jedem Startknoten zu jedem ersten Knoten in der Zeile und von jedem letzten aus jeder Zeile zu allen Zielknoten. Außerdem gibt es noch von jedem Zielknoten z_i Kanten nach s_{i+1} und von z_k nach s_1.
G hat eine überdeckende Knotenmenge der Größe k, genau dann wenn G' einen Hamilton Kreis hat.

$\Rightarrow$ Die Idee ist hier, dass wir den Hamilton Kreis bei s_1 beginnen und die Kanten-Gadgets der Knoten der überdeckenden Knotenmenge nacheinander abarbeiten bei Knoten $v_i \in V'$, falls $v_j \notin V'$. Ansonsten gehen wir einfach nur weiter nach rechts.

$\Leftarrow$ Stimmt auch.

Vorlesung 27, 30. Januar 2009

6.6 Komplexitätsklasse Co-NP

Definition 18 (Co-NP). *Co-NP ist die Menge aller Sprachen, deren Komplement in NP ist. Formal aufgeschrieben also*

$$Co - NP = \{L | L^C \in NP\}$$

Es gibt ein Algorithmus A mit polynomieller Laufzeit und ein $k \in \mathbb{N}$, so dass $L = \{w | \forall x$ mit $|x| \leq |w|^k$ gilt $A(x, w) = 0\}$.

Beispiel Wir schauen uns das Beispiel CLIQUE an. Gegeben sein ein Graph $G = (V, E)$ und ein $g \in \mathbb{N}$. Die Frage ist nun, ob der Graph keine Clique der Größe g enthält.
Hierbei kann man beobachten, dass es natürlich einfach ist, nachzuprüfen, ob es eine Clique gibt (also die Eingabe nicht dazu gehört), aber wie soll man verifizieren, ob eine Eingabe zur Sprache gehört?

6.7 Jenseits von NP

Unser Ziel ist die Definition und Beschreibung weiterer Komplexitätsklassen. Dazu benötigen wir noch folgendes Handwerkszeug. Die Funktion $T, S : \mathbb{N} \to \mathbb{N}$ sind wie folgt definiert:

$$TIME(T(n)) = \text{Menge aller in Zeit } T(n) \text{ entscheidbaren Sprachen}$$
$$SPACE(S(n)) = \text{Menge aller mit Platz } S(n) \text{ entscheidbaren Sprachen}$$

Wir hatten bereits folgende Klasse

$$P = \bigcup_{k \in \mathbb{N}} TIME(O(n^k)) = \bigcup_{\text{Polynom } P} TIME(p(n))$$

Definition 19 (PSPACE). *Die Komplexitätsklasse PSPACE ist die Menge aller Sprachen, die mit polynomiellem Platz entschieden werden können.*

$$PSPACE = \bigcup_{k \in \mathbb{N}} SPACE(O(n^k))$$

Beobachtung

1. Es gilt $TIME(T(n)) \subset SPACE(T(n))$. Also $P \subset PSPACE$.
2. Es gilt auch: $NP \subset PSPACE$ und $Co - NP \subset PSPACE$

Quantifizierte Boolesche Formel (QBF)

Wir gucken uns jetzt ein typisches Problem in PSPACE an. Und zwar Quantifizierte Boolesche Formeln (QBF). Gegeben ist hier also eine Quantifizierte Boolesche Formel Φ, d.h. eine Boolesche Formel der Form $Q_1 x_1 \dots Q_n x_n : \phi(x_1, \dots, x_n)$, wobei $Q_i \in \{\exists, \forall\}$. Wir wollen natürlich wieder wissen, ob eine solche Formel erfüllbar ist, oder nicht.

Satz 16. *QBF ist in PSPACE.*

Die Beweisidee sieht so aus, dass man sich überlegt, wie man die Quantoren wegbekommt. Wenn der erste Quantor ein Allquantor ist, also $\forall x_1 Q_2 x_2 \dots Q_n x_n : \phi(x_1, \dots, x_n)$, können wir den Allquantor wie folgt wegbekommen: $Q_2 x_2 \dots Q_n x_n : \phi(1, \dots, x_n) \wedge Q_2 x_2 \dots Q_n x_n : \phi(0, x2, \dots, x_n)$. Rekursiv kann man nun so für alle Quantoren vorgehen. Das $\exists$ kann man so ersetzten, dass man statt dem $\wedge$ ein $\vee$ verwendet. Der Platzbedarf bei dem Kram ist natürlich polynomiell.

