7. Übung zur Vorlesung

Algorithmentechnik

am

12.02.2008

Organisatorisches zur Klausur

- Anmeldung möglich bis Freitag, 13.02.2009 (das ist **morgen**!)
- Abmeldung möglich bis Freitag, 13.02.2009, und direkt vor der Klausur im Hörsaal
- Die Hörsaaleinteilung wird vermutlich ab Montag, 16. Februar auf der Algotech-Homepage stehen, sowie vor Raum 332 im Informatik Hauptgebäude aushängen
- Wichtig: Wer nicht auf dieser Liste steht, darf die Klausur nicht mitschreiben, auch nicht unter Vorbehalt.
- Klausurdauer: 1 Stunde

Inhaltliches zur Klausur

- Hilfsmittel sind nicht erlaubt, auch keine Wörterbücher
- Es gibt keine Stoffausschlüsse, relevant ist der komplette, in Vorlesung und Übung behandelte Stoff
- Klausurvorbereitung
  - Alte Klausuren
  - Skript
  - Übungsfolien & Lösungshinweise

Möglicherweise unklare Bezeichnungen

- Kardinalität vs. Gesamtgewicht einer gewichteten Menge M
  - Kardinalität: Anzahl der Elemente von M
  - Gesamtgewicht: Summe der Einzelgewichte der Elemente

- ϵ-approximierender Algorithmus
  - Konvention I: $\frac{apx(I)}{opt(I)} \leq 1 + \epsilon$
  - Konvention II: $\frac{apx(I)}{opt(I)} \leq \epsilon$
  - In der Klausur wird klar sein, welche Formulierung gemeint ist

- ILP-Formulierungen
  - verschiedene Standardformen existieren
  - Beispiel 1: $\min c^T x$ auf $\{x \mid Ax \leq b, x \geq 0\}$
  - Beispiel 2: $\min c^T x$ auf $\{x \mid Ax = b\}$
  - In der Klausur ist keine spezielle Standardform verlangt

Algorithmus 1 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

1 Berechne eine zufällige Permutation $p_1, \ldots, p_n$ der Punkte in S
2 B $\leftarrow$ bounding box von $\{p_1, \ldots, p_4\}$
3 **if** *alle* p_i *sind in* B **then**
4   return B
5 **else**
6   return RandomizedBoundingBox(S)

Aufgabe 1 - Bounding Box

Die *bounding box* einer Menge von Punkten in der Ebene ist das kleinste achsenparallele Rechteck, das alle Punkte der Menge beinhaltet.

Algorithmus 2 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

1 Berechne eine zufällige Permutation $p_1, \ldots, p_n$ der Punkte in S
2 B $\leftarrow$ bounding box von $\{p_1, \ldots, p_4\}$
3 **if** *alle* p_i *sind in* B **then**
4   return B
5 **else**
6   return RandomizedBoundingBox(S)

- Wir nehmen vereinfachend an, dass genau 4 Punkte auf dem Rand der bounding box von S liegen.
- Berechne die erwartete Laufzeit des Algorithmus.
- Schildere dazu eine einfache Vorgehensweise, die überprüft, ob ein Punkt in der bounding box liegt.

Algorithmus 3 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

1 Berechne eine zufällige Permutation $p_1, \ldots, p_n$ der Punkte in S
2 B $\leftarrow$ bounding box von $\{p_1, \ldots, p_4\}$
3 **if** *alle* p_i *sind in* B **then**
4   return B
5 **else**
6   return RandomizedBoundingBox(S)

Was ist die erwartete Laufzeit?

Algorithmus 4 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

1 Berechne eine zufällige Permutation $p_1, \ldots, p_n$ der Punkte in S
2 B $\leftarrow$ bounding box von $\{p_1, \ldots, p_4\}$
3 **if** *alle* p_i *sind in* B **then**
4 return B
5 **else**
6 return RandomizedBoundingBox(S)

Das Berechnen der Bounding Box $[\min_x, \max_x] \times [\min_y, \max_y]$ von vier Punkten $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$ liegt in $O(1)$

- $\min_x = \min\{x_1, x_2, x_3, x_4\}$
- $\min_y = \min\{y_1, y_2, y_3, y_4\}$
- $\max_x = \max\{x_1, x_2, x_3, x_4\}$
- $\max_y = \max\{y_1, y_2, y_3, y_4\}$

Algorithmus 5 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

1 Berechne eine zufällige Permutation $p_1, \ldots, p_n$ der Punkte in S
2 B $\leftarrow$ bounding box von $\{p_1, \ldots, p_4\}$
3 **if** *alle* p_i *sind in* B **then**
4 return B
5 **else**
6 return RandomizedBoundingBox(S)

