Grundlagen Theoretischer Informatik 3

SoSe 2010 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Grundlagen Theoretischer Informatik 3 Gesamtübersicht

- Organisatorisches; Einführung
- Algorithmenanalyse: Komplexität und Korrektheit
- Zur Behandlung NP-schwerer Probleme: Ausblicke
- Randomisierte Algorithmen und ihre Analyse

Randomisierte Algorithmen

- Keine formale Einführung entsprechender Komplexitätsklassen
- Beispiele im Polynomzeitbereich:
  Rabins Primzahltest
  Quicksort
- Randomisierte Approximation
- Randomisierte exakte Algorithmen für harte Probleme

Randomisierte Algorithmen: Was ist die Idee?

Während der Durchführung der Berechnungen darf “gewürfelt” werden, formaler hat also eine (Turing-)Maschine Zugriff auf ein Zufallsbitorakel, z.B. bereitgestellt auf einem Extra-Band, das nur in einer Richtung gelesen werden darf.

Die Algorithmen kann man danach klassifizieren, ob sie stets die richtige Antwort liefern, ob sie nur bei JA (o.B.d.A.) evtl. “sich täuschen” oder aber zwar nie lügen, aber evtl. (mit kleiner Wahrsch.) nicht halten.

Das kann dazu führen, dass schlechte Fälle unwahrscheinlich werden.

Miller-Rabin-Test (Quelle: Wikipedia)

Sei n eine ungerade Zahl, von der festgestellt werden soll, ob sie prim ist oder nicht. Zuerst wählt man eine Zahl a zufällig aus den Zahlen $2, 3, \ldots, n-1$.

Der nächste Schritt benutzt einen Test, den nur Primzahlen und sogenannte starke Pseudoprimzahlen bestehen können.
Hierzu wird $n-1$ eindeutig in seinen ungeraden und geraden Anteil faktorisiert: $n-1 = d \cdot 2^j$.

Wenn $a^d \equiv 1 \mod n$ ist oder $a^{d \cdot 2^r} \equiv -1 \mod n$ für ein r mit $0 \leq r \leq j-1$ gilt, dann ist n entweder eine Primzahl oder eine starke Pseudoprimzahl.

Miller-Rabin-Test (Forts.) Der Miller-Rabin-Test berechnet modulo n die Folge

$$(a^d, a^{2d}, a^{4d}, \ldots, a^{2^{j-1}d}, a^{2^j d})$$

mit $2^j d = n - 1$.

Die einzigen Zahlen x mit $x^2 \equiv 1 \pmod{n}$ sind für eine Primzahl n die Zahlen 1 und -1.

$\rightsquigarrow$ Für eine Primzahl n ist die Folge des Miller-Rabin-Tests entweder $(1, 1, \ldots, 1)$ oder $(?, ?, \ldots, ?, -1, 1, \ldots, 1)$ ist, wobei die Fragezeichen für beliebige Zahlen stehen.

Miller-Rabin-Test (Forts.)

Wird a richtig gewählt, so lässt sich die (angenommene) Nicht-Primalität von n sicher feststellen.
Eine Zahl, die der Miller-Rabin-Test als “prim” klassifiziert, ist aber nur “wahrscheinlich prim”.
Die Wahrscheinlichkeit dafür, dass n zusammengesetzt ist und ein zufälliges a dafür kein Zeuge ist, ist kleiner als $\frac{1}{4}$.
Wiederholt man den Test mehrfach für verschiedene a aus $[2, n-2]$, so sinkt die Wahrscheinlichkeit weiter ab.

Es gibt auch deterministische Varianten dieses Tests.
Wenn $n < 341.550.071.728.321$, ist es ausreichend, $a = 2, 3, 5, 7, 11, 13$, und 17 zu testen !

Randomisierte Algorithmen: Wozu?

Der Sinn des Würfelns sollte aus den folgenden Beispielen klar werden.

Ziel ist meist: Größere Geschwindigkeit.

Dass hierbei Würfeln hilft, ist vermutlich zunächst kontraintuitiv.