Es gilt, dass $SAT \leq_P QBF$. Man bekommt so eine Eingabe für SAT zu einem für QBF, in dem man für jede Variable den Existenzquantor wählt. Man kann übrigens zeigen, dass sich jedes Problem in PSAPCE in polynomieller Zeit auf QBF reduzieren lässt. QBF ist also auch PSAPCE-schwer, also PSPACE vollständig.

GEO

Gegeben sei ein Graph $G = (V, E)$. Spieler A setzt einen Stein auf einen Knoten. Jeder Knoten darf nur einmal belegt werden. Spieler B setzt auf einen Nachfolgerknoten von A. Das geht dann immer so weiter, bis einer verliert. Verloren hat derjenige, der nicht mehr setzen kann. Es ist nun die Frage, gibt es eine Gewinnstrategie für A, bei der A immer gewinnt?

Es kann gezeigt werden, dass $QNF \leq_P GEO$ und $GEO \in PSPACE$. GEO ist also PSPACE vollständig.

6.8 Jenseits von PSPACE

Definition 20 (EXPTIME). *Die Komplexitätsklasse EXPTIME ist die Menge aller Sprachen, die deterministisch in exponentieller Zeit entschieden werden könnnen.*

$$EXPTIME = \bigcup_{\text{Konstanten } c, \text{Polynome } p} TIME(c^{p(n)})$$

Unsere Ziele sind:

1. $PSPACE \subset EXPTIME$
2. $P \subset EXPTIME$ (diesmal wirklich eine echte Teilmenge!)

Satz 17. *Ist ein Problem lösbar mit $S(n)$ Platz, so auch in $c^{S(n)}$ Zeit, für eine konstante c.*

Daraus folgt dann $SPACE(S(n)) \subset \bigcup_{c\in\mathbb{N}} TIME((c^{S(n)})$. Und somit ist $PSPACE \subset EXPTIME$.

Hierarchiesätze

1. Ist $S_1 = o(S_2)$, so ist $SPACE(S_1)$ echte Teilmenge von $SPACE(S_2)$
2. Ist $T_1 \log T_1 = o(T_2)$, so ist $TIME(T_1)$ echte Teilmenge von $TIME(T_2)$.