In $O(1)$ kann überprüft werden ob ein Punkt (x, y) in einer bounding box $[\text{min}_x, \text{max}_x] \times [\text{min}_y, \text{max}_y]$ liegt:

- teste ob $\text{min}_x \leq x \leq \text{max}_y$
- teste ob $\text{min}_y \leq y \leq max_y$

Die Zeilen 1-4 haben also eine Laufzeit in $O(|S|)$

Algorithmus 6 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

```
1 Berechne eine zufällige Permutation p_1, ..., p_n der Punkte in S
2 B ← bounding box von {p_1, ..., p_4}
3 if alle p_i sind in B then
4     return B
5 else
6     return RandomizedBoundingBox(S)
```

- Damit ist die erwartete Laufzeit genau die erwartete Anzahl der Aufrufe von RandomizedBoundingBox(S) multipliziert mit $|S|$
- Wir berechnen hier nur die erwartete Anzahl der **rekursiven** Aufrufe

Algorithmus 7 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

```
1 Berechne eine zufällige Permutation p_1, ..., p_n der Punkte in S
2 B ← bounding box von {p_1, ..., p_4}
3 if alle p_i sind in B then
4     return B
5 else
6     return RandomizedBoundingBox(S)
```

- Der Algorithmus bricht ab, wenn die richtigen 4 Punkte gewählt wurden
- Wahrscheinlichkeit, dass die 4 Punkte richtig gewählt wurden ist

$$p := \frac{1}{\binom{|S|}{4}} = \frac{(|S|-4)! \cdot 4!}{|S|!}$$

Algorithmus 8 : RandomizedBoundingBox(S)

Eingabe : Menge S von Punkten in $\mathbb{R} \times \mathbb{R}$
Ausgabe : Bounding Box B von S

```
1 Berechne eine zufällige Permutation p_1, ..., p_n der Punkte in S
2 B ← bounding box von {p_1, ..., p_4}
3 if alle p_i sind in B then
4 |   return B
5 else
6 |   return RandomizedBoundingBox(S)
```

Die Wahrscheinlichkeit, dass sich RandomizedBoundingBox

0	mal selbst aufruft ist			p
1	mal selbst aufruft ist	$(1-p)$	$\cdot$	p
2	mal selbst aufruft ist	$(1-p)^2$	$\cdot$	p
...				
k	mal selbst aufruft ist	$(1-p)^k$	$\cdot$	p

Die Wahrscheinlichkeit, dass sich RandomizedBoundingBox

0	mal selbst aufruft ist			p
1	mal selbst aufruft ist	$(1-p)$	$\cdot$	p
2	mal selbst aufruft ist	$(1-p)^2$	$\cdot$	p
...				
k	mal selbst aufruft ist	$(1-p)^k$	$\cdot$	p

Damit ist die erwartete Anzahl an rekursiven Aufrufen E

$$E = \sum_{k=0}^{\infty} k \cdot p(1-p)^k = p \cdot \sum_{k=0}^{\infty} k(1-p)^k$$

Lösungsweg 1 (empfohlen):

$$
\begin{aligned}
E &= \sum_{k=0}^{\infty} pk \cdot (1-p)^k = p \cdot \sum_{k=0}^{\infty} k\underbrace{(1-p)}_{q}^{k} = p \cdot \sum_{k=0}^{\infty} kq^k \\
&= pq \cdot \sum_{k=0}^{\infty} kq^{k-1} = pq \cdot \left(\sum_{k=0}^{\infty} q^k\right)' = pq\left(\frac{1}{1-q}\right)' \\
&= \frac{pq}{(1-q)^2} = \frac{1-p}{p}
\end{aligned}
$$

Lösungsweg 2:

$$E = \sum_{k=0}^{\infty} pk \cdot (1-p)^k = p \cdot \sum_{k=0}^{\infty} k\underbrace{(1-p)}_{q}^k = p \cdot \underbrace{\sum_{k=0}^{\infty} kq^k}_{T}$$

$$\begin{aligned} T(1-q) &= \sum_{k=0}^{\infty} kq^k - \sum_{k=0}^{\infty} kq^{k+1} \\ &= \sum_{k=0}^{\infty} kq^k - \sum_{k=1}^{\infty} (k-1)q^k \\ &= \sum_{k=0}^{\infty} q^k - 1 = \frac{1}{1-q} - 1 = \frac{1-(1-q)}{1-q} = \frac{q}{1-q} \\ \Longrightarrow T &= \frac{q}{(1-q)^2} = \frac{1-p}{p^2} \\ \Longrightarrow E &= \frac{1-p}{p} \end{aligned}$$