Quicksort als Las-Vegas-Algorithmus

Gegeben Feld $a[1], ..., a[n]$, Ziel: rekursive Sortierfunktion qs:

- $qs(i, j)$ sortiert Teilfeld $a[i]...a[j]$, d.h. Aufruf mit $qs(1, n)$
- $qs(i, j)$ arbeitet wie folgt:

  (1) Wähle k mit $i \leq k \leq j$

  (2) Modifiziere $a[i]...a[j]$ so, dass $a[k]$ an eine Stelle m kommt mit

$$(\forall \nu, i \leq \nu \leq m) \quad a[\nu] \leq a[m]$$
$$(\forall \nu, m \leq \nu \leq j) \quad a[m] \leq a[\nu]$$

  (3) Falls $i < m-1$: Rekursiver Aufruf von $qs(i, m-1)$

  (4) Falls $m+1 < j$: Rekursiver Aufruf von $qs(m+1, j)$

- Übliche, deterministische Setzungen bei (1):

  (1a) Wähle $k = i$

  (1b) Wähle $k = \lfloor \frac{i+j}{2} \rfloor$

- Resultat: Mittlere Laufzeit ist $\mathcal{O}(n \log n)$ (gemittelt über alle möglichen Felder und im Einheitskostenmaß)
- Es gibt Felder mit Worst-Case-Verhalten von $\mathcal{O}(n^2)$.
- Problem: Kommen bei der Anwendung nicht alle Felder mit gleicher Wahrscheinlichkeit vor, greift die Mittelwertanalyse nicht!
- Ausweg: Probabilistischer Algorithmus mit

  (1c) Wähle k zufällig aus $\{i, ..., j\}$

- Dann gilt offensichtlich:

  1. Der Algorithmus hält immer.
  2. Er liefert stets das gleiche Resultat (das sortierte Feld).

- Durchschnittliche Laufzeit mit (1c) sofort:

$$T^{\emptyset}_{qs}(n) = \underbrace{c \cdot n}_{(1c),(2)} + \frac{1}{n} \sum_{m=1}^{n} \Big(T^{\emptyset}_{qs}(m-1) + T^{\emptyset}_{qs}(n-m-1) \Big)$$

  wieder mit Lösung $T^{\emptyset}_{qs}(n) = \mathcal{O}(n \log n)$, 'Mittelwertanalyse' unnötig!

Anmerkungen

- Sortierzeit bei gegebenem Feld nicht mehr konstant, sondern zufällig...
- Mittlere Laufzeit gilt jetzt bei jedem Feld!
- Achtung: Worst-Case könnte jetzt auch bei jedem Feld eintreten...
- Weitere Verbesserungen z.B.:

(1d) Wähle drei Werte k_1, k_2, k_3 zufällig aus $\{i, ..., j\}$,
setze k:=Index des mittleren Wertes aus $\{a[k_1], a[k_2], a[k_3]\}$

Dann tritt der schlimmste Fall seltener ein, ist aber immer noch nicht ausgeschlossen!

Analoge Vorgehensweise ist bei vielen Algorithmen möglich, die Mengen elementweise verarbeiten:

- Statt determinierter Auswahl eines Element wähle zufällig aus...

Vorteile der Randomisierung:

- (Komplexitäts-)Analyse des randomisierten Algorithmus ist oft signifikant leichter
- keine umfangreichen Wahrscheinlichkeitsuntersuchungen und -annahmen über die Menge aller Eingaben
- zufällige Auswahl ermöglicht Mittelwertuntersuchungen
- unbekannte Verteilung der Eingabe bei det. Algorithmen erlaubt i.A. nur Worst-Case-Analyse
- Nach ‘Randomisierung’ eines Algorithmus evtl. erstmals Average-Case-Analyse möglich
- Bei randomisierten Algorithmen sind alle Eingaben gleich gut (bzw. gleich schlecht).
- Der Anwender muss sich nicht mehr darum kümmern, dass seine Eingaben den Mittelwert-bedingungen genügen.

Nochmal: **Greedy-Approximation für das Knotenüberdeckungsproblem**

Kleine (randomisierende) Änderung:

Wähle Knoten v mit Wahrscheinlichkeit $p(v) = \frac{\delta(v)}{2|E|}$.