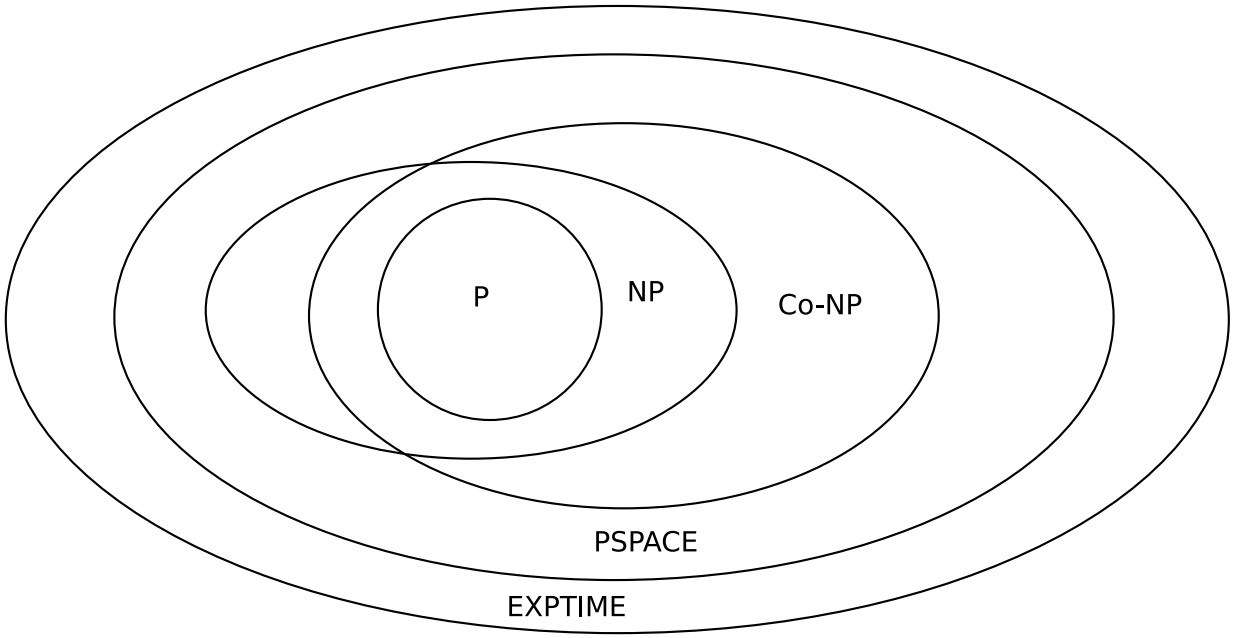
Die Situation ist folgende

$$P \subset NP \subset PSPACE \subset EXPTIME$$

Bei all diesen, wissen wir nicht, ob sie echte Teilmengen sind oder nicht. Aber dadurch, dass P eine echte Teilmenge von $EXPTIME$ ist, muss irgendwo auf jedenfall eine echte Teilmenge sein.

Nachweisbar schwere Probleme

Alles, was wir bis jetzt angeguckt haben, könnte noch irgendwie in P liegen. Jetzt schauen wir uns Probleme an, die bewiesenermaßen nicht in P liegen. Eines ist die Äquivalenz Regulärer Ausdrücke (ÄRA)



Vorlesung 28, 2. Februar 2009

7 Approximationsalgorithmen

Man hat verschiedene Möglichkeiten.

- Heuristiken helfen das Problem einzuschränken. In der Praxis will man häufig z.B. nur Teilmengen der Problematik betrachten, so dass Eingaben, die eine schlechte Laufzeit haben, nicht betrachtet werden müssen, da sie nicht vorkommen können.
- Approximationsalgorithmen. Man hat hier einen polynomiellen Algorithmus, der nicht immer die optimale Lösung findet, sondern nun etwas ähnliches. Was das heißt, gucken wir uns noch an.

Bei der Klasse NP hatten wir uns ja nur Entscheidungsprobleme angeguckt, die nur die Antwortmöglichkeit "ja" oder "nein" zulassen. Bei TSP wollten wir ja so wissen, ob bei einem Graphen G und einer Zahl k es eine Rundreise $\leq k$ gibt.
Beim Approximieren wollen wir uns nun aber die Optimierungsprobleme angucken, d.h. dass für einen Graphen G die kleinste Rundreise gefunden werden soll.

Anmerkung Gibt es für ein Optimierungsproblem einen Algorithmus mit polynomieller Laufzeit, dann auch für das zugehörige Entscheidungsproblem.

Definition 21. *Ein Optimierungsproblem heißt NP-schwer, wenn das zugehörige Entscheidungsproblem NP-schwer ist.*

Man sollte beachten, dass Optimierungsprobleme per Definition nicht in NP liegen, und man sich so immer das zugehörige Entscheidungsproblem angucken muss, um irgendwelche Aussagen treffen zu können.

Definition 22. *Sei P ein Optimierungsproblem mit Kostenfunktion c. Sei I eine Eingabe für P. Eine Lösung $f_{Opt}(I)$ für das Optimierungsproblem heißt optimal, falls*

$$c(f_{Opt}(I)) \leq c(f(I))$$

für jede andere Lösung, falls P ein Minimierungsproblem ist bzw.

$$c(f_{Opt}(I) \geq c(f(I))$$

falls P ein Maximierungsproblem ist.