- Die erwartete Zahl der rekursiven Aufrufe ist $\frac{1-p}{p}$
- Die erwartete Zahl an Aufrufen ist damit $\frac{1-p}{p} + 1 = \frac{1}{p} = \frac{|S|!}{|S-4|!4!} \in O(|S|^4)$
- Die erwartete Laufzeit ist damit in $O(|S|^5)$

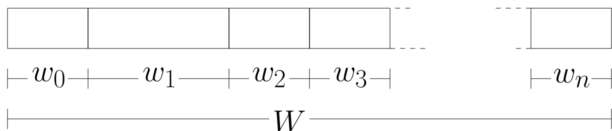
Aufgabe 2 - Zufälliges Element - nicht gleichverteilt

- Sei $S = \{x_1, \ldots, x_n\}$ eine Menge von Elementen mit Gewichten $w_1, \ldots, w_n$.
- Es sei $W := \sum_{i=1}^{n} w_i$.

Aufgabe

- Beschreibe einen Algorithmus, der ein zufälliges Element x_i aus S mit Wahrscheinlichkeit w_i/W wählt.
- Die Laufzeit des Algorithmus soll in $O(n)$ liegen.
- Dazu darf eine Routine random(a, b) benutzt werden, die in Zeit $O(1)$ eine ganze Zahl gleichverteilt aus der Menge $\{a, a+1, \ldots, b\}$ wählt.

Idee:



Algorithmus 9 : Ein einfacher, aber langsamer Ansatz

Eingabe : Gewichte $w_1, \ldots, w_n$
Ausgabe : Index des zufällig gewählten Elements

```
1  W ← 0
2  for i=1 to n do
3  |  W ← W + w_i
4  R ← random(0,W)
5  X ← 1
6  S ← w_1
7  while R > S do
8  |  X ← X + 1
9  |  S ← S + w_X
10 RETURN X
```

Leicht verändertes Setting

- In der Praxis sind häufig häufig nur rationale Zufallszahlen im Bereich $[0, 1]$ verfügbar
  (mit der Genauigkeit des benutzen Datentyps)
- Ab jetzt betrachten wir immer einen solchen Zufallszahlengenerator $\textsc{Random}[0,1]$
- Anstatt Gewichten haben wir Wahrscheinlichkeiten $p_1, \ldots, p_n$

Algorithmus 10 : Ein einfacher, aber langsamer Ansatz

Eingabe : Wahrscheinlichkeiten $p_1, \ldots, p_n$
Ausgabe : Index des zufällig gewählten Elements

```
1 R ← Random([0,1])
2 X ← 1
3 S ← p_1
4 while R > S do
5     X ← X + 1
6     S ← S + p_X
7 RETURN X
```

Algorithmus 11 : bessere average-case Laufzeit

Eingabe : Wahrscheinlichkeiten $p_1, \dots, p_n$

Ausgabe : Index des zufällig gewählten Elements

1 Vorberechnung: Sortiere die p_i so um, dass $p_1 \geq p_2 \geq \dots \geq p_n$

2 $R \leftarrow \textsc{Random}([0,1])$

3 $X \leftarrow 1$

4 $S \leftarrow p_1$

5 **while** $R > S$ **do**

6 $\quad X \leftarrow X + 1$

7 $\quad S \leftarrow S + p_X$

8 RETURN X

Algorithmus 12 : mehr Vorberechnung

Eingabe : Wahrscheinlichkeiten $p_1, \ldots, p_n$
Ausgabe : Index des zufällig gewählten Elements

```
1  // Vorberechnung:
2  S[0] ← 0
3  for i = 1 to n do
4      S[i] ← S[i − 1] + p_i
5  R ← Random([0,1])
6  X ← 1
7  S ← p_1
8  while R > S[X] do
9      X ← X + 1
10 RETURN X
```

Algorithmus 13 : mehr Vorberechnung und binäre Suche

Eingabe : Wahrscheinlichkeiten $p_1, \ldots, p_n$
Ausgabe : Index des zufällig gewählten Elements

1 // Vorberechnung:
2 $S[0] \leftarrow 0$
3 **for** $i = 1$ **to** n **do**
4 $\quad S[i] \leftarrow S[i-1] + p_i$