Hinweis: $\sum_{v \in V} \delta(v) = 2|E|$ durch doppeltes Abzählen bei Inzidenzmatrix (Tafel).

Es sei C^* eine optimale Überdeckung.

Der Erwartungwert für die Anzahl der Knoten aus C^*, die der Algorithmus zufällig wählt, ist:

$$\sum_{v \in C^*} p(v) = \sum_{v \in C^*} \frac{\delta(v)}{2|E|} \geq \frac{1}{2},$$

denn jede Kante wird mindestens einmal abgedeckt. Daher ist der Algorithmus "wahrscheinlich" nach $2|C^*|$ Schritten fertig und hat dann auch eine Faktor-2-Approximation gefunden.

Näheres hierzu zeigen (hier weggelassene) Induktionsbeweise, siehe VL Näherungsalgorithmen (Approximationsalgorithmen).

MAX-2-SAT

Aufgabe: Maximiere durch Finden einer geeigneten Belegung der Booleschen Variablen $\{x_1, \ldots, x_n\}$ der vorliegenden 2SAT-Formel die Zahl der erfüllten 2-Klauseln (aus m vielen).

Idee: Entscheide jedes Belegungs-Bit durch fairen Münzwurf.

Jede Klausel wird so mit Wahrsch. $\frac{3}{4}$ erfüllt.

Daher ist der Erwartungswert für die Zufallsgröße "Anzahl erfüllter Klauseln" $\frac{3}{4} \cdot m$.

Da nicht mehr als m Klauseln erfüllt werden können, liefert dies eine randomisierte $\frac{4}{3}$-Approximation.

Analoges Argument für randomisierte $\frac{8}{7}$-Approx. für MAX-3-SAT.

Schönings 3-SAT Algorithmus

Eingabe: $3-$KNF Formel F mit n Booleschen Variablen.
Ausgabe: Erfüllende Belegung A oder NEIN.

Wiederhole $(4/3)^n$-mal:

1. Wähle zufällige Belegung A. Setze $w \leftarrow 0$.
2. Falls A erfüllbar ist: liefere A zurück und halte.
3. Wähle irgendeine unerfüllte Klausel C in F.
4. Wähle zufällig eine Variable x in C und ändere A (nur) bei x.
5. $w \leftarrow w + 1$. Falls $w < 2n^2$, gehe zu 2.

Gib NEIN aus.

Das ist ein Beispiel für einen Random-Walk Algorithmus mit mehrfachem Neustart.
Mit einer Fehlerwahrsch. < 1 löst Schönings Algorithmus die 3-SAT Instanz.
Ein JA ist dabei immer richtig, ein NEIN kann falsch sein.

Schönings 3-SAT Algorithmus

Einige intuitive Erläuterungen

—Es ist nicht sinnvoll, den Random-Walk “zu lange” durchzuführen:
Ist man nämlich “nahe dran”, so ist die Wahrsch. ziemlich groß,
in die falsche Richtung zu laufen.

—Die Neustarts ermöglichen es, mit großer Wahrscheinlichkeit oft nahe an eine Lösung zu kommen, die der Random-Walk dann auch mit gewisser Wahrsch. finden kann.

Literatur: U. Schöning: A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica* 32:615–623, 2002.

Exakte Algorithmen für 3-SAT

Folgende Beobachtung gestattet einen deterministischen Alg. mit Laufzeit $\mathcal{O}(1.912^n)$:

Wähle Belegung Klausel C beliebig.
Es gibt genau eine (aus 8) Belegungen der in C vorkommenden drei Variablen, die C falsch machen. Verzweige nach den verbleibenen sieben Möglichkeiten.

Das führt auf einen Suchbaum mit der Rekursion: $T(n) \leq 7T(n-3)$ für dessen Blattanzahl.

Man prüfe nach, dass diese Ungleichung für $T(n) = 1.912^n$ erfüllt wird, denn $1.912^3 \leq 6.99$.

Die vorher angedeutete Idee der lokalen Suche gestattet det. Alg. mit Laufzeit $\mathcal{O}^*(1.5^n)$, siehe:
Theoretical Computer Science 289:69–83, 2002.