Definition 23. *Ein Algorithmus A heißt α-approximativ für ein Optimierungsproblem P, genau dann wenn er eine Lösung f_A für P liefert mit der Eigenschaft, dass*

$c(f_A(I)) \leq \alpha * c(f_{Opt}(I))$ *Minimierungsproblem*
$c(f_A(I)) \geq \alpha * c(f_{Opt}(I))$ *Maximierungsproblem*

Beispiel Wir gucken uns VERTEXCOVER (ÜK) an. Das Problem ist bekanntermaßen NP-vollständig. Das Optimierungsproblem, welches wir betrachten wollen, sieht so aus: Gegeben sei ein ungerichteter Graph $G = (V, E)$ und wir wollen die überdeckende Knotenmenge mit minimaler Elementanzahl finden. Der Algorithmus sieht wie folgt aus. Die Eingabe ist $G = (V, E)$.

1. $U = \emptyset$
2. Nimm beliebige Kante $e = \{u, v\}$ und füge u und v zu U hinzu.
3. Entferne alle Kanten, die zu u oder v inzident sind.
4. Wiederhole von 2) an so lange, bis $E = \emptyset$.

Dieser Greedy-Algorithmus hat eine Laufzeit von $O(|E| + |V|)$.

Satz 18. *Der Greedy-Algorithmus für VERTEXCOVER ist 2-approximativ.*

Beweis $f_A(G) = U$, die durch den Algorithmus konstruierte Knotenmenge. $f_{Opt}(G) = U_{Opt}$ ist die optimale Knotenmenge, d.h. mit minimaler Anzahl von Knoten. Daraus folgt $c(f_A(G)) = |U|$ und $c(f_{Opt}(G)) = |U_{Opt}|$. Wir müssen nun zeigen, dass $|U| = c(f_A(G)) = 2c(f_{Opt}(G)) = 2|U_{Opt}|$ gilt.
U ist eine überdeckende Knotenmenge, da nur Kanten entfernt werden, die mindestens zu einem Element von U inzident sind, also von U überdeckt werden. Am Ende vom Algorithmus sind laut Definition alle Kanten entfernt und somit von U überdeckt. Jede überdeckende Knotenmenge, insbesondere auch U_{Opt}, muss mindestens einen der Knoten enthalten, die in Schritt 2) ausgewählt werden. Sei p die Anzahl der Schleifendurchläufe. Das entspricht genau der Anzahl der ausgewählten Kanten. Dann ist $p \leq |U|$ für jede überdeckende Knotenmenge. Also auch $p \leq |U_{Opt}|$. $|U| = 2p$. Daraus folgt, dass $|U| = 2p \leq 2|U_{Opt}|$.

7.1 Approximationsalgorithmen für Δ-TSP

Bisher haben wir uns TSP auf vollständigen, gerichteten Graphen betrachtet. Die Approximationsalgorithmen funktionieren für diese Graphe allerdings nicht. Daher betrachten wir ab jetzt immer vollständige, ungerichtete Graphen mit der folgenden Einschränkung, dass die Kostenfunktion die Dreiecksungleichung erfüllen muss.
Die Eingabe für dieses Optimierungsproblem Δ-TSP sieht dann so aus, dass man eine $n \times n$ Kostenmatrix D hat mit $n = |V|$. D erfüllt die Dreiecksungleichung, d.h. $a_1 + a_2 > a_3$ für ein Dreieck.

Satz 19. *Δ-TSP ist NP-vollständig.*

Beweis Um zu zeigen, dass es in NP liegt, wählen wir als Zeugen eine Folge von Knoten. Wir testen dann, ob alle Knoten genau einmal vorkommen und ob die Summe aller Gewichte kleiner k ist.
Für NP-schwer reduzieren wir das ungerichtete Hamiltonpfad-Problem auf Δ-TSP. $F(G) = (D, n+1)$ mit

$$d_{i,j} = \begin{cases} 1 & \text{falls } \{i,j\} \in E \\ 2 & \text{sonst} \end{cases}$$