5 $R \leftarrow \textsc{Random}([0,1])$

6 $X \leftarrow$ Binäre Suche in $S[]$ nach X mit $S[X-1] \leq R \leq S[X]$

7 RETURN X

- Anstatt eine binäre Suche auf dem Array $S[]$ auszuführen, kann man auch einen binären Suchbaum T verwenden
- Blätter von T korrespondieren zu je einem Element in $\{1, \ldots, n\}$
- Jedem Nichtblatt v in T ist eine Zahl $r[v] \in [0, 1]$ zugeordnet

Algorithmus 14 : **Mit binärem Suchbaum** T

Eingabe : Binärer Suchbaum T
Ausgabe : Index des zufällig gewählten Elements

```
1 R ← Random([0,1])
2 n ← Wurzel von T
3 while n ist kein Blatt von T do
4     if R > r[n] then
5         n ← linker Nachfolger von n
6     else
7         n ← rechter Nachfolger von n
8 RETURN X
```

Wie bekommt man einen solchen binären Suchbaum?

- Möglichkeit 1: Vollständiger binärer Baum mit n Blättern
- Möglichkeit 2: Huffmann-Tree

Aufgabe 3: Gegeben ist folgender Monte-Carlo Algorithmus

Algorithmus 15 : Überprüfe ob Bitfolgen gleich sind

Eingabe : Bitfolgen $a_1 \dots a_n$ und $b_1 \dots b_n$

Ausgabe : Entscheidung ob Bitfolgen gleich sind

1 $p \leftarrow$ Primzahl, zufällig gleichverteilt aus denen $\leq n^2 \log n^2$ gewählt

2 $a \leftarrow \sum_{i=1}^{n} a_i 2^{i-1}$

3 $b \leftarrow \sum_{i=1}^{n} b_i 2^{i-1}$

4 **if** $a \mod p = b \mod p$ **then**

5 **return** JA

6 **else**

7 **return** NEIN

Zeigen Sie, dass die Wahrscheinlichkeit, dass der Algorithmus ein falsches Ergebnis liefert in $O(1/n)$ liegt.

Benutzen Sie dafür die folgenden beiden Resultate:

Satz Chebyshew (1)

Für die Anzahl $\pi(n)$ der Primzahlen kleiner oder gleich n gilt

$$\frac{7}{8}\frac{n}{\ln n} \leq \pi(n) \leq \frac{9}{8}\frac{n}{\ln n}$$

Es ist also $\pi(n) \in \Theta(n/\ln n)$.

Satz (2)

Die Anzahl k verschiedener Primteiler einer Zahl kleiner oder gleich 2^n ist höchstens n.

Motivation

- Klar: Zwei Bitfolgen lassen sich leicht deterministisch auf Gleichheit überprüfen
- Vorstellung: Bitfolgen $a_1 \ldots a_n$ und $b_1 \ldots b_n$ liegen an verschiedenen Orten
- Aufgabe: Überprüfe auf Gleichheit aber übertrage möglichst wenige Daten
- Hilfsmittel: Fingerprints

Algorithmus 16 : Überprüfe ob Bitfolgen gleich sind

Eingabe : Bitfolgen $a_1 \dots a_n$ und $b_1 \dots b_n$

Ausgabe : Entscheidung ob Bitfolgen gleich sind

1 $p \leftarrow$ Primzahl, zufällig gleichverteilt aus denen $\leq n^2 \log n^2$ gewählt

2 $a \leftarrow \sum_{i=1}^{n} a_i 2^{i-1}$

3 $b \leftarrow \sum_{i=1}^{n} b_i 2^{i-1}$

4 **if** $a \mod p = b \mod p$ **then**

5   **return** JA

6 **else**

7   **return** NEIN

- Falls $a = b$ liefert der Algorithmus immer das richtige Ergebnis
- Falls $a \neq b$ können Fehler auftreten

Algorithmus 17 : Überprüfe ob Bitfolgen gleich sind

Eingabe : Bitfolgen $a_1 \dots a_n$ und $b_1 \dots b_n$

Ausgabe : Entscheidung ob Bitfolgen gleich sind

1 $p \leftarrow$ Primzahl, zufällig gleichverteilt aus denen $\leq n^2 \log n^2$ gewählt
2 $a \leftarrow \sum_{i=1}^{n} a_i 2^{i-1}$
3 $b \leftarrow \sum_{i=1}^{n} b_i 2^{i-1}$
4 **if** $a \mod p = b \mod p$ **then**
5   **return** JA
6 **else**
7   **return** NEIN

- $F_p(x) : \begin{cases} \mathbb{Z} & \rightarrow & \mathbb{Z}_p = \{0, 1, 2, \dots, p-1\} \\ x & \mapsto & x \mod p \end{cases}$
- Der Algorithmus überprüft, ob $F_p(a) = F_p(b)$ und liefert falsche Antwort, falls $F_p(a) = F_p(b)$ und $b \neq a$
- Algorithmus liefert falsche Antwort, falls p den Wert $c = |b - a|$ teilt und $a \neq b$