G hat einen Hamiltonpfad, daraus folgt $f(G)$ hat eine Tour maximal der Größe $n+1$. Daraus folgt, dass die Knoten $v_1, \ldots, v_n$ jeder genau einmal besucht werden. Dann ist $\sum_{i=1}^{n-1} a_{i,i+1} + a_{n,1} = (n-1) + a_{n,1} \leq n+1$ Daraus folgt, dass $f(G) \in \Delta$-TSP.
Für $f(G) \in \Delta$-TSP folgt, dass eine Rundreise mit Kosten $\leq n+1$ existiert, sei diese $v_1, \ldots, v_n$. Dann ist $\sum_{i=1}^{n-1} a_{i,i+1} + a_{n,1} \leq n+1$, denn die Rundreise hat n Kanten, höchstens eine kann das Gewicht > 1 haben. Falls es eine Kante mit Gewicht 2 gibt, entferne diese, das ist dann ein Hamiltonpfad in G, auf dem alle Kanten Gewicht 1 haben. Ansonsten gibt es ein Hamiltonkreis in G.

Vorbereitung Ein Multigraph $G = (V, E)$ ist ein Graph mit Knotenmenge V und Kantenmenge E, wobei zwischen zwei Knoten mehrere Kanten existieren können. G heißt Eulersch, genau dann wenn es einen geschlossenen Weg gibt, der jeden Knoten mindestens einmal und jede Kante genau einmal durchläuft.

Satz 20. *Ein Multigraph $G = (V, E)$ ist Eulersch, genau dann wenn*

1. *G ist zusammenhängend.*
2. *Jeder Knoten hat geraden Grad.*

Baumheuristik für Δ-TSP

Die Eingabe ist ein Graph G, gegeben durch seine Abstandsmatrix $D = (d_{i,j})$, die die Dreiecksungleichung erfüllt.
Die Baumheuristik funktioniert dann wie folgt:

1. Berechne den minimalen Spannbaum T des Graphen G bzgl. D.
2. Verdopple jede Kante in T. Wir erhalten dann einen Eulerschen Multigraphen M.
3. Bestimme in M einen Eulerschen Weg und konstruiere daraus eine Rundreise τ in G gemäß des folgenden Lemmas.

Lemma 7. *Sei M ein Eulerscher Multigraph auf der Knotenmenge $V = \{1, \ldots, n\}$ mit Kosten $c(M) = \sum_{\{i,j\} \in E} d_{i,j}$. Dann lässt sich eine Rundreise τ auf V in $O(|E|)$ Zeit finden, mit $c(\tau) \leq c(M)$.*

Beweis Man betrachte den Eulerschen Weg $w = i_1\alpha_1 i_2\alpha_2 \dots i_n\alpha_n$ in M. Die α_i sind dabei die Knoten, die schon einmal auf dem Weg besucht wurden und die i_i die Knoten, die zum ersten Mal in w vorkommen, die Menge aller i_i ist natürlich genau die gesamte Knotenmenge. Ein solcher Weg kann natürlich in polynomieller Zeit gefunden werden.
Die Rundreise $\tau = (i_1, \dots, i_n)$ hat die Länge $c(\tau) = d_{i_1,i_2} + d_{i_2,i_3} + \dots + d_{i_{n-1},i_n} + d_{i_n,i_1}$. Aus der Dreiecksungleichung gilt $c(\alpha_j) \geq d_{j,j+1}$ wobei $j \in \{1, \dots, n\}$ ist. Damit ist $c(\tau) \leq c(w) = c(M)$, da wir ja jede Kante besuchen.

Vorlesung 29, 6. Februar 2009

Satz 21. *Die Baumheuristik ist ein 2-approximativer Algorithmus für Δ-TSP.*

Beweis Wir müssen zeigen, dass $c(\tau) \leq 2c(\tau_{Opt})$. Der zu τ_{Opt} gehörige Graph enthält einen aufspannenden Baum $\tilde{T}$. Daraus folgt, dass $c(\tau_{Opt}) \geq c(\tilde{T}) \geq c(T)$ wobei T der MST von G ist. Wegen Schritt 2) des Algorithmus wissen wir $2c(T) = c(M)$. Nach dem Lemma gilt $c(\tau) \leq c(M)$. Setzt man dies alles zusammen, erhält man also $c(\tau) \leq c(M) \leq 2c(T) \leq 2c(\tau_{Opt})$.

Die Baumheuristik ist übrigens nicht α-approximativ für $\alpha < 2$.
Das es 2-approximativ ist, kommt letztendlich daher, dass wir einfach alle Kanten verdoppeln. Die Frage ist nun, ob man nicht auch durch weniger Kanten den Graphen Eulersch hinbekommen kann. Natürlich geht das, und zwar mit der Christofides Heuristik.

Christofides Heuristik

Die Idee ist hier, dass wir die Eulerschen Multigraphen nicht durch Kantenverdopplung erzeugen, sondern durch Hizufügen von Kanten, die Knoten mit ungeraden Grad verbinden.

Zur Erinnerung:

- In jedem Graphen ist die Anzahl der Knoten mit ungeradem Grad gerade.
- Matching. Bei einen Graphen $G = (V, E)$ ist $P \subset E$ ein Matching, wenn alle Kanten in P paarweise disjunkt sind. Ein Matching heißt perfekt, wenn $|P| = \frac{|V|}{2}$. D.h. alle Knoten sind zu genau einer Kante aus P inzident. Auf vollständigen Graphen mit gerader Anzahl von Knoten existiert stets ein perfektes Matching. Bei gewichteten Graphen kann man dann nach dem perfekten Matching mit dem minimalen Gewicht fragen.

Für die Christofides-Heuristik betrachten wir auf der Menge U der Kanten ungeraden Grads in T den von G induzierten vollständigen Graphen auf U. Wir erzeugen einen Eulerschen Multigraphen aus T und einem perfekten Matching minimalen Gewichts auf U. Dieser Multigraph ist Eulersch, denn wir fügen den Knoten mit ungeradem Grad aus T jeweils eine neue Matching-Kante hinzu, d.h. ihr Grad ist dann gerade.

Algorithmus Die Eingabe ist wieder G gegeben durch die Abstandsmatrix $D = (d_{i,j})$, die die Dreiecksungleichung erfüllt.

1. Berechne den MST T von G bzgl. D.
2. Sei U die Menge der Knoten ungeraden Grades in T. Dann berechne das pefekte Matching P minimalen Gewichts auf U
3. Betrachte den eulerschen Multigraphen M, der aus den Kanten aus T und P besteht und berechnen darauf einen Eulerweg und die Rundreise τ gemäß des Lemmas.

Dei Laufzeit von dem Algorithmus ist polynomiell.

Satz 22. *Die Christofides-Heuristik ist ein $\frac{3}{2}$-approximativer Algorithmus für Δ-TSP.*

Beweis Zu zeigen ist $c(\tau) \leq \frac{3}{2}c(\tau_{Opt})$. Wieder gilt $c(\tau_{Opt}) \geq c(T)$. Dann gilt noch $c(M) = c(T) + c(P)$ und $c(\tau) \leq c(M)$. Es gilt also $c(\tau) \leq c(M) = c(T) + c(P) \leq c(\tau_{Opt}) + c(P)$. Wir müssen nun also noch zeigen, dass die Kosten für das Matching die Hälfte der Kosten für die Tour sind. Also $c(P) \leq \frac{1}{2}c(\tau_{Opt})$. Wir betrachten die optimale Tour τ_{Opt}. Wir nummerieren die Knoten $U = \{i_1, i_2, \ldots, i_{2m}\}$, so dass die Knoten aus U in dieser Reihenfolge in der Tour τ_{Opt} auftreten.
Man kann jetzt perfekte Matchings auf U finden. Zum einen P_1, wenn man Kanten zwischen $\{i_1, i_2\}$, $\{i_3, i_4\}$ usw. legt bis $\{i_{2m-1}, i_{2m}\}$ bzw. bei P_2 zwischen $\{i_2, i_3\}$, $\{i_4, i_5\}$ usw. bis $\{i_{2m}, i_1\}$.
Wegen der Dreiecksungleichung gilt $c(\tau_{Opt}) \geq c(P_1) + c(P_2)$.Da P das minimale perfekte Matching bzgl. des Gewichts ist, gilt $c(\tau_{Opt}) \geq c(P) + c(P) \Rightarrow c(P) \leq \frac{1}{2}c(\tau_{Opt})$. Und das wollten wir ja zeigen.