- Algorithmus liefert falsche Antwort, falls p den Wert $c = |b - a|$ teilt und $a \neq b$
- also $\mathbb{P}\{\text{falsche Antwort}\} = \frac{\text{Anzahl Primzahlen } p \text{ mit } p|c}{\text{Anzahl Primzahlen kleiner } n^2 \log n^2}$

- Algorithmus liefert falsche Antwort, falls p den Wert $c = |b - a|$ teilt und $a \neq b$
- also $\mathbb{P}\{\text{falsche Antwort}\} = \frac{\text{Anzahl Primzahlen } p \text{ mit } p|c}{\text{Anzahl Primzahlen kleiner } n^2 \log n^2}$

Anzahl Primzahlen p mit $p|c$

- a und b besitzten n bits $\Rightarrow c = |a - b| \leq 2^n$
- Lemma: Die Anzahl k verschiedener Primteiler einer Zahl kleiner oder gleich 2^n ist höchstens n.
- c hat höchsten n Primteiler

- Algorithmus liefert falsche Antwort, falls p den Wert $c = |b - a|$ teilt und $a \neq b$
- also $\mathbb{P}\{\text{falsche Antwort}\} = \frac{\text{Anzahl Primzahlen } p \text{ mit } p|c}{\text{Anzahl Primzahlen kleiner } n^2 \log n^2}$

Anzahl Primzahlen kleiner $n^2 \log n^2$

- Satz Chebyshev: Es ist $\pi(t) \in \Theta(t/\ln t)$
- Im Bereich $[2, n^2 \ln n^2]$ gibt es

$$\pi(n^2 \ln n^2) \in \Theta(n^2 \ln n^2 / \ln(n^2 \ln n^2))$$

Primzahlen

- Algorithmus liefert falsche Antwort, falls p den Wert $c = |b - a|$ teilt und $a \neq b$
- also $\mathbb{P}\{\text{falsche Antwort}\} = \frac{\text{Anzahl Primzahlen } p \text{ mit } p|c}{\text{Anzahl Primzahlen kleiner } n^2 \log n^2}$

Anzahl Primzahlen p mit $p|c$: $\leq n$

Anzahl Primzahlen kleiner $n^2 \log n^2$: $\Theta(n^2 \ln n^2 / \ln(n^2 \ln n^2))$

$$\begin{aligned}
\mathbb{P}\{\text{falsche Antwort}\} &\in O\left(\frac{n}{n^2 \ln n^2 / \ln(n^2 \ln n^2)}\right) \\
&= O\left(\frac{\ln(n^2 \ln n^2)}{n \ln n^2}\right) \\
&= O\left(\frac{1}{n} + \frac{\ln(\ln(n^2))}{n \ln(n^2)}\right) \\
&= O\left(\frac{1}{n}\right)
\end{aligned}$$

ILP-Modellierung: Das Facility Location Problem

Gegeben:

- Eine Menge $N = \{1, \ldots, n\}$ an möglichen Stützpunkten
- Eine Menge $M = \{1, \ldots, m\}$ an Kunden
- Für jeden möglichen Stützpunkt k: Kosten c_k, falls der Stützpunkt eröffnet wird
- Für jeden möglichen Stützpunkt k: Maximale Anzahl an Kunden s_k die k versorgen kann
- Für jeden Stützpunkt i und jeden Kunden j: Kosten h_{ij} falls der Kunde j von Stützpunkt i versorgt wird

Gesucht:

- Eine Menge $N' \subseteq N$ an Stützpunkten, so dass die Gesamtkosten um alle Kunden zu versorgen minimal ist.

Variablen:

- $x_i = \begin{cases} 1 & \text{, Stützpunkt } i \text{ wird eröffnet} \\ 0 & \text{, sonst} \end{cases}$
- $y_{ij} = \begin{cases} 1 & \text{, Kunde } j \text{ wird durch Stützpunkt } i \text{ versorgt} \\ 0 & \text{, sonst} \end{cases}$

Minimierungsfunktion:

$$\min \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{m} h_{ij} y_{ij}$$

Constraints:

$$\sum_{i \in N} y_{ij} = 1 \quad \text{für alle } j \in M$$

$$y_{ij} - x_i \leq 0 \quad \text{für alle } i \in N \text{ und } j \in M$$

$$\sum_{i \in M} y_{ij} \leq s_i \quad \text{für alle } i \in N$$