7.2 Nichtapproximierbarkeit des allgemeinen TSP

Satz 23. *Für kein $\alpha > 1$ existiert ein α-approximativer Algorithmus polynomieller Laufzeit für TSP, es sei denn $P = NP$.*

Beweis Angenommen, für $\alpha > 1$ existiert ein solcher Algorithmus A_α. Wir zeigen, dass dann das NP-vollständige Hamiltonkreis-Problem in P liegt.
Bei dem Problem HK ist ein Graph $G = (V, E)$ gegeben und wir fragen nach der Existenz einen Hamilton Kreises in G.
Ein (fiktiver) Polynomialzeitalgoritmus für HK sieht so aus:

1. Konstruiere eine Abstandsmatrix D für G wie folgt

$$d_{i,j} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 2 + (\alpha - 1)|V| & \text{sonst} \end{cases}$$

2. Berechne Tour τ für G bezüglich D mit A_α

3. Die Ausgabe ist 1, falls $c(\tau) = n$ und 0 sonst.

Die Behauptung ist nun, dass die Ausgabe 1 ist, genau dann wenn $G \in HK$.

$\Rightarrow$ Aus $c(\tau) = n$ folgt, dass alle Kanten von τ das Gewicht 1 haben. Das bedeutet dann, dass alle Kanten von τ in E liegen, also ist τ ein Hamilton Kreis in G.

$\Leftarrow$ Hat G einen Hamiltonschen Kreis, so gilt $c(\tau_{Opt}) = n$. Daraus folgt, dass A_α eine Tour τ berechnet mit $c(\tau) \leq \alpha n$.
Wir zeigen nun, dass alle Kanten von τ das Gewicht 1 haben, also τ ist ein Hamtiltonkreis in G, d.h. die Ausgabe ist 1.
Angenommen nicht, dann hat eine Kante von τ das Gewicht $2 + (\alpha - 1)n$. Daraus folgt, dass $c(\tau) \geq 2 + (\alpha - 1)n + (n - 1) * 1 = 2 - n + \alpha n + n - 1 = \alpha n + 1 > \alpha n$.
Das ist ein Widerspruch!

Man kann Optimierungsprobleme danach klassifizieren wie ihre Approximierbarkeit ist.

- Nicht α-approximierbar
- Mit konstantem $\alpha > 1$ approximierbar
- Für alle $\alpha > 1$ approximierbar
- Es gibt ein PTAS
- Es gibt ein FPTAS

Vorlesung 30, 9. Februar 2009

Definition 24. *PTAS (polynomial time approx scheme) ist ein Algorithmus für ein Optimierungsproblem, der bei Eingabe (Problemstellung, ϵ) mit $\epsilon > 0$ eine Approximation zur optimalen Lösung liefert mit Faktor $1 + \epsilon$ (für Minimierungsprobleme) bzw. $1 - \epsilon$ (für Maximierungsprobleme). Die Laufzeit ist polynomiell in der Größe der Problemstellung für jede Konstante ϵ.*

7.3 PTAS für SUBSET-SUM

Als Optimierungsproblem formulieren wir das so: Gegeben sei eine Folge $c_1, \ldots, c_n$ und ein $K \in \mathbb{N}$. Die Fragestellung ist hier dann so, dass man eine Teilmenge $I \subset \{1, \ldots, n\}$ findet, mit $\sum_{i \in I} c_i \leq K$ und $\sum_{i \in I} c_i$ ist maximal.

Algorithmus zur exakten Lösung

```
ExSUB(c_1, ..., c_n, K) {
  L_0 := (0)      // Liste
  for i=1 to n do
    L_i := MISCHE(L_{i-1},L_{i-1}+c_i)
    Entferne alle Elemente > K aus L_i
  Gib das größte Element von L_n aus
}
```

Auch wenn es nicht so aussieht, aber die Laufzeit ist exponentiell, weil die Länge der Listen exponentiell steigen kann.

Eine Verbesserung geht so, dass man jede Liste L_i trimmt nach dem sie konstruiert wurde. Das geht so, dass man einen Parameter δ hat und so viele Elemente wie möglich entfernt, so dass in der neuen Liste L_i' für jedes entfernte $y \in L$ ein z existiert mit $(1 - \delta)y \leq z \leq y$. Die zugehörige Prozedur $TRIM(L, \delta)$ macht folgendes: Seien $L = (y_1, \ldots, y_n)$ mit $y_1 \leq y_2 \leq \ldots \leq y_m$.

```
TRIM(L,d) {
  L' = (y_1)
  for i=2 to m do
    if letztes Element von L' < (1-d)y_i then
      hänge y_i an L' an
```

```
}
```

Mit diesem Trimmen können wir nun also einen approximativen Algorithmus bauen.

```
ApSUB(c_1, ..., c_n, K,epsilon) {
  L_0 := (0)
  for i=1 to n do
    L_i := MISCHE(L_{i-1}, L_{i-1} + c_i)
    L_i := TRIM(L_i, epsilon / n)
    Entferne alle Elemente > K aus L_i
  Gib das größte Element von L_n aus
}
```

Analyse Wir wollen den Approximationsfaktor und die Laufzeit analysieren.
Zunächst den Approximationsfaktor. Sei $P_i = \{\sum_{j\in S} c_j | S \subset \{1,\ldots,n\} \wedge \sum_{i\in I} c_i \leq K\}$. Für jedes L_i gilt weiterhin, dass alle seine Elemente in P_i liegen. Also wird bei der Ausgabe eine Teilmengensumme $z \in P_n$ ausgegeben. Die Behauptung ist nun, dass $z \geq (1-\epsilon)S_{max}$ wobei S_{max} die optimale Lösung ist.
Das Trimmen in ApSUB bewirkt folgendes: Zu jedem Element s der ursprünglichen Liste L_i existiert ein Element s' der neuen Liste L_i mit $s \geq s' \geq (1-\frac{\epsilon}{n})s$. Durch Induktion über i folgt unmittelbar, dass $\forall s \in P_i \exists s' \in L_i$ mit $s \leq s' \leq (1-\frac{\epsilon}{n})^i$. Insbesondere gilt $s_{max} \geq z \geq (1-\frac{\epsilon}{n})^n s_{max}$. Es ist $(1-\frac{\epsilon}{n})^n \geq 1-\epsilon$ für alle $n \geq 1$ und $\epsilon \leq 1$.
Wir betrachten die Funktion $f(x) = (1-\frac{\epsilon}{n})^n$. Sie ist strikt monoton wachsend für $x \geq 1$. Mit $f(1) = 1-\epsilon$ also $f(x) \geq 1-\epsilon\ \forall x \geq 1$. Also ist $z \geq (1-\epsilon)s_{max}$.

Jetzt zur Laufzeit. Wie lang kann nun L_i beim Trimmen werden? Für aufeinanderfolgende t, s in L_i gilt, dass $\frac{s}{t} \geq \frac{1}{1-\frac{\epsilon}{n}}$. Sonst wäre s nicht mit in die Liste aufgenommen worden. $0 \leq s_1 \leq s_2 \leq \ldots \leq s_m$ sind die Elemente von L_i. Alle Elemente sind natürlich $\leq K$, es folgt die Ungleichung

$$\begin{aligned} \frac{1}{(1-\frac{\epsilon}{n})^n} &\leq K \\ -(m-1)\ln(1-\frac{\epsilon}{n}) &\leq \ln K \\ m &\leq \frac{\ln K}{-\ln(1-\frac{\epsilon}{n})} + 1 \end{aligned}$$

Für $x > 0$ gilt $\ln x \leq x - 1$, also

$$m \leq \frac{\ln Kn}{\epsilon} + 1$$

Also benötigt eine Ausführung des Mischens und des Trimmens $O(m) = O(\frac{\ln Kn}{\epsilon})$. Insgesamt ist die Laufzeit $O(n^2 \ln K)$. Das ist eigentlich im EKM, aber im LKM ist es auch polynomiell.

Satz 24. *Der Algorithmus ApSUB ist ein PTAS für das SUBSET-SUM-Problem